JAN

Ein Visualisierungssystem zur objektorientierten Animation von Java Programmen

André Vratislavsky

Vorwort

Der vorliegende Textauszug und das darin dokumentierte Visualisierungssystem gehen auf die Diplomarbeit des Autors (jan@vratislavsky.de) bei Prof. Dr.-Ing. Klaus-Peter Löhr (lohr@inf.fu-berlin.de) am Fachbereich Mathematik und Informatik des Instituts für Informatik (http://www.inf.fu-berlin.de) an der Freien Universität Berlin zurück. Die Diplomarbeit wurde im August 2002 fertiggestellt.

Im Rahmen der Präsentation auf dem fachwissenschaftlichen Kongress *Informatiktage 2002* (http://www.informatiktage.de) wurde das Visualisierungssystem *JAN* getauft.

Das System ist auf JAN's Internetseiten dokumentiert:

http://www.inf.fu-berlin.de/~vratisla/Jan/Jan.html

Oktober 2003

Inhaltsverzeichnis

1 Überblick

Das vorliegende Dokument befasst sich mit dem Entwurf und der Implementierung von *JAN*, einem Visualisierungssystem zur objektorientierten Animation von Java Programmen. JAN kann den Ablauf eines beliebigen Java Programmes in animierter Form, d.h. zeitabhängig visualisieren. Insbesondere werden dabei Datenstrukturen und die Kommunikation zwischen den Objekten auf Ebene der Methodenaufrufe dargestellt.

Voraussetzung ist das Vorhandensein des Quellcodes des zu inspizierenden Programmes. Dieser wird analysiert und (vor dem Anwender versteckt) um speziellen Code für die Visualisierung erweitert.

Damit nicht alle programmtechnischen Details dargestellt werden, gibt es die Möglichkeit, die gewünschten bzw. ungewünschten Details im Programm mit sogenannten Tags in Kommentarform zu kennzeichnen. Dafür kommen insbesondere Methodenaufrufe, Schleifendurchläufe, Unterobjekte und Referenzen in Frage. Diese Erweiterung unterscheidet das System von einem gewöhnlichen Debugger.

Eine besondere Bedeutung hat die Datenstruktur der Objekte. In Java ist im Programmtext nicht zu erkennen, ob eine Variable auf ein Unterobjekt zeigt, welches semantisch als Komponente zu betrachten ist, eine beliebige Referenz darstellt oder nur die Funktion einer Hilfsvariable hat. Das Visualisierungssystem ermöglicht, Variablen im Java Programm mit speziellen Kommentaren um eine semantische Eigenschaft zu ergänzen, die sie als *Komponenten* kennzeichnen. Eine Komponente ist in diesem Zusammenhang ein Unterobjekt, welches als Teil eines größeren Objekts betrachtet wird und im Gegensatz zu einfachen Referenzen semantisch eng zu diesem gehört. In der objektorientierten Welt wird dieses Verhältnis auch als Aggregation bezeichnet. Es ist wahlweise auch eine automatische Erkennung von Komponenten vorgesehen, z.B. für Felder, die `final` oder `private` deklariert sind.

Die Schnittstelle zum Anwender wird durch eine graphische Benutzeroberfläche (GUI) realisiert, mit der das zu inspizierende Programm gesteuert werden kann und die die gewünschten Informationen graphisch darstellt. Dabei werden neben textuellen Darstellungen Ablauf- und Objektdiagramme verwendet. Der Inhalt der Diagramme wird nicht nur durch die Vorauswahl bestimmt, die durch die Kommentare im Programmtext getroffen werden kann, sondern kann der Übersichtlichkeit wegen interaktiv reduziert werden.

Die Hauptidee ist zunächst die Animation einer Software. Nah verwandt damit ist das Gebiet der Animation von Algorithmen, bei der die Aufmerksamkeit mehr auf der abstrakten Darstellung eines Algorithmus als auf der Funktionsweise eines Programmes liegt. Eine Algorithmenanimation kann mit dem Visualisierungssystem vorgenommen werden, indem ein entsprechendes Programm geschrieben wird, das den Algorithmus realisiert. Der Schwerpunkt liegt dabei aber nicht auf einer speziell für den Algorithmus vorgesehenen graphischen Darstellung, sondern auf der Präsentation von Variablen und Methodenaufrufen. Die dargestellten Details und ihre Anordnung in Graphiken lassen sich dennoch von Hand vielfältig gestalten.

Die Methodenaufrufe im inspizierten Programm werden nach den Threads unterschieden, zu denen sie gehören. Für alle Threads gibt es einen Methodenaufrufstack und eine Historie der Aufrufe. Die Aufrufe aller Threads werden gemeinsam in einem Ablaufdiagramm dargestellt und farblich getrennt gekennzeichnet, so dass der Ablauf nebenläufiger Programme dargestellt werden kann.

2 Vorüberlegungen

2.1 Grundsätzliches zum Thema Animation

In einem dynamischen System, wie einem laufenden Programm, kann eine Folge von Zuständen festgestellt werden. Das Visualisierungssystem soll aus dieser Folge von Zuständen eine Abfolge von Bildern, also eine Animation herstellen.

Das folgende Beispiel enthält einen über die Zeit veränderlichen Wert. Im Diagramm (**Abbildung 1**) sind vier aufeinander folgende Messwerte durch blaue Balken dargestellt . Eine alternative Darstellung würde z.B. eine Kurve durch die Punkte der zweidimensionalen Graphik legen. Dabei könnten die Punkte direkt verbunden werden (hier in Rot ausgeführt), oder es könnte eine ausgleichende Kurve (z.B. ein *B-Spline*) durch die Punkte gelegt werden. Eine Kurve, insbesondere eine „runde“ Kurve[1], ist sicherlich eine geeignetere Darstellung für einen zeitlichen Vorgang als ein Balkendiagramm.

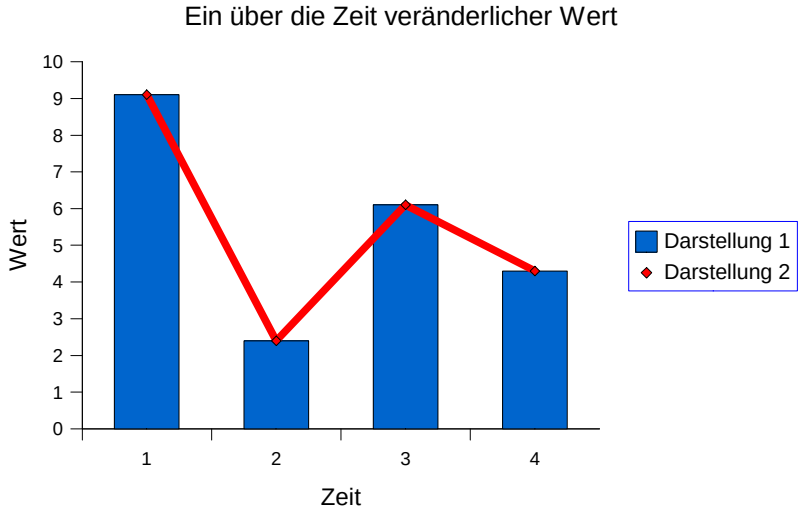


Abbildung 1 Theoretische Betrachtung eines über die Zeit veränderlichen Wertes.

In einer anderen Darstellung könnte der Wert durch einen einzigen über die Zeit veränderlichen Balken präsentiert werden. Je nach Darstellungsart ergeben sich folgende Möglichkeiten:

1. Wenn der Balken entsprechend der blauen Balkendarstellung animiert wird, ändert sich seine Größe mit der Zeit sprungartig.
2. Eine Animation gemäß der roten Kurve lässt den Balken kontinuierlich wachsen und schrumpfen. In den Umkehrpunkten zwischen Wachstum und Schrumpfung ändert sich dieser Vorgang spontan.
3. Bei einer Animation, die einer Darstellung mit einem *Spline* im Zeit-Wert-Diagramm entspricht, wächst und schrumpft der Balken kontinuierlich ohne plötzliche Geschwindigkeitsänderungen.

Die dritte Alternative entspricht sicherlich am ehesten einem Film, d.h. die harmonischen Änderungen im Bild werden als angenehm empfunden. Das bedeutet aber nicht, dass die Darstellung deswegen für die Visualisierung eines technischen Ablaufs besonders geeignet ist. Die zweite Darstellung liefert ein akzeptables Ergebnis, wohingegen die erste Möglichkeit sehr verwackelt erscheint. Bei sprunghaften Änderungen besteht zudem die Gefahr, dass der Beobachter die Änderungen nicht wahrnimmt.

1 'rund' bedeutet, dass an allen Stellen der zugrunde liegenden Funktion die erste Ableitung existiert.

Nun kann die Frage gestellt werden, inwieweit die aufgeführten Darstellungen durch die Wirklichkeit gerechtfertigt sind. Angenommen,

- von dem System sind nur die vier Messpunkte bekannt oder,
- noch schärfer formuliert, innerhalb der gemessenen Zeit existieren nur die vier Zustände, d.h. zwischen den Messpunkten liegen gar keine unbekannten, nicht ermittelten Werte.

Im ersten Fall würden die beiden letzten Darstellungen Zustände präsentieren, die nicht ermittelt wurden, das heißt deren Werte möglicher Weise falsch sind! Im zweiten Fall würden gar Werte angezeigt, die überhaupt nicht vorhanden sein könnten.

Bei der Wahl der Animation ist also zu berücksichtigen, ob es sich um diskrete Größen handelt und ob das Risiko eingegangen werden soll, falsch interpolierte Werte darzustellen. Es stehen sich zwei Positionen gegenüber:

Animation ohne interpolierte Zwischenwerte	**Animation mit Zwischenwerten**
Plötzliche sprunghafte Änderungen	Harmonische Bewegungen
Wertänderungen fallen u.U. nicht auf	Ablauf gut verfolgbar
	Hoher Programmieraufwand
	Möglicherweise Performanceprobleme
Unverfälschte Abbildung der Wirklichkeit	Verfremdung unterstützt das Verständnis
Technisch realistische Darstellung	Präsentation mit künstlerischem Anspruch

Tabelle 1 Vor- und Nachteile der Animation mit und ohne interpolierte Zwischenwerte.

Das Visualisierungssystem soll eine wirklichkeitsnahe, unverfälschte Abbildung des realen Ablaufs eines Programmes erreichen. Eine Animation, die ohne die Darstellung von interpolierten Werten auskommt, ist dafür eine geeignete Lösung.

2.2 Anwendungen mit verwandter Problematik: Debugger

Es gibt Debugger wie z.B. JSWAT [JSWAT] oder der Debugger der *JBuilder* [JBUILDER] Entwicklungsumgebung, die den Zustand der beobachteten Objekte in Fenstern darstellen. Der Inhalt der Fenster besteht im Wesentlichen aus Textzeilen und Tabellen und gegebenenfalls Objektbäumen. Dargestellt werden z.B. Quelltext, Objekte und ihre Unterobjekte, Variablen, Stacks für Methodenaufrufe, Threads, eine Ausgabe des Programmes und Breakpoints zum Steuern des Debuggers. Mit Hilfe von Buttons kann z.B. bestimmt werden, ob der Debugger eine Methode ohne Pause abarbeiten oder nur bis zum nächsten Breakpoint innerhalb dieser Methode laufen soll.

Der Inhalt der dargestellten Fenster basiert also i.d.R. auf Text. Bei der *Together* Entwicklungsumgebung [TOGETHER] gibt es die Möglichkeit, Klassen- und Ablaufdiagramme (siehe **Abbildung 3** auf Seite 11) statisch aus dem Quelltext zu generieren und anzeigen zu lassen.

Debugger haben in der Regel keinen besonderen Anspruch an die graphische Darstellung von Objekten und Abläufen. Das Visualisierungssystem soll neben einer intuitiven Graphik zudem auch die Möglichkeit haben, spezielle Annotationen im Quelltext auswerten zu können, über die der dargestellte Inhalt definiert werden kann. Darin soll sich das Visualisierungssystem von einem gewöhnlichen Debugger unterscheiden.

2.3 Das *BARAT* Framework

Es besteht zum einen der Anspruch, spezielle Annotationen im Quelltext auswerten zu können, zum anderen müssen Informationen aus dem laufenden Programm hinaus geschleust werden.

Das Framework *Barat* wurde von Boris Bokowski [BARAT] entwickelt. *Barat* kann Java Quelltext statisch analysieren und dabei einen abstrakten Syntaxbaum aufbauen, der eine Java Klasse repräsentiert. Innerhalb des Baums kann auf Vorgänger Knoten zugegriffen werden. So kann z.B. nach der Methode gefragt werden, in der sich eine bestimmte Anweisung befindet. Desweiteren kann auf Kommentare bestimmter Form, die sogenannten *Tags*, zugegriffen werden. Der

Syntaxbaum kann wieder in Form von Java Quelltext in eine Datei ausgegeben werden. *Barat* wird dazu verwendet, Schnittstellen-Aufrufe in den Quelltext der zu visualisierenden Programme einzufügen, die Informationen aus dem Programm an die Visualisierung liefern. Das modifizierte Programm muss dann vor der Ausführung zunächst neu übersetzt werden.

Annotationen in Form von speziellen Kommentaren (Tags), können im Quelltext angebracht und automatisch ausgewertet werden. So kann z.B. bestimmt werden, ob der Inhalt eines Objekts visualisiert werden soll.

2.4 Diagramme

Für den Entwurf eines Diagramms, das für ein beliebiges Szenario einen guten subjektiven Eindruck des Szenarios ermöglicht, spielt insbesondere die Menge und die Qualität der darzustellenden Informationen eine Rolle.

Der Umfang des Szenarios ist im Allgemeinen so groß, dass nicht alles in einem Diagramm dargestellt werden kann, ohne den Überblick zu verlieren. Außerdem sind zu einem bestimmten Thema nicht alle Details notwendig, da sie das Verständnis in einem bestimmten Zusammenhang nicht erhöhen.

Es soll also nur ein Teil des gesamten Szenarios in einem Diagramm bzw. Bild dargestellt werden. Daher muss es möglich sein, auf Wunsch den darzustellenden Teil bzw. die Details zu bestimmen, bzw. zu vergrößern (*browsing*). Das ist auf zwei Arten möglich:

- Zum einen kann das dargestellte Diagramm größer als das Fenster bzw. der Bildschirm sein. In diesem Falle muss das Fenster zu rollen sein.
- Zum anderen können Details der Darstellung vergrößert bzw. expandiert oder weggelassen werden. In diesem Falle muss das Diagramm umorganisiert, oder ein neues Diagramm geöffnet werden.

Die Steuerung der Ansicht kann entweder durch den Benutzer (per Maus) oder automatisch geschehen.

2.4.1 Diagrammtypen

Es kommen verschiedene Diagrammtypen in Frage. Diese orientieren sich zum Teil an der *Unified Modeling Language* (*UML*), einer graphischen Sprache für Visualisierung, Spezifikation, Konstruktion und Dokumentation komplexer Softwaresysteme. [BOOCH] [OESTEREICH]

2.4.1.1 Ablaufdiagramm

Ein Ablaufdiagramm, auch Interaktions- oder Sequenzdiagramm genannt, besteht aus horizontal angeordneten Rechtecken für die Objekte und nach unten gerichteten Zeitbalken an jedem Objekt. Methodenaufrufe werden durch waagerechte Verbindungen dargestellt. Es wäre auch eine dreidimensionale Darstellung für z.B. sternförmige Abhängigkeiten denkbar. Jedoch ist dann eventuell die Zeit in der zweidimensionalen Darstellung am Bildschirm nicht mehr ausreichend erkennbar (Aufrufe sind nicht mehr horizontal dargestellt). Das Diagramm "wächst" während des Programmlaufs nach unten. Im Gegensatz zum Kollaborationsdiagramm steht der zeitliche Verlauf der Interaktionen im Vordergrund.

2.4.1.2 Kollaborationsdiagramm

Das Kollaborationsdiagramm ist ein Interaktionsdiagramm und zeigt die gleichen Sachverhalte wie das Ablaufdiagramm, jedoch stehen hier die Objekte und ihre Zusammenarbeit im Vordergrund. Die Objekte sind beliebig im Diagramm verteilt. Interaktionen zwischen den Objekten, d.h. Methodenaufrufe werden durch Verbindungen (Pfeile) eingezeichnet. Die Reihenfolge der Aufrufe kann durch eine Nummerierung dargestellt werden.

Die Objekte werden durch Rechtecke dargestellt. Im Gegensatz zum Klassendiagramm werden konkrete Objekte von Klassen und konkrete Referenzen angezeigt.

2.4.1.3 Objektdiagramm

Um eine Methode in einem Objekt aufzurufen, muss der Aufrufer eine Referenz auf dieses Objekt haben. Diese Referenz kann der Aufrufer über ein lokales Attribut, über ein globales (statisches öffentliches) Attribut einer beliebigen Klasse oder über einen Parameter einer Methode beziehen, oder es liegt eine Selbstreferenz vor. Das Objektdiagramm zeigt beliebige Referenzen zwischen Objekten und ist als ein Exemplar eines Klassendiagramms oder als Momentaufnahme eines Kollaborationsdiagramms anzusehen.

Ein Spezialfall einer Referenz ist die Aggregation. Sie beschreibt die Zusammensetzung eines Objekts aus anderen Objekten, den Komponenten, und ist eine „Teil-von“ Beziehung. Aggregationen werden durch Attribute vom Typ der Teilobjekte implementiert. (Umgekehrt müssen Attribute nicht auf eine Aggregation hinweisen!) Wenn ein Attribut eine Referenz auf ein Teilobjekt enthält, kann das Teilobjekt prinzipiell auch für sich existieren. Es gibt Programmiersprachen (z.B. Eiffel), in denen auch Wertattribute möglich sind. Mit diesen kann ein physisches Enthaltensein, d.h. eine existenzabhängige Aggregationsbeziehung implementiert werden. Das Teilobjekt kann dann nicht alleine existieren. In Java sind die semantischen Eigenschaften der Aggregation und der Existenzabhängigkeit nicht darstellbar (sie können höchstens mit entsprechendem Aufwand imitiert werden). Genauso wenig kann alleine aus der Existenz eines Attributs auf eine Aggregation geschlossen werden.

Es soll zum Zwecke einer intuitiven Darstellung möglich sein, eine Aggregation semantisch im Programm auszudrücken. Zur Darstellung einer Aggregation in einem Objektdiagramm müssen die Felder mit einer bestimmten Annotation (`/**@component*/`) im Quelltext gekennzeichnet sein. Für Felder, die in dieser Art gekennzeichnet sind, wird eine spezielle Darstellung für die Komponenten, die sie bezeichnen, gewählt. Eine Existenzabhängigkeit der Beziehung wird nicht berücksichtigt. Es ist wahlweise auch eine automatische Erkennung von Komponenten vorgesehen, z.B. für Felder, die `final` oder `private` deklariert sind.

In UML besteht ein Objektdiagramm aus Rechtecken für die Objekte der Klassen. Teilobjekte sind durch Linien (mit einer Raute beim Hauptobjekt) zwischen den Rechtecken dargestellt. Davon abweichend wird hier eine Darstellung gewählt wo Teilobjekte und Attribute primitiver Datentypen durch Kästchen innerhalb des Objekts dargestellt werden. Diese Kästchen können zur detaillierteren Darstellung expandiert werden. Die Darstellung durch verschachtelte Kästchen entspricht dem intuitiven Verständnis des Komponenten Begriffs.

Da eine Aggregation immer eine baumförmige, zyklenfreie Struktur hat, bietet sich für die Aggregationen alternativ auch eine Baum Darstellung an (die in Swing implementiert werden kann).

2.4.2 Resümee

Objektdiagramme und Interaktionsdiagramme können nur in einem begrenzten Umfang statisch initialisiert werden. Erst während des Programmlaufes ist bekannt, wie Variablen belegt werden, wie viele Objekte ein Listenattribut beinhaltet, wer wen aufruft, etc. Die Diagramme müssen daher während des Programmlaufes dynamisch auf- bzw. abgebaut werden.

Um die Übersichtlichkeit zu bewahren muss die Fülle der Verweise und Objekte reduzierbar sein. Objekte müssen daher versteckt werden können.

Listen mit Objekten, Methodenaufrufstacks und Quelltext runden die Darstellung ab. Desweiteren kann der Inhalt von Zahlen Attributen (z.B. vom Typ `int` oder `float`) durch einen Balken dargestellt werden, wenn der Definitionsbereich bekannt ist.

Objektdiagramme zeigen im Wesentlichen die Objektstruktur und dynamisch entstandene Referenzen. Interaktionsdiagramme zeigen die Historie der Aufrufe bzw. Interaktionen.

2.5 Beziehungen zwischen Objekten

Falls ein Unterobjekt eines Objekts nicht durch eine entsprechende Annotation als Komponente gekennzeichnet ist, wäre das automatische Zeichnen eines Objektbaums möglich, indem ein aufspannender Baum im Objektgraph konstruiert wird. In einem aufspannenden Baum kommen alle Knoten nur einmal vor. Es resultiert zwar ein Objektbaum, die gewünschte „Teil-von“ Semantik wäre damit aber noch nicht automatisch erkannt. Dazu müsste das System Komponenten und

andere Referenzen selbständig unterscheiden können. Es ist wahlweise zwar eine automatische Erkennung von Komponenten vorgesehen, z.B. für Felder, die `final` oder `private` deklariert sind. Dies ist aber nur eine einfache Heuristik, die einen bestimmten Programmierstil voraussetzt.

An dieser Stelle wären Werkzeuge interessant, die den Quelltext auf die Eigenschaften hin analysieren, ob Aggregationen enthalten sind. Clarke, Noble und Potter beschäftigen sich in [CLARKE] in [ECOOP] mit sogenannten *Ownership Types*. Diese beschreiben gewisse *Enthaltensein Beziehungen* zwischen Objekten. Objektorientierte Programme besitzen in ihrem Konzept bereits *Enthaltensein Beziehungen*, denn Objekte können auf andere Objekte verweisen, oder, anders interpretiert, andere Objekte enthalten. Die *Ownership Types* erweitern dieses Konzept auf unterschiedliche *Enthaltensein Beziehungen*, die selbst definiert werden können und sich zur Laufzeit eines Programmes ausprägen. Die Eigenschaften dieser Beziehungen werden in abstrakter Form beschrieben, was eine Voraussetzung für eine automatische Erkennung gewisser Beziehungen ist.

Ohne Einschränkungen in der Sichtbarkeit ist ein Unterobjekt auch für ganz andere Objekte als die umgebenden Oberobjekte sichtbar. Ein Unterobjekt ist eines, welches in einem Oberobjekt enthalten ist, das Oberobjekt umgibt das Unterobjekt. Die Sichtbarkeit kann in Java mit den Eigenschaften `public`, `protected` und `private` für Unterobjekte definiert werden. Zugriffsmethoden können wiederum geschützte Unterobjekte, oder nur gewisse Aspekte dieser Objekte für fremde Objekte sichtbar machen. Wird ein Unterobjekt sichtbar gemacht, kann in Java ein anderes Objekt eine Referenz darauf erhalten. Es kann jedoch nur eingeschränkt auf die Paketstruktur und die Vererbungsverhältnisse definiert werden, wer Zugriff erhält. Wird ein Objekt für einen lesenden Zugriff freigegeben, kann es prinzipiell auch von dem Empfänger verändert werden. Eine feinkörnigere Definition der Sichtbarkeit ist in Java nicht Bestandteil der Sprache und muss gegebenenfalls unter Verwendung geeigneter Muster im Code implementiert werden. Anpassbarer Zugriffschutz und Sichtbarkeit sind auch Thema bei Noble, Vitek und Potter in [NOBLE]. *Ownership Types* können die Sichtbarkeit bestimmen, denn ein Objekt, welches als Besitz eines Oberobjekts definiert ist, soll gewissen Zugriffschutz genießen. Damit beschäftigen sich auch Clarke, Potter und Noble in [POTTER] und [CLARKE2].

Wenn die im Programm definierten Zugriffeigenschaften analysiert werden, könnte darauf basierend eine Komponentenstruktur interpretiert werden. Die Zugriffeigenschaften werden nicht nur durch die Schlüsselworte `public`, `protected` und `private` bestimmt, sondern auch durch die Art der Zugriffsmethoden. Wenn z.B. eine öffentliche Methode existiert, die einen schreibenden Zugriff auf eine Variable hat, so ist es nicht unbedingt sinnvoll, das Unterobjekt in der Variable als Unterkomponente zu betrachten, da eine Unterkomponente nicht von fremden Objekten modifiziert werden soll. Es kann also von der Sichtbarkeit auf die Besitzverhältnisse im Sinne von Aggregationen bzw. Komponentenbeziehungen geschlossen werden. Um die Regeln dieser Abbildung zu definieren und ihre Widerspruchfreiheit zu beweisen, ist ein Formalismus notwendig, wie er in den vorgestellten Werken behandelt wird.

3 Beispiele

Das Visualisierungssystem enthält neben einfachen Beispielen zum Testen zwei komplexere Beispiele. Diese sind bereits lauffähig vorbereitet. Beide Beispiele zeigen Abläufe und Datenstrukturen. Das erste Beispiel ist besonders im Hinblick auf Abläufe in einem nebenläufigen System, das zweite in Bezug auf seine Datenstruktur interessant. Die Beispiele werden an verschiedenen Stellen in diesem Dokument zur Erläuterung verwendet. Der Quelltext der Beispiele befindet sich im Anhang.

3.1 Ein Beispiel mit mehreren konkurrierenden Threads

Mehrere Arbeiter (`Worker extends Thread`) versuchen, sich gegenseitig Arbeitsaufwand zuzuschieben. Dazu können sie sich von einer zentrale Instanz (`Loadbalancing`) Arbeitszeit bei dem am wenigsten beschäftigten reservieren lassen. Die Arbeiter sind hier als Bestandteile (Komponenten) der Zentralinstanz eingezeichnet (**Abbildung 2**).

Ein Arbeiter ist ein Thread, der fortwährend einen Teil seines Arbeitsaufwandes an einen anderen abgibt (`delegate`), falls er zu viel zu tun hat (`tooMuchToDo`), und selbst arbeitet (`doWorkCycle`). Zum Delegieren fragt er zunächst bei der Zentralinstanz nach, wer am wenigsten zu tun hat (`freeWorker`), schiebt diesem Arbeiter dann Arbeit zu (`putWork`) und verringert schließlich seine Auslastung (`decreaseUtilization`). Auf die Frage nach einem freien Arbeiter (`freeWorker`) lässt sich die Zentralinstanz (Loadbalancing) zunächst von jedem Arbeiter seine Auslastung geben (`utilization`), ermittelt den mit der kleinsten Auslastung, reserviert bei diesem Arbeiter Arbeitszeit (`increaseUtilization`) und gibt eine Referenz auf ihn an den Aufrufer zurück.

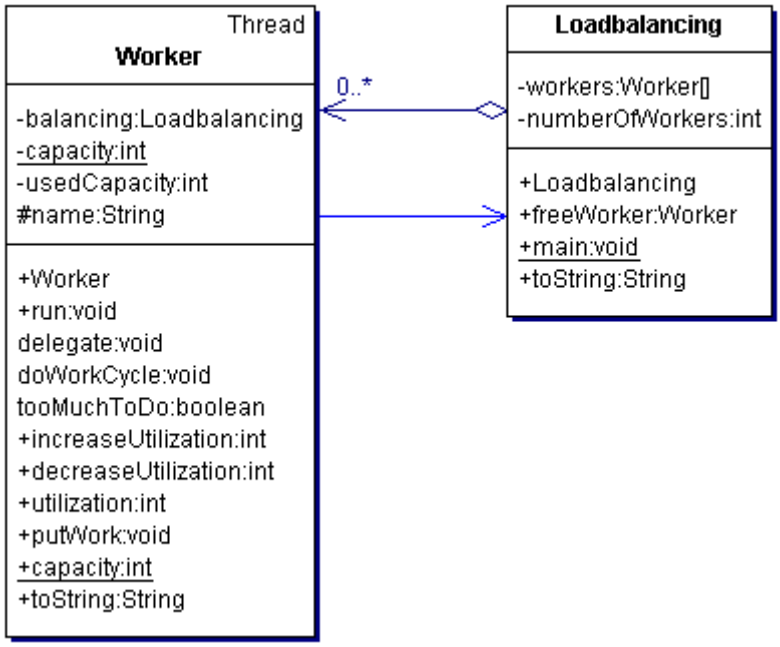


Abbildung 2 Entwurf eines Beispiels aus zwei Klassen.

Die Entwicklungsumgebung *Together* kann für dieses Beispiel statisch aus dem Quelltext ein Ablaufdiagramm erzeugen (**Abbildung 3**). Dieses hat folgende Nachteile:

- Es macht nicht deutlich, dass ein Arbeiter aus einer Menge ausgesucht wird, und zwar der, der über die Bedingung '`u<=minWorkLoad`' ermittelt wird. Der Aufruf von `utilization` wird nicht nur auf dem Objekt `worker` ausgeführt, sondern auch auf `freeworker`.
- Der ausgewählte Arbeiter (`freeworker`) und der belastete (`free`) werden als zwei Objekte eingezeichnet, obwohl es sich um dasselbe Objekt handelt.

- Die Nebenläufigkeit der Threads kommt nicht zur Geltung. In Wirklichkeit laufen die Zyklen aller Arbeiter je nach Synchronisation verzahnt ab.

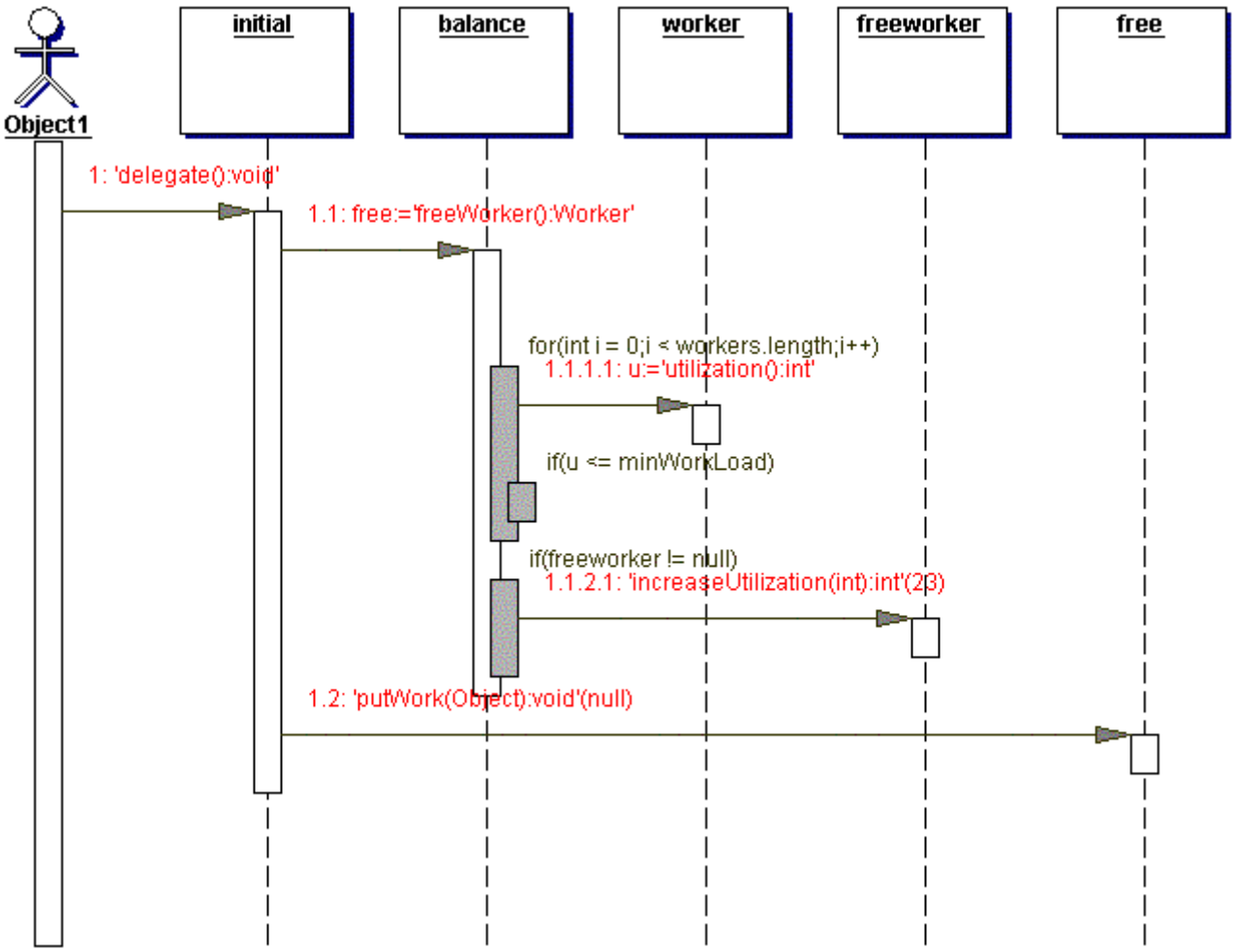


Abbildung 3 Ein Ablaufdiagramm, welches von der Entwicklungsumgebung *Together* generiert wurde.

Ein statische erzeugtes „Ablaufdiagramm“ kann nicht den wirklichen Ablauf zeigen, wenn es, so wie in diesem Beispiel, nicht deterministische Einflüssen gibt. Aber auch bei einem deterministisch ablaufenden Programm ist es z.B. schwierig, die Folge von Entscheidungen festzustellen. Dazu wäre eine inhaltliche Analyse notwendig.

Das vorliegende Visualisierungssystem zeigt einen realen Ablauf, siehe dazu auch die **Abbildung 21** im Abschnitt 4.6.3.5 *Ablaufdiagramm* auf Seite 28.

3.2 Ein Beispiel mit Tabellen und Mengen (Bibliothek)

Das Beispiel besteht aus den selbst erklärenden Klassen `Bibliothek`, `Benutzerverzeichnis`, `Benutzer`, `Katalog` und `Buch` (**Abbildung 4**). Die Klasse `StartBibliothek` (nicht abgebildet) enthält einige Aufrufe, um einen Datenbestand über die Methoden der Klassen aufzubauen bzw. einen Ablauf zu simulieren.

Die Klasse `Bibliothek` ist die Anlaufstelle. Über die Methode `ausleihen` kann ein Buch ausgeliehen werden. Die `Bibliothek` besitzt ein `Benutzerverzeichnis` und einen `Katalog`. Beides sind Verzeichnisse, die Schlüssel auf Werte abbilden. Im Benutzerverzeichnis kann mit der Methode `suchen` ein `Benutzer` Objekt über seine Nummer, im Katalog ein `Buch` über die Signatur identifiziert werden. Das Benutzerverzeichnis enthält also beliebig viele (`0..*`) Benutzer, der Katalog beliebig viele Bücher. Wenn in der Bibliothek die Methode `ausleihen` aufgerufen wird,

werden in den Verzeichnissen über die Methode `suchen` das `Benutzer` und das `Buch` Objekt gesucht. Dann erfolgt über die Methode `ausleihen` des Benutzers und des Buches die Erstellung der Referenzen zwischen Benutzer und Buch. Beim Buch wird das Attribut `leser` eingetragen, beim Benutzer wird das Buch in die Menge der ausgeliehenen Bücher (`ausgeliehen`) eingefügt.

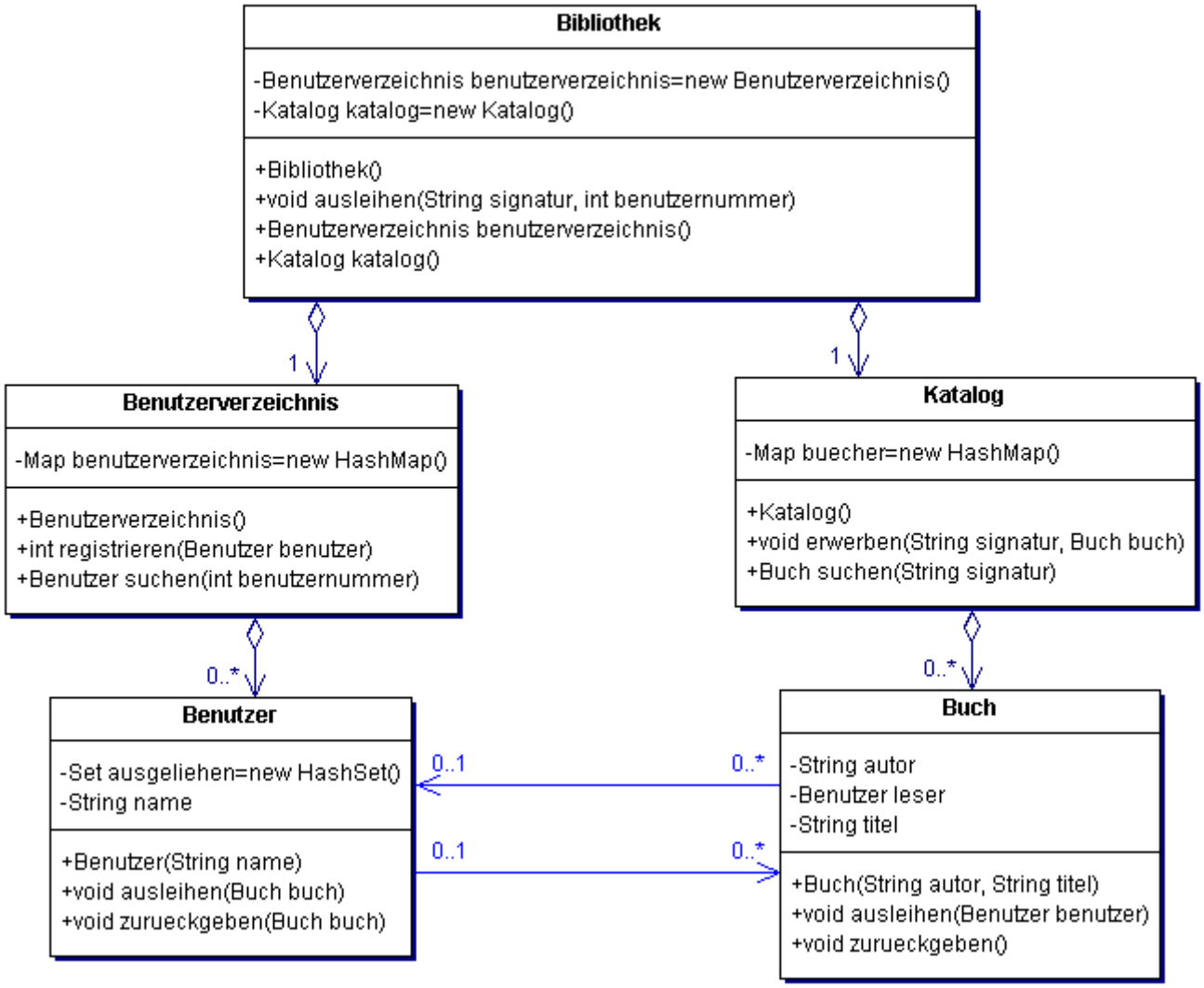


Abbildung 4 Das Beispiel *Bibliothek*.

Das vorliegende Visualisierungssystem zeigt ein Objektdiagramm einer Ausprägung des Beispiels, siehe dazu auch die **Abbildung 20** im Abschnitt 4.6.3.5 *Ablaufdiagramm* auf Seite 27.

4 Funktionsumfang und Bedienung

In diesem Kapitel wird die Installation und Bedienung des Visualisierungssystems beschrieben. Dabei werden auch die sichtbaren Elemente der graphischen Oberfläche und der Funktionsumfang erläutert. Die Verwendung der Annotationen im inspizierten Programm wird in Kapitel *5 Tags: Annotationen im Quelltext* beschrieben.

4.1 Installation des Visualisierungssystems

Die zip-Datei muss in einem beliebigen Verzeichnis Ihres Dateisystems ausgepackt werden. Es entsteht ein Verzeichnis mit dem Namen 'JAN'.

Hinweis für Windows: Es dürfen **keine Leerzeichen** oder Sonderzeichen im Dateipfad verwendet werden!

Das System benötigt eine Java Laufzeitumgebung [JAVA] der Version 1.4 von SUN. Diese muss von java.sun.com runtergeladen und installiert werden, falls nicht schon vorhanden. Die Java VM kann in das Verzeichnis 'JAN/Java' installiert werden. Wenn ein anderes Verzeichnis gewünscht ist, muss dem System aber der Ort der Java Laufzeitumgebung mitgeteilt werden. Dazu müssen an zwei Orten Pfade gesetzt werden: Im Verzeichnis 'JAN/Properties' gibt es die Datei 'path.properties', in der der Java Pfad entsprechend angepasst werden muss. Die gleiche Änderung muss im Verzeichnis 'JAN' in der Skriptdatei 'RUN.bat' für Windows vorgenommen werden, falls das Programm unter Windows mit diesem Skript gestartet werden soll.

4.2 Schnellstart unter Windows

Es können sofort die mitgelieferten Beispielprogramme visualisiert werden. Dazu muss nichts weiteres installiert werden. Es ist folgendermaßen vorzugehen:

1. Skript '*RUN.bat*' im Ordner '*JAN*' ausführen. Es dauert dann etwas, bis sich die graphische Oberfläche öffnet.[2]

2. Zum Ausführen des voreingestellten Beispiels auf die Karte '*execution view*' wechseln und wiederholt den Button '*step*' drücken. In der Liste links unten auf '*Loadbalancing*' klicken. Dann auf dem Baum rechts daneben '*Loadbalancing*' doppelt klicken und rechts unten auf die Karte '*object diagram*' wechseln. Im Objektdiagramm mit der rechten Maustaste auf Objekte klicken und das Menü testen. Nach einigen Schritten kann auch die Karte '*sequence diagram*' betrachtet werden. Auf beiden Karten können Fenster mit der Maus verschoben werden. Auf der Karte '*object diagram*' können über den dargestellten Objekten mit der rechten Maustaste ein Menü geöffnet und die Menüpunkte ausgeführt werden.

4.3 Einbringen der zu inspizierenden Programme

Das Visualisierungssystem kann nur betrieben werden, wenn ein zu inspizierendes Java Programm in Form seines Quelltextes vorhanden ist. Der Quelltext muss in unverpackter Form in seiner kompletten Package Struktur in den Ordner '*JAN/Sandbox/Source*' gestellt werden. Einige Beispielprogramme sind zu Testzwecken bereits vorhanden.

Hinweis: Der Ordner '*JAN/Sandbox*' und seine Unterordner müssen schreibbar sein. Ansonsten können keine eigenen Programme visualisiert werden!

4.4 Starten des Visualisierungssystems

Auf Windows kann die Skriptdatei '*RUN.bat*' ausgeführt werden. Nach kurzer Zeit öffnet sich die graphische Oberfläche des Systems. Für andere Betriebssysteme oder bei Änderungen an der Pfadstruktur kann ein analoges Skript selbst geschrieben werden. Alternativ können die Befehle des Skripts natürlich auch direkt in einer Konsole eingegeben werden.

---

2 Je nach Konfiguration des Windows Systems kann es vorkommen, dass die Meldung '*Kein Speicherplatz im Umgebungsbereich*' ausgegeben wird. In diesem Falle erhöhen Sie den Speicherplatz über das Menü der Windows Eingabeaufforderung.

4.5 Beenden des Visualisierungssystems

Das Programm ist über das Symbol '*Schließen*' am Fensterrand zu beendet. Bei Beendigung durch '*Steuerung-C*' oder ähnlichem kann es vorkommen, dass nicht alle Prozesse beendet werden, die das Programm gestartet hat. In diesem Falle müssen evtl. zurückgebliebene Java Prozesse im Windows Taskmanager (aufrufbar mit der Tastenkombination 'Strg-Alt-Entf') beendet werden, bevor JAN erneut gestartet wird. (Ein Neustart des Computers bewirkt dasselbe.)

4.6 Benutzung des Visualisierungssystems und Beschreibung der Oberfläche

Die sichtbare Oberfläche des Visualisierungssystems besteht aus graphischen Elementen, wie sie üblicher Weise für Anwenderoberflächen Verwendung finden.[3] Auf die einzelnen Elemente wird daher nicht weiter eingegangen. Die Erscheinungsform ist vom Betriebssystem abhängig, besitzt aber immer die gleiche Funktionalität.

Die graphische Oberfläche des Visualisierungssystems enthält zwei Karten: '*development view*' und '*execution view*'.

4.6.1 Entwickler-Sicht

Auf der Karte '*development view*' (**Abbildung 5**) wird ein zu inspizierendes Programm für die Visualisierung vorbereitet. Die Karte besitzt oben eine Leiste mit Buttons und darunter eine Leiste mit Textmustern. Auf der linken Seite befindet sich die Liste '*list of classes*' mit geöffneten Quelldateien, auf der rechten können Fenster mit Quelltext geöffnet werden. Unten gibt es einen Bereich für Meldungen.

Mit dem Button '*open*' wird ein Dateidialog Fenster geöffnet, in dem Javaquelldateien ausgewählt werden können. Wenn der Dateidialog das erste Mal geöffnet wird, ist der Inhalt des Ordners '*JAN/Sandbox/Source*' zu sehen. Es kann durch alle Unterordner dieses Ordners (und durch die Ordner in Richtung Laufwerksangabe) gewandert werden, andere Ordner außerhalb dieser Pfade sind über diesen Dialog aus Sicherheitsgründen nicht zu erreichen. Es lässt sich jeweils eine Java Datei auswählen und in eine Liste ('*list of classes*' auf der linken Seite der Karte) übernehmen.

Durch Doppelklick auf die Liste '*list of classes*' öffnet sich der Quelltext im Quelltexteditor rechts daneben. Der Quelltext kann editiert und mit 'save' gespeichert werden (**Abbildung 6**). Sinn des einfachen Editors ist es, die für die Visualisierung benötigten Annotationen einzufügen (siehe dazu das Kapitel *5 Tags: Annotationen im Quelltext*). Als Hilfe können die Tags aus den oben vorhandenen Textfeldern kopiert werden. Der Editor kann grundsätzlich auch zum Editieren des Javacodes verwendet werden. Nach dem Editieren können die Dateien der Liste einzeln über den Button 'save' gespeichert werden.

Hinweis: Die Java Dateien können auch mit einem externen Editor eigener Wahl (außerhalb des GUI Fensters) bearbeitet werden.

3 Es werden hier die gebräuchlichen englischen Begriffe verwendet wie z.B. 'Button' statt 'Knopf'.

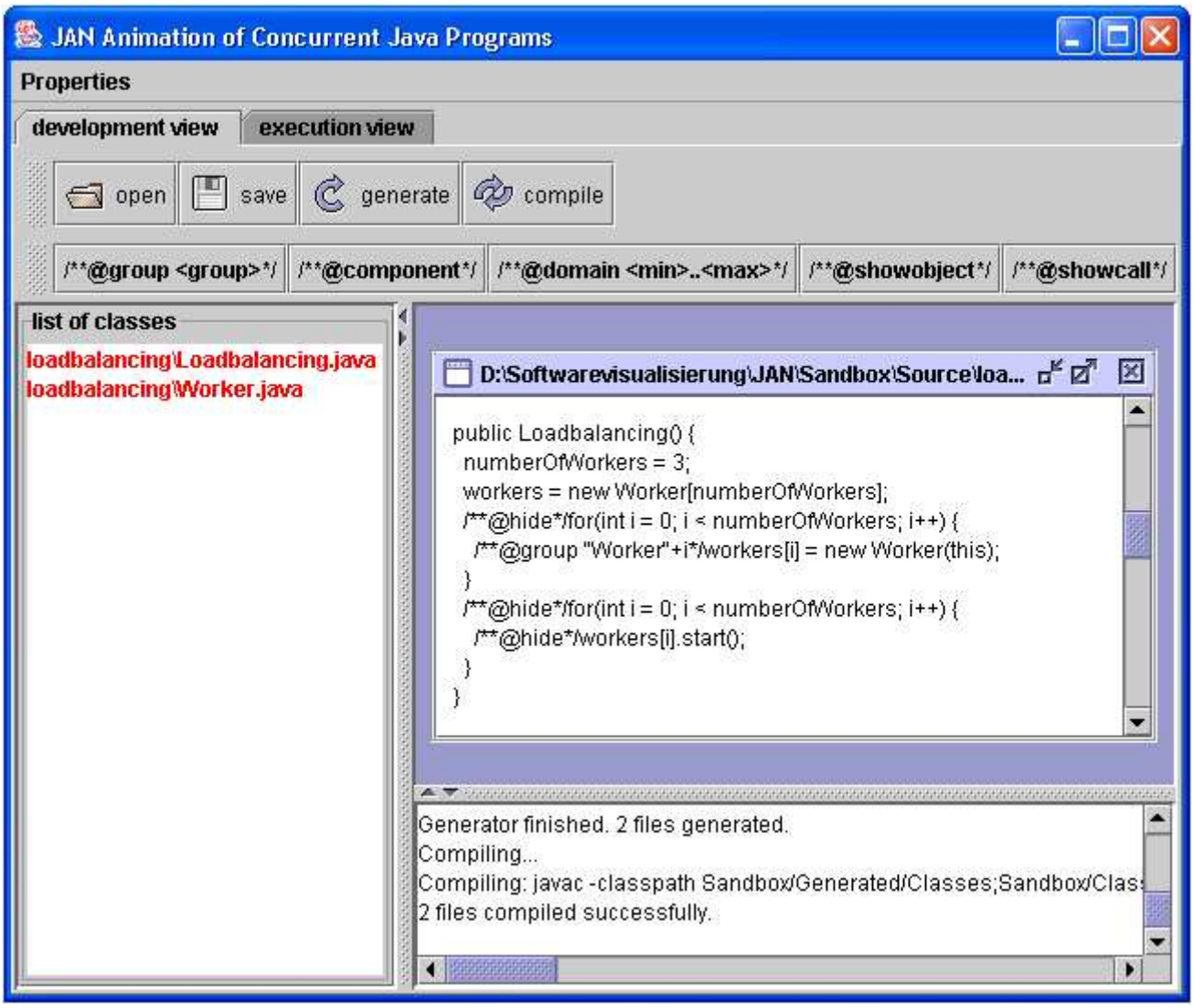


Abbildung 5 Das Visualisierungstool in der Entwickler-Sicht.

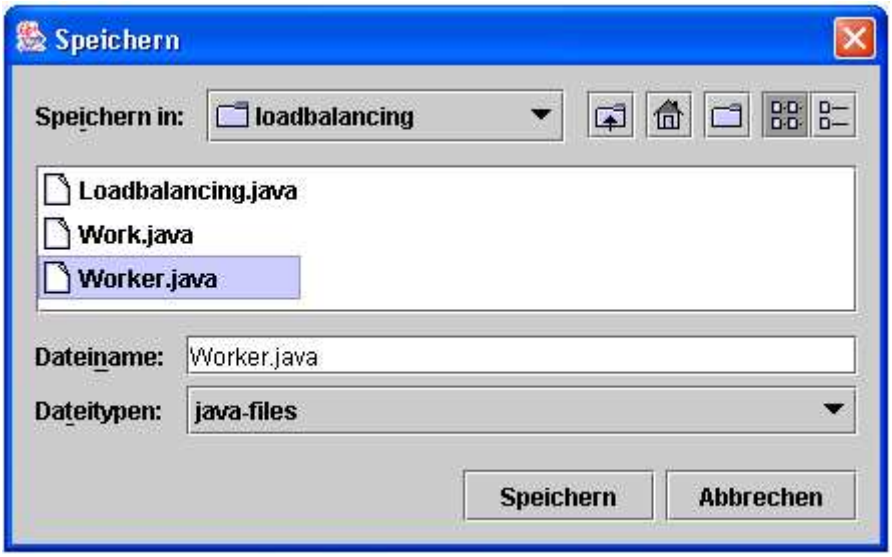


Abbildung 6 Der Dialog zum Speichern einer Java Datei.

Mit dem Button '*generate*' werden für alle in der Liste markierten Quelldateien modifizierte Dateien generiert. Diese werden vom Visualisierungssystem benötigt[4]. Nach dem Generieren müssen die modifizierten Quelldateien durch Betätigung des Buttons '*compile*' übersetzt werden. Für das Generieren und Übersetzen können mit Hilfe der Steuerungs- und Umschalttaste auf der Tastatur auch mehrere Einträge gleichzeitig markiert werden. Über den Erfolg oder Misserfolg beim Generieren und Übersetzen wird in einem Meldungsbereich rechts unten informiert. Z.B. werden dort die Übersetzungsfehler in Textform dargestellt.

Hinweis: Veränderte Dateien müssen immer erst gespeichert, dann generiert und schließlich übersetzt werden! Wird z.B. das Generieren vergessen, bleibt das Übersetzen wirkungslos. Es werden dann gegebenenfalls alte Versionen für den Programmlauf verwendet. Wenn einzelne Dateien nicht im übersetzten Zustand vorliegen, wird auf die Klassen im Ordner '*JAN/Sandbox/Classes*' zurückgegriffen.

4.6.2 Organisation des Dateisystems

Die Installation des Visualisierungssystems hat folgende Ordnerstruktur:

- *JAN*
    - *Program*
    - *Extern*
        - *Barat*
        - *graphics*
    - *Properties*
    - *Java*
    - *Sandbox*
        - *Source*
        - *Classes*
        - *Generated*
            - *Source*
            - *Classes*

Die Ordner '*Program*' und '*Java*' enthalten das Visualisierungssystem und die Java Laufzeitumgebung und müssen nicht beachtet werden. Im Ordner '*graphics*' in *Extern* können Bilder abgelegt werden, die für die Darstellung verwendet werden sollen. Der Ordner '*Properties*' enthält Dateien mit Grundeinstellungen.

Der Ordner '*Sandbox*' enthält die zu visualisierenden Programme. In den direkten Unterordnern '*Source*' und *'Classes*' werden die Quelltexte und gegebenenfalls Klassen abgelegt. Auf die Klassen kann verzichtet werden, wenn alle Quelldateien generiert und übersetzt werden sollen.

Die vom Visualisierungssystem generierten Quelldateien speichert das System standardmäßig im Ordner '*Source*' in '*Generated*'. Die Klassen der darin enthaltenen und vom System übersetzten Quellen werden im Ordner '*Classes*' in '*Generated*' abgelegt.

Quellen, die nicht visualisiert werden sollen, müssen nicht generiert und übersetzt werden. Das System sucht zum Ausführen eines Programmes immer erst im Ordner '*Sandbox/Generated/Classes*' nach Klassen und dann erst im Ordner '*Sandbox/Classes*'.

4 Die generierten Java Dateien stehen im Ordner '*JAN/Sandbox/Generated/Source*'. Für die Verwendung des Visualisierungssystems ist es nicht erforderlich, diese einzusehen. Sie können daher über den Button '*open*' nicht geöffnet werden. Neugierigen bleibt es vorbehalten, sich diese in einem externen Editor anzusehen.

4.6.3 Ausführung-Sicht

In der Ausführung-Sicht (**Abbildung 7**) werden der Ablauf und die Datenstruktur eines laufenden Programmes dargestellt. Im linken Bereich werden die aus dem Programmlauf bekannten Strukturen aufgelistet, der rechte Bereich ist für Quelltext und Diagramme vorgesehen. Oben gibt es eine Leiste mit Buttons, direkt darunter einen Balken mit Angaben zum Status. Im mittleren Bereich gibt es Listen für beobachtete Threads und ihre Historien und Stacks, sowie für Objekte und ihre Darstellung als Baum. Rechts oben ist Quelltext mit aktuellen Anweisungen zu sehen, rechts unten werden ein Ablauf- oder Objektdiagramm dargestellt. Ganz unten gibt es einen Bereich für Meldungen.

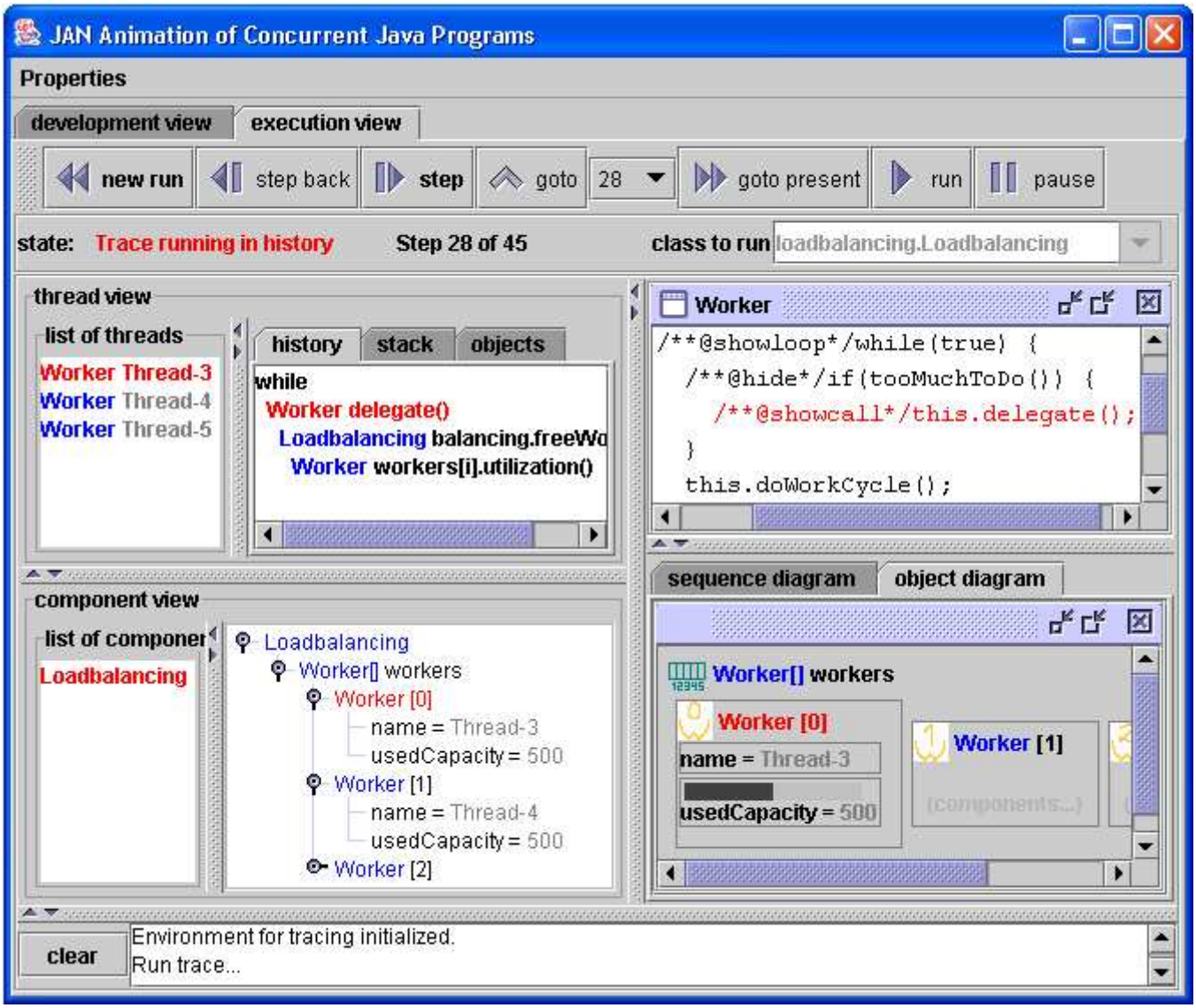


Abbildung 7 Das Visualisierungstool in der Ausführung-Sicht.

4.6.3.1 Ausführen eines Programmes

Im oberen Bereich der dynamischen Ansicht befinden sich Steuerungselemente, mit denen ein zu inspizierendes Programm ausgesucht und ausgeführt werden kann. Die Buttons dafür haben folgende Funktionen:

Button	**Funktionalität**
new run	Initialisierung eines neuen Ablaufs.
step back	Einen Schritt im Ablauf zurückgehen.
step	Einen Schritt im Ablauf weitergehen.
goto + Zahlauswahl	Zu einem beliebigen Schritt aus der Vergangenheit springen.
goto present	Aus der Vergangenheit zurück in die Gegenwart springen.
run	Ablauf kontinuierlich ausführen.
pause	Den kontinuierlichen Ablauf anhalten.

Tabelle 2 Die Buttons zum Steuern des Programmablaufs.

Mit den Buttons '*run*' und '*stop*' kann das inspizierte Programm fortlaufend ausgeführt und angehalten werden. Dafür wird ein Zeitgeber verwendet, der den Button '*step*' automatisch betätigt. Die Dauer einer Periode kann über das Menü eingestellt werden.

Der Button '*step*' ist der wichtigste. Bei jeder Betätigung wird jeweils ein Schritt in der Ablaufverfolgung ausgeführt[5].

Hinweis: Ein einzelner Schritt im Programm wird nicht unmittelbar dargestellt, wenn das betroffene Objekt gerade nicht eingeblendet ist. Das Ereignis im Programm wird aber gespeichert und bei entsprechender Auswahl der dargestellten Elemente mit angezeigt.

Mit '*new run*' kann ein neuer Lauf des zu inspizierenden Programmes initialisiert werden. Bevor daraufhin das erste Mal '*step*' oder '*run*' gedrückt wird, kann die auszuführende Klasse in der Auswahl '*class to run*' bestimmt werden[6]. Für den ersten Lauf muss '*new run*' nicht betätigt werden.

Die Liste enthält nur Klassen, die auf der Karte '*development view*' geöffnet wurden und eine `main` Methode in der üblichen Signatur '`public static void main(String[] ...`' enthalten.[7] Es kann aber auch eine Klasse von Hand eingetragen werden.[8]

Außer der auszuführenden Klasse, können im Textfeld '*method to run*' die auszuführende Methode und im Textfeld '*arguments*' die String Parameter der Methode eingetragen werden. (Falls keine Parameter vorhanden sind, bleibt das Feld leer.) Im Allgemeinen heißt die Methode '*main*' und die Klasse ist diejenige, die das auszuführende '*main*' enthält. Es kann aber auch eine andere statische Methode ausgeführt werden, sofern sie als Parameter ein '*String[]*' erwartet.

Die eben beschriebenen Funktionen beziehen sich auf eine Verfolgung aktueller Abläufe in einem Programm. Es kann aber auch in vergangene Zustände gesprungen werden. Mit '*step back*' wird jeweils der vorhergehende Zustand des Programmes visualisiert. Dieses Vorgehen vermittelt die Illusion einer Rückwärtsausführung des inspizierten Programmes. Mit '*goto*' kann ein beliebiger Schritt aus der Vergangenheit angesprungen werden. Vor der Betätigung des Buttons kann dazu im Rollfeld neben dem Button eine Nummer ausgewählt werden. Mit '*goto present*' kann wieder zurück in die Gegenwart gesprungen werden.

Die Verwaltung der Ablaufverfolgung kennt im Wesentlichen zwei Zustände des Ablaufs: Das Ausführen neuer Schritte im inspizierten Programm (Gegenwart) und die Inspektion bereits vergangener Ereignisse (Vergangenheit). Der jeweilige Zustand wird oben links in Rot angezeigt. Der Button '*step*' besitzt seine Funktionalität sowohl in der Vergangenheit als auch in der Gegenwart. Die Angabe des Zustands ist insofern wichtig, als dass die Vergangenheit einer bestimmten Programmausführung unabänderlich ist, auch wenn ein nicht deterministisches Programm untersucht wird. Außer dem Zustand wird auch die Nummer des gerade dargestellten Schritts angezeigt.

5 Ein Schritt in der Ablaufverfolgung ist durch die Annotationen im Quelltext des zu inspizierenden Programmes bestimmt und kann viele Befehle des Programmes beinhalten.

6 Ohne eigene Auswahl ist ein Testprogramm vorgegeben.

7 Die Schlüsselwörter müssen durch jeweils genau ein Leerzeichen getrennt sein, der *String[]* Parameter kann beliebig benannt sein.

8 Dies macht nur Sinn, wenn sichergestellt ist, dass die Klasse irgendwann einmal generiert und übersetzt worden ist.

Unten auf der Karte '*development view*' gibt es einen Bereich für Meldungen. Unter anderem werden dort die Exceptions und Ausgaben des inspizierten Programmes ausgegeben[9]. Da nicht festgestellt werden kann, ob das zu inspizierende Programm terminiert ist oder zur Zeit keine Ereignisse gesendet werden[10], erscheint nach einer gewissen Zeit die Meldung '*No event since last pushed step. Maybe the watched program is terminated*', wenn kein Ereignis empfangen wurde.

Es sei angemerkt, dass nur die Ereignisse im inspizierten Programm berücksichtigt und dargestellt werden, die gemäß der im Quelltext eingebauten Annotationen ausgewählt wurden und somit verarbeitet werden. Im inspizierten Programm kommt es also unter Umständen zu viel mehr Ereignissen wie z.B. Methodenaufrufen als das Visualisierungssystem beobachtet. Unter anderem kann es passieren, dass das inspizierte Programm noch läuft, aber keine Ereignisse mehr an das Visualisierungssystem sendet.

Hinweis: Es darf nicht vergessen werden, das Programm in der statischen Ansicht zu generieren und zu übersetzen, falls dies nicht schon in früheren Sitzungen geschehen war! (Die mitgelieferten Beispiele sind bereits generiert und übersetzt.) Ein nicht generiertes Programm meldet gar keine Ereignisse an das Visualisierungssystem.

Je nach Verwendung von Annotationen im Quelltext erscheinen oder verschwinden während der schrittweisen Ausführung in der Liste '*list of threads*' die beobachteten Threads und in der Liste '*list of component-roots*' die beobachteten Komponenten.

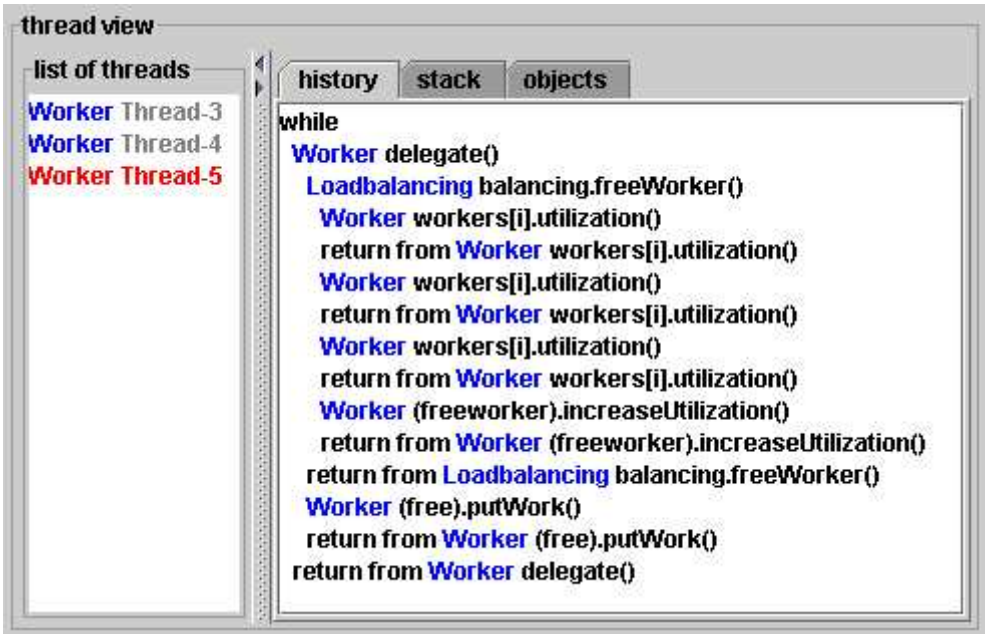


Abbildung 8 Die Sicht auf die Threads des inspizierten Programmes und einen ausgewählten Thread im Besonderen.

Durch Mausklick kann ein Thread ausgewählt oder die Historie und der Stack seiner Methoden- und Schleifenaufrufe auf den Karten '*history*' und '*stack*' dargestellt werden (**Abbildung 8**). Ferner werden Quelltextausschnitte mit der Umgebung der aktuellen Aufrufe des dargestellten Stacks angezeigt, wobei die aktuelle Zeile rot markiert ist (**Abbildung 9**). Quelltext Fenster können minimiert, maximiert und in der Größe mit der Maus verändert werden. Es werden Rollbalken angezeigt, falls ihr Inhalt größer als ihr Rahmen ist.

9 Die Ausgaben des inspizierten Programmes landen von sich aus auf der Standardausgabe der virtuellen Maschine, auf der das Programm läuft. Diese Standardausgabe ist in der Regel ein Konsolenfenster. Da eine Ausgabe auf diesem Fenster nicht schön wäre, wird die Ausgabe auf das Visualisierungssystem umgelenkt.

10 Es werden nur Ereignisse gemeldet, die aus Anweisungen im Quelltext hervorgehen, die speziell markiert sind bzw. vom System aufgrund gewisser Voreinstellungen aufbereitet wurden.

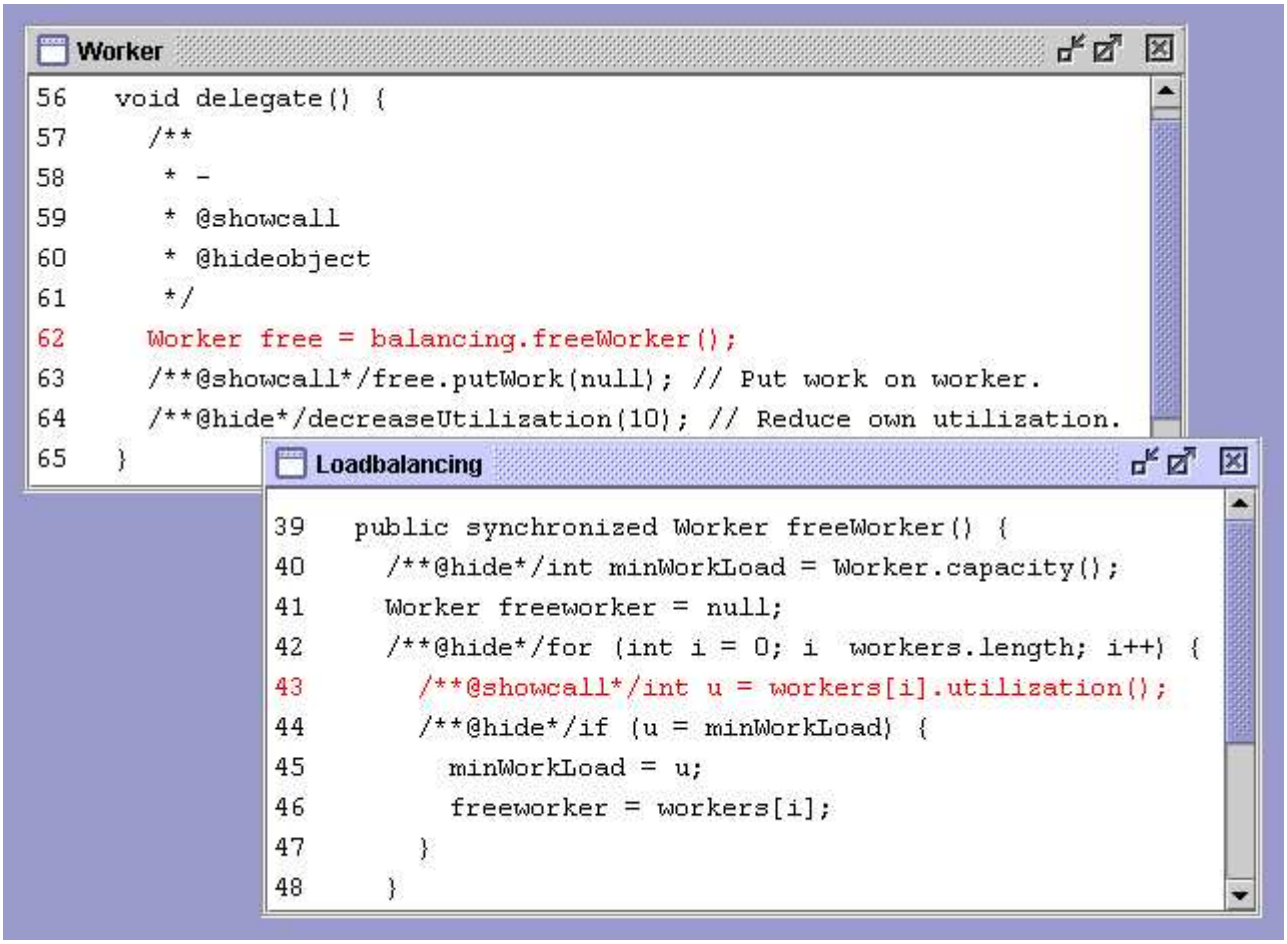


Abbildung 9 Quelltexte mit Rot markierten Anweisungen aus dem Stack eines Threads.

4.6.3.2 Threads, Historien und Stacks

In der Liste '*list of threads*' werden alle bekannten Threads aufgelistet. Es gibt für jeden Thread eine Historie und einen Stack. In der Historie sind Aufrufe und Schleifen, die in einer anderen enthalten sind, jeweils eingerückt dargestellt (**Abbildung 10**). Das Ende eines Aufrufs bzw. einer Schleife ist durch die zurückgenommene Einrückung und einem `return` bzw. `end` Eintrag zu erkennen. Der Stack beinhaltet die noch nicht abgearbeiteten Aufrufe und Schleifen der Historie (**Abbildung 11**). Wenn eine Schleife oder ein Methodenaufruf abgearbeitet ist, wird sein Eintrag aus dem Stack entfernt. Außerdem werden unter der Karte '*objects*' alle am Ablauf beteiligten Objekten und Klassen aufgelistet.

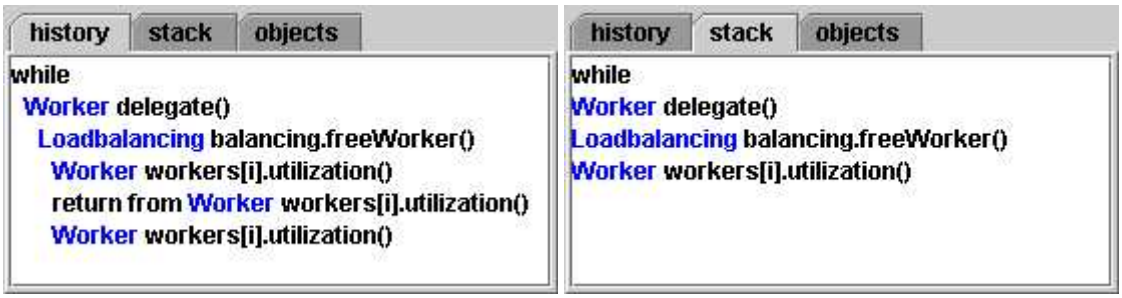


Abbildung 10 und **11** Historie und Stack eines Threads.

4.6.3.3 Objekte bzw. Komponenten

Im unteren Teil der Auflistungen gibt es in der Liste '*list of component-roots*' die Wurzeln aller bekannten Komponenten (**Abbildung 12**). Es werden dort *keine* Teilkomponenten aufgelistet. Wird z.B. eine aufgelistete Komponente als Teilkomponente an eine andere angehängt, so verschwindet die angehängte Teilkomponente aus der Liste.

Durch Mausklick kann eine Komponente aus der Liste ausgewählt und als Komponentenbaum dargestellt werden. (Hier erscheint die angehängte Teilkomponente wieder.)

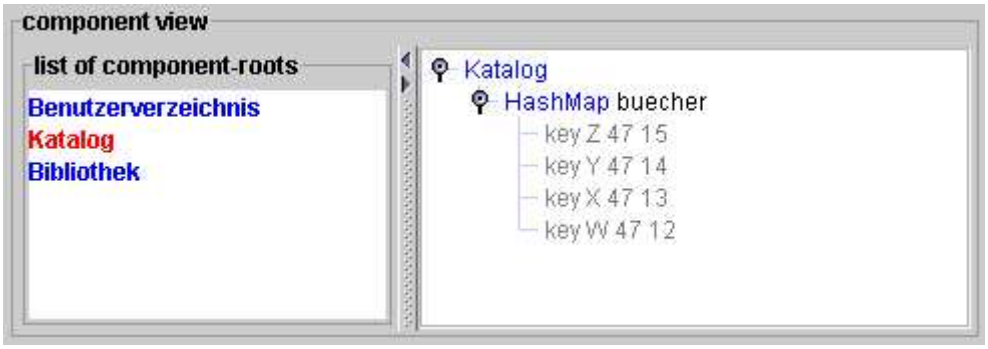


Abbildung 12 Die Sicht auf die Komponenten des inspizierten Programmes mit der Liste der Komponenten und einem ausgewählten Komponentenbaum.

Die Knoten des Baums sind Komponenten. Referenzen werden nicht im Baum dargestellt sondern nur im Objektdiagramm.[11] Eine Komponente kann primitiven oder zusammengesetzten Typs sein. Zu den primitiven Komponenten zählen hier die Datentypen `int`, `short`, `long`, `float`, `double`, `byte`, `char`, `boolean`, sowie `String`. Primitive Komponenten sind durch Blätter im Baum repräsentiert, zusammengesetzte Komponenten durch innere Knoten. Knoten, welche einen zusammengesetzten Typ darstellen, lassen sich durch Mausklick auf- bzw. zuklappen.

Eine Komponente komplexen Typs wird durch ihren Typ und einen Variablennamen bezeichnet, falls die Komponente in einer bekannten Variable enthalten ist. Für die Wurzeln und Komponenten von Collections[12] kann keine Variable angegeben werden. Bei Arrays wird der in eckigen Klammern eingeschlossene Index verwendet. Eine Besonderheit gibt es bei Behältern und primitiven Komponenten, die in lokalen Variablen deklariert sind. Sie gelten auch als Komponenten, ihr Variablenname wird zusätzlich in runden Klammern notiert.

Primitive Komponenten werden durch den Variablennamen, sofern er existiert (nicht bei Elementen einer Collection), und ihren Wert dargestellt. Der Typ kann aus dem Wert geschlossen bzw. aus dem Komponentendiagramm (**Abbildung 13**) entnommen werden.

Durch Doppelklick auf eine Komponente oder Unterkomponente in einer Baum Darstellung wird die angeklickte Komponente auf der Karte '*object diagram*' als Objektdiagramm geöffnet. Ein einfacher Klick in der Baum Darstellung markiert die angeklickte Komponente im Baum und im Objektdiagramm. Es können mit Hilfe der Steuerungstaste auf der Tastatur auch mehrere Einträge gleichzeitig markiert werden. Die Markierungen dienen der persönlichen Übersicht.

11 Eine Ausnahme bilden Behälter, die im Programm in lokalen Variablen definiert sind. Lokale Variablen können keine Komponenten beinhalten. Da Behälter aber nie als eigenständige Objekte, d.h. als Wurzel eines Komponentenbaums auftreten können, werden sie in jedem Falle als Unterkomponente in einem Komponentenbaum abgebildet. Bei lokalen Behältern wird die Beschriftung in runde Klammern eingeschlossen.

12 '*Collection*' ist ein Sammelbegriff für mengenwertige Komponenten und leitet sich von dem gleichnamigen Java Interface ab. Im Folgenden wird bevorzugt auch dieser Begriff verwendet, da die Bedeutung der behandelten Struktur eng an das Java Interface gekoppelt ist. Collections umfassen Mengen und Listen, welche sich in ihrem Modell der Datenhaltung unterscheiden.

4.6.3.4 Objektdiagramme

Die Karte '*object diagram*' ermöglicht die Darstellung von Objekten und ihren (Unter-) Komponenten, sowie von Referenzen zwischen den Komponenten.

Es können mehrere Fenster mit Komponenten im Diagramm geöffnet werden (**Abbildung 13**). Die Fenster können mit der Maus verschoben, minimiert, maximiert, in ihrer Größe verändert und geschlossen werden.

4.6.3.4.1 Komponenten

Die Komponentenstruktur von Komponenten wird durch geschachtelte Kästchen dargestellt. Kästchen, die *primitive Komponenten* darstellen, enthalten einen Schriftzug mit Variablennamen und Wert wie in der Baumdarstellung. Zahlenattribute, für die ein Definitionsbereich bekannt ist, werden außerdem mit einem Balken dargestellt, der entsprechend dem aktuellen Wert gefüllt ist.

Komplexe Komponenten, d.h. solche, die wiederum Komponenten enthalten können, werden durch Kästchen dargestellt, die einen Text mit Typ und Variablennamen wie in der Baumdarstellung, sowie gegebenenfalls weitere Kästchen enthalten. Enthaltene Kästchen können ausgeblendet sein, dann ist in Hellgrau der Schriftzug '*components...*' eingebettet. Außerdem wird für komplexe Komponenten ein kleines Bild (Icon) gezeigt, welches objekt- oder typabhängig sein kann. Icons zeigen eine Gruppenzugehörigkeit von Komponenten. Arrays, Listen und Mengen haben z.B. ihre eigenen Bilder. Wie Klassen und Objekten eine Gruppe zugeordnet werden kann, ist in Abschnitt 5.2.4 *Gruppenzugehörigkeit* beschrieben.

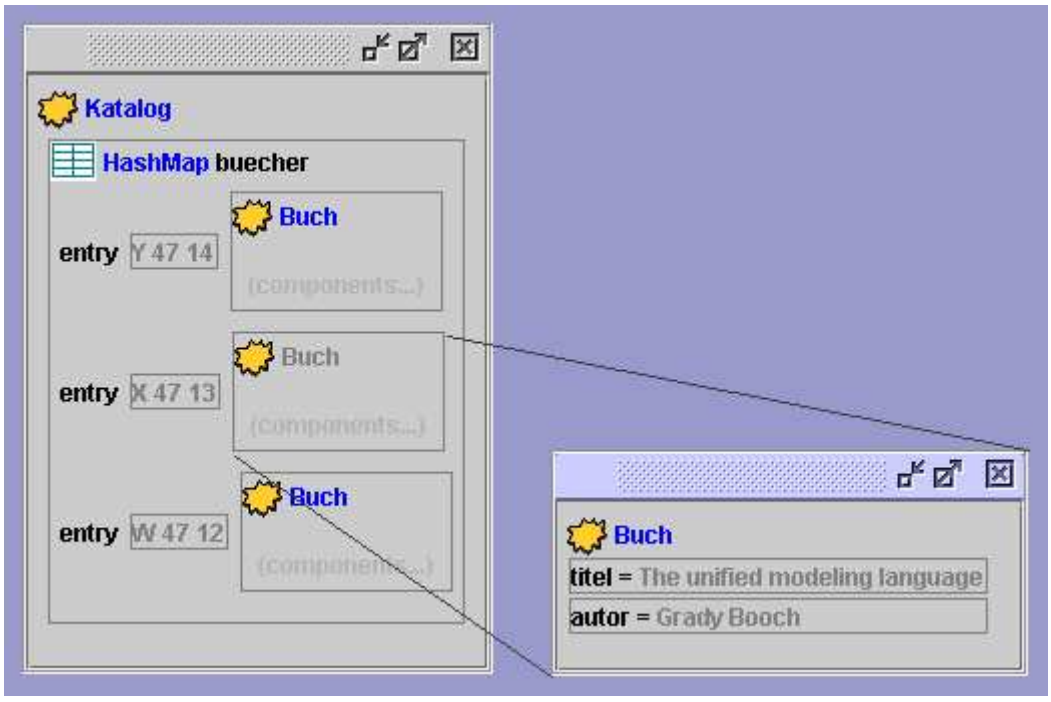


Abbildung 13 Ein Komponentendiagramm einer Komponente mit explodierter Unterkomponente.

Bei Berührung mit dem Mauszeiger werden bei primitiven Komponenten der Typ und bei komplexen Komponenten eine repräsentierende Zeichenkette angezeigt. Diese Zeichenkette ist der Rückgabewert der Methode `toString` des originalen Objekts. Der Wert hängt somit von der Implementierung dieser Methode ab und ist nur als Hinweis zu betrachten[13]. Bei Zahlenattributen wird ein Wertebereich angezeigt, sofern dieser bekannt ist (**Abbildung 14**).

Abbildung 14 Popup-Anzeige von Zusatzinformationen

Es gibt neben dem Verschieben, Minimieren und Schließen von Komponenten Fenstern verschiedene Möglichkeiten, die graphische Darstellung neu zu ordnen: Im Objektdiagramm können mit dem Menü der rechten Maustaste Darstellungen markiert, explodiert bzw. expandiert werden (**Abbildung 15**). Ebenso können alle referenzierten Objekte geöffnet werden, falls sie nicht bereits geöffnet sind.

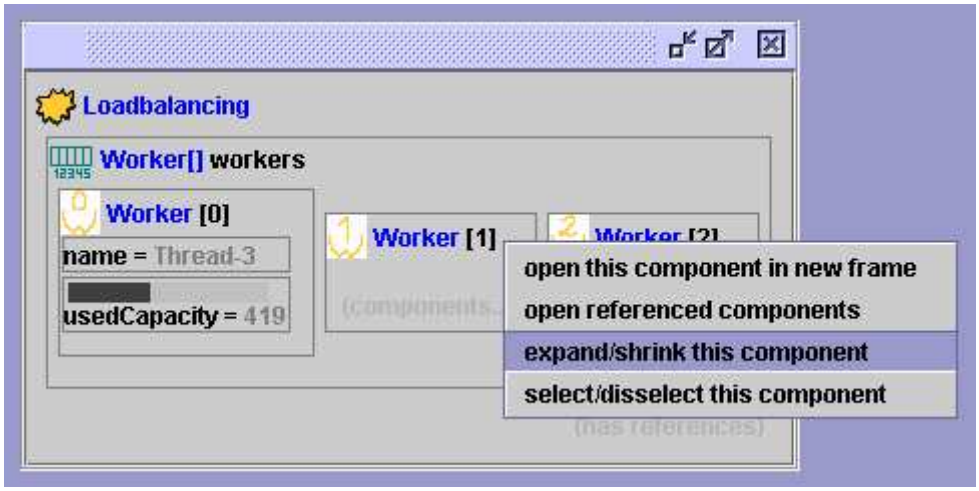


Abbildung 15 Popup-Menü bei Bedienung der rechten Maustaste

Markierungen können auch mit der linken Maustaste vorgenommen werden. Bei gedrückter Steuerungstaste können im Baum und im Objektdiagramm auch mehrere Komponenten markiert werden. Diese Markierung wird automatisch auch in allen anderen Diagrammen, die die Komponente enthalten, und in der Baum Darstellung übernommen. Die Markierung dient der persönlichen Übersicht und dem Auffinden von Komponenten des Diagramms im Baum und umgekehrt.

Kästchen von Unterkomponenten können innerhalb eines Fensters versteckt oder aufgeklappt werden (*Komprimieren bzw. Expandieren*). Dieser Vorgang kann außer über das Kontextmenü der rechten Maustaste auch durch Doppelklick mit der linken Maustaste vorgenommen werden. Im

13 Der Sinngehalt dieser Angabe hängt davon ab, wie die Methode implementiert ist. Es kann im Einzelfall vorkommen, dass die Methode für zwei Objekte den gleichen Wert zurückgibt. Falls sie nicht vom Programmierer implementiert ist, so wird die Implementierung der Klasse `Object` verwendet, die eine Zeichenkette mit dem Typ und dem Hashcode, einer „eindeutigen“ Nummerierung des Objekts, zurückgibt. Die „Eindeutigkeit“ des Hashcodes wiederum hängt davon ab, ob bzw. wie der Programmierer die Methode `hashCode` implementiert hat.

expandierten Zustand sind vorhandene Unterkomponenten innerhalb des Rahmens einer Komponente zu sehen, im komprimierten Zustand wird gegebenenfalls in hellgrauer Schrift mit der Zeichenfolge '*components...*' angedeutet, dass Unterkomponenten vorhanden sind.

Beim *Explodieren* öffnet sich ein neues Fenster mit einer (Unter-) Komponente. Das Explodieren führt dazu, dass eine Komponente in mehreren Kästchen repräsentiert wird, u.U. auch in mehr als zwei Kästchen. Wenn eine Komponente, die ein eigenes Fenster hat, auch in anderen Kästchen dargestellt wird, so wird die Schrift der anderen Darstellungen hellgrau eingefärbt. Außerdem werden von diesem Fenster aus Explosionslinien zu einem der korrespondierenden Kästchen gezeichnet. Falls mehrere in Frage kommen, wird eines davon nach folgendem Algorithmus ausgewählt: Die Kandidaten werden der Reihe nach untersucht. Es wird dasjenige Kästchen ausgewählt, dessen umschließendes Kästchen nicht hellgrau ist (**Abbildung 16**).

Das Explodieren von Unterkomponenten kann auch durch Doppelklick im Komponentenbaum geschehen. Durch Schließen oder Minimieren lassen sich Komponentenfenster aus dem Diagramm wieder ausblenden.

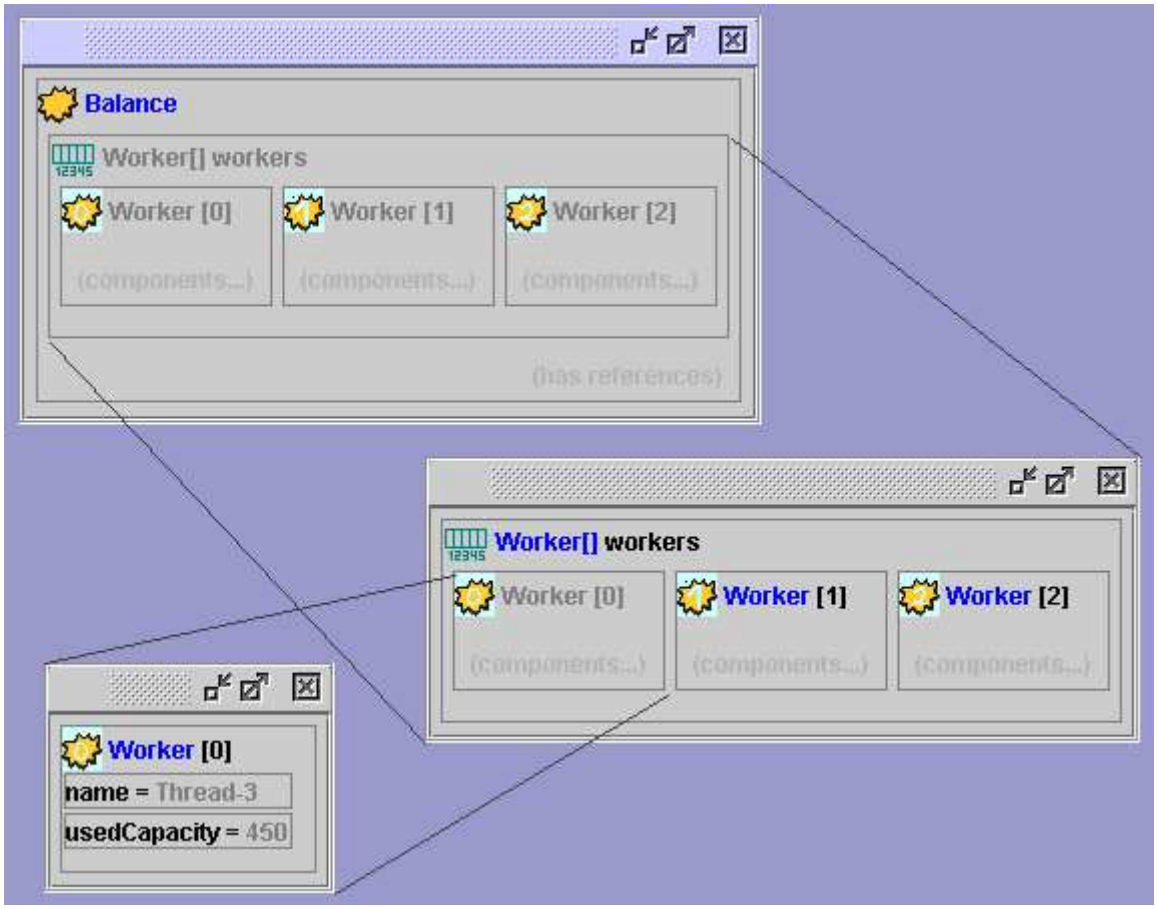


Abbildung 16 Das Fenster der Komponente '*Worker [0]*' (unten) hat Explosionslinien zu seinem korrespondierenden Kästchen im Fenster in der Mitte, weil dort die umgebende Komponente '*Worker[] workers*' nicht hellgrau ist.

Funktionalität	**Ausführen**	**Betroffene Elemente**
Fenster verschieben, in der Größe verändern, minimieren, maximieren oder schließen.	Mit Maus und linker Maustaste auf Fenstersymbole bzw. Fensterrand.	Interne Fenster von Komponenten.
Einblenden von Zusatzinformationen.	Mauszeiger über Darstellung schieben.	Kästchen von Komponenten und Attributen.

Funktionalität	Ausführen	Betroffene Elemente
Rot Markieren.	Linke Maustaste über Element in Diagramm oder Baum drücken. Mehrere Markierungen mit '*Strg-C*' während des Mausklicks möglich. Oder Punkt '*select/disselect this component*' im Kontextmenü wählen.	Kästchen von Komponenten und Attributen.
Expandieren und Komprimieren.	Doppelklick mit der linken Maustaste über Element. Oder Punkt 'expand/*disexpand this component*' im Kontextmenü wählen.	Kästchen von Komponenten.
Explodieren.	Punkt '*open this component in new frame*' im Kontextmenü wählen. Oder Doppelklick im Komponentenbaum.	Kästchen von Komponenten.

Tabelle 3 Aktionen auf den graphischen Elementen im Objektdiagramm.

4.6.3.4.2 Referenzen

Referenzen zwischen Komponenten werden durch Pfeile dargestellt, die als Beschriftung den Namen der Variable haben, welche die Referenz im Programm enthält (**Abbildung 17**). Die Variable kann im Gegensatz zur Komponentenbeziehung nicht nur ein Feld einer Klasse sein, sondern auch eine lokale Variable. Referenzen, die auf Komponenten zeigen, welche gerade nicht im Diagramm geöffnet sind, werden durch einen ganz kurzen Pfeil ohne Beschriftung am rechten Rand eines Kästchens angedeutet, Referenzen auf minimierte Komponenten durch einen ganz kurzen Pfeil am unteren Rand eines Kästchens. Wenn eine Komponente Referenzen besitzt, trägt sie in Hellgrau den Schriftzug '*has references*'.

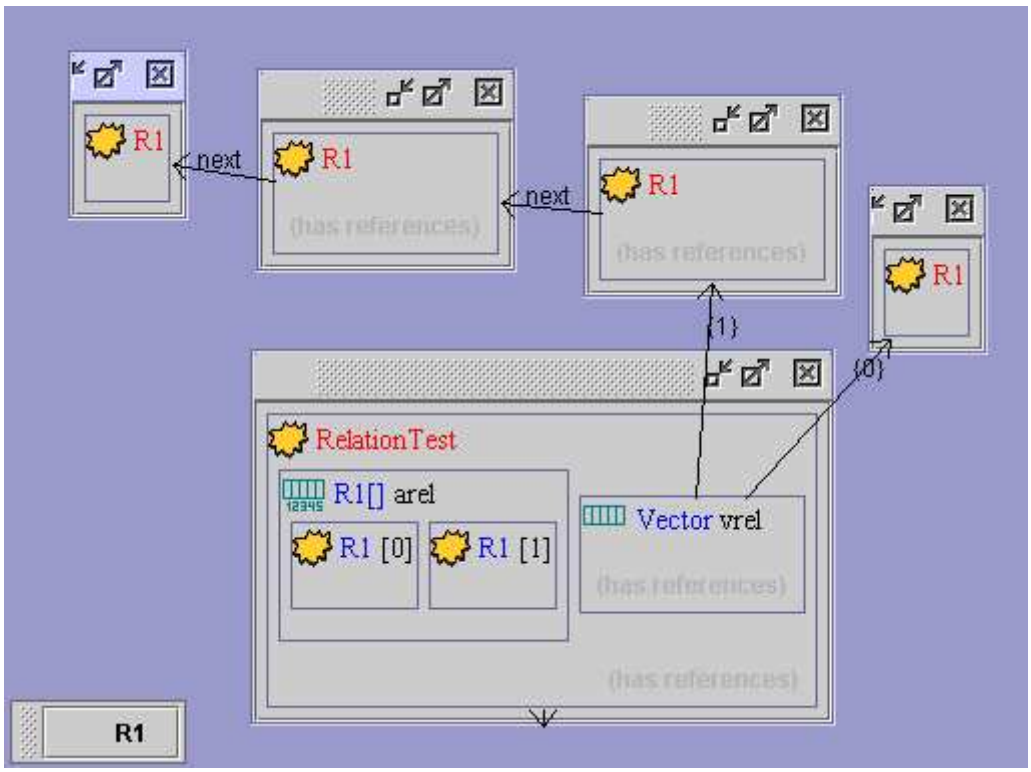


Abbildung 17 Die Komponentendiagramme mehrerer Objekte, die untereinander durch Referenzen in Verbindung stehen.

Selbstreferenzen werden nicht dargestellt. Die Darstellung von Referenzen entfällt aus Gründen der Übersichtlichkeit auch, wenn das referenzierte Kästchen sich in bzw. über demjenigen befindet, von dem die Referenz ausgeht. Dies kann der Fall sein, wenn das Fenster einer Komponente über ein anderes Fenster geschoben wird, oder bei Referenzen einer Komponente auf ihre Unterkomponenten.[14]

Bei Referenzen, die sich auf lokale Variablen beziehen, ist die Beschriftung am Pfeil in runde Klammern eingeschlossen. Behälter, d.h. Arrays, Listen und Mengen können nicht referenziert werden, sie werden automatisch als Komponenten behandelt. Falls Behälter in lokalen Variablen auftreten, ist ihr Variablenname in runden Klammern notiert. Collections und Arrays können aber sehr wohl Objekte nicht nur als Komponenten enthalten sondern auch referenzieren. Die Beschriftung am Pfeil ist dann eine laufende Nummer in geschweiften Klammern.

4.6.3.4.3 Abbildungen

Neben Mengen, Listen und Arrays gibt es noch eine besondere Form von mengenwertigen Komponenten, die in Java durch das Interface `Map` beschrieben wird. Es handelt sich dabei um Verzeichnisse, die eine Abbildung von Schlüssel Objekten zu Wert Objekten enthalten. Diese Verzeichnisse können auch dargestellt werden, wie das Beispiel der Bibliothek zeigt (siehe auch (Abschnitt 3.2 *Ein Beispiel mit Tabellen und Mengen (Bibliothek)*).

Wie in Kapitel 5 *Tags: Annotationen im Quelltext* beschrieben, kann die Visualisierung über sogenannte Tags im Quelltext gesteuert werden. Das Beispiel der Bibliothek kommt für die Darstellung der Daten vollkommen ohne Annotationen aus. Es muss nur die Grundeinstellung des System so eingestellt sein, dass Objekte bzw. Methodenaufrufe angezeigt werden.[15]

Wenn keine Tags im Quelltext verwendet werden, betrachtet das Visualisierungssystem die Beziehung zwischen der `Bibliothek` und ihren Verzeichnissen `Benutzerverzeichnis` und `Katalog` als Referenz und nicht als Aggregation bzw. Komponentenbeziehung (**Abbildung 20**). Die Verzeichnisse werden dann nicht als Unterkomponenten, sondern als Referenzen dargestellt, was in diesem Falle die Graphik optimal gestaltet.[16] Die Tabellen (`Map`) der Verzeichnisse `Benutzerverzeichnis` und `Katalog` bilden mit ihren Einträgen per Standardeinstellung jeweils eine Komponentenstruktur. Die Referenzen zwischen `Benutzer` und `Buch` sind als Pfeile dargestellt. Die Darstellungen der Komponentenbäume enthalten nur die Schlüssel der Verzeichniseinträge (**Abbildungen 18 und 19**).

Abbildungen 18 und 19 Die Darstellungen der Verzeichnisse des Beispiels *Bibliothek* als Bäume.

14 Referenzen auf Unterkomponenten können sich zusätzlich zur Komponenteneigenschaft ergeben, falls z.B. ein Objekt zunächst lokal erzeugt wird, bevor es als Unterkomponente eingetragen wird.

15 Vor dem Generieren im Menü oder in der Property Datei *trace.properties* '*defaultshowobject* = *true*' und *trace.properties* '*defaultshowcall* = *true*' setzen. Das mitgelieferte Beispiel ist schon in dieser Einstellung fertig generiert.

16 Die Eigenschaft als Komponente kann selbstverständlich durch eine Annotation im Quelltext definiert und dargestellt werden.

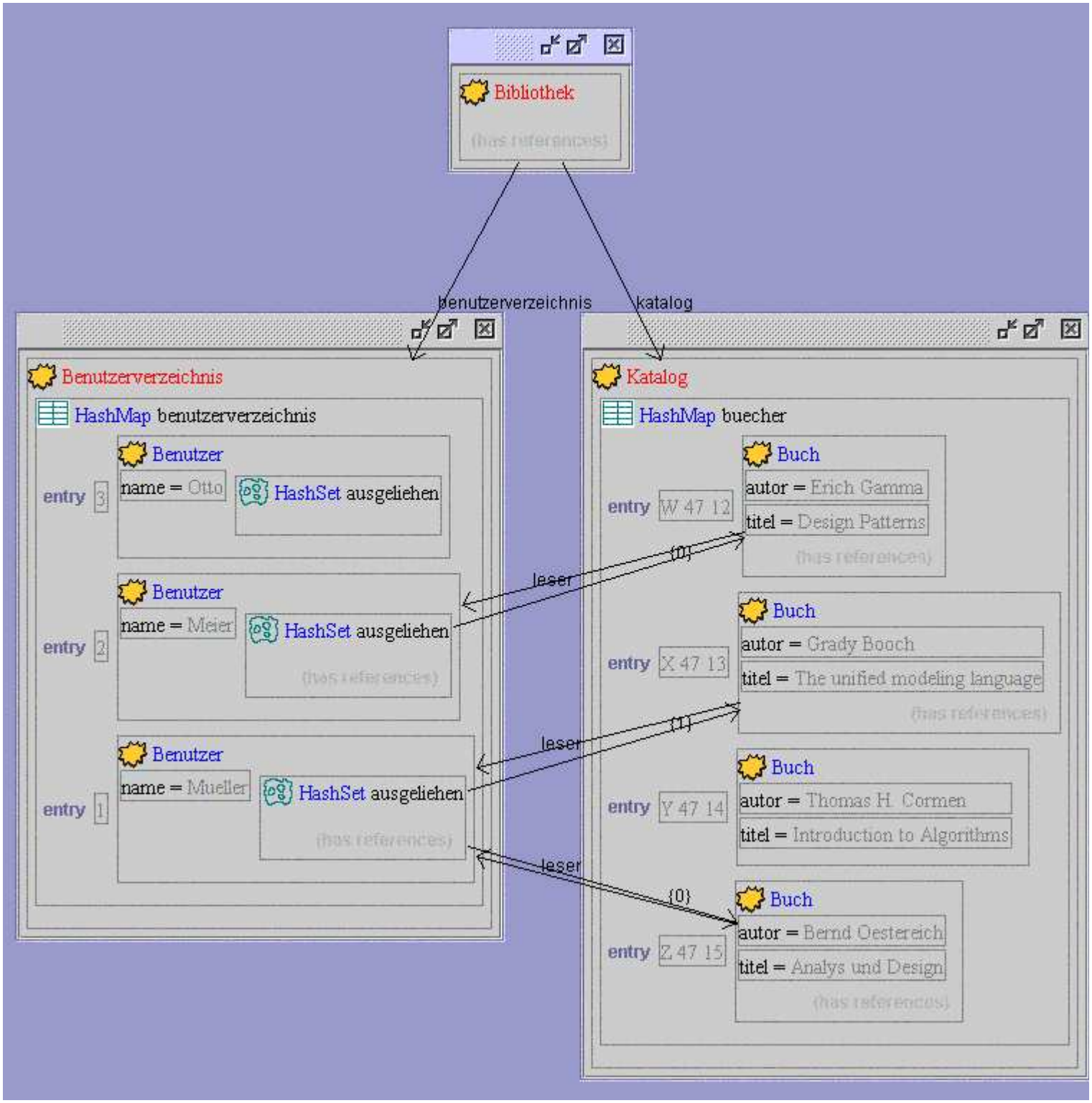


Abbildung 20 Das Komponentendiagramm des Beispiels *Bibliothek*.

4.6.3.5 Ablaufdiagramm

Das Ablaufdiagramm auf der Karte '*sequence diagram*' zeigt die Methodenaufrufe aus den Historien. Es werden nicht nur die Aufrufe eines Threads sondern die aller Threads gleichzeitig dargestellt (**Abbildung 21**). Das Diagramm gehört zum Beispiel *Loadbalancing* in Abschnitt 3.1 *Ein Beispiel mit mehreren konkurrierenden Threads*.

Aufrufende (*caller*) und aufgerufene (*callee*) Objekte und Klassen werden als kleine Fenster dargestellt. Sie lassen sich verschieben, minimieren und schließen. Objekten sind zweizeilig beschriftet, wobei in der ersten Zeile der Typ und in der zweiten der Name der Variable steht, die auf dieses Objekt als Unterkomponente verweist, falls es sich um eine Unterkomponente handelt. Klassen sind einzeilig mit ihrem Typ beschriftet und werden für statische Aufrufe verwendet.

Bei Berührung mit dem Mauszeiger erscheint auch eine textuelle Repräsentation des Objekts, die nur als Hinweis zu betrachten ist[17].

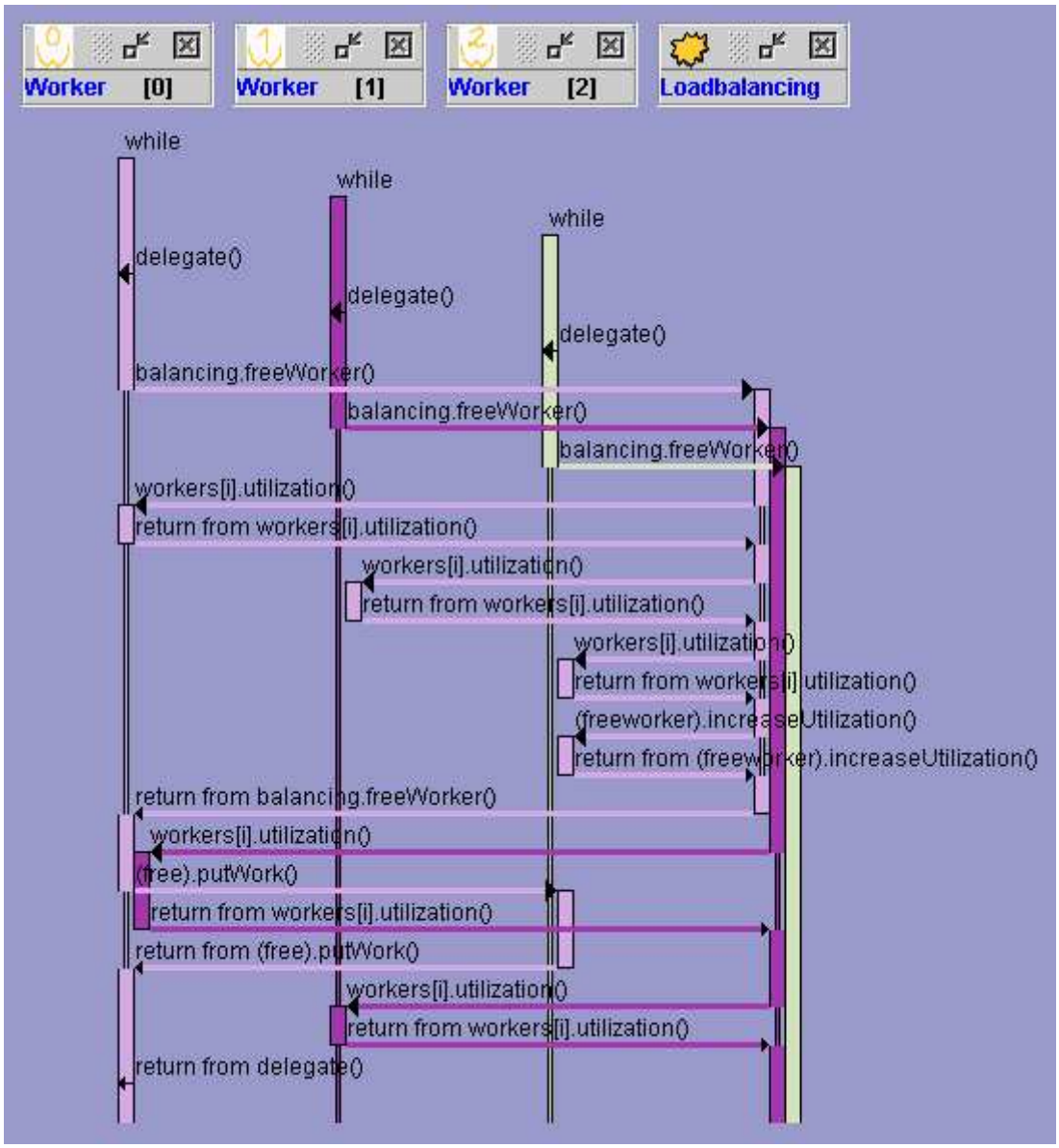


Abbildung 21 Ein Ablaufdiagramm mit drei Threads in verschiedenen Farben.

Der Ablauf wird durch horizontale Pfeile zwischen den (unsichtbaren) vertikalen Zeitschienen, die nach unten gerichtet unter den Objekten verlaufen, dargestellt. Ein Pfeil enthält als Beschriftung eine Angabe der aufgerufenen Methode und des Objekts oder der Klasse, auf der die Methode

17 Diese Darstellung ist der Rückgabeparameter der Methode `toString` des Objekts. Der Sinngehalt dieser Angabe hängt davon ab, wie die Methode implementiert ist. Falls sie nicht vom Programmierer implementiert ist, so wird die Implementierung der Klasse `Object` verwendet, die eine Zeichenkette mit dem Typ und dem Hashcode, einer „eindeutigen“ Nummerierung des Objekts, zurückgibt. Die „Eindeutigkeit“ des Hashcodes wiederum hängt davon ab, ob bzw. wie der Programmierer die Methode `hashCode` implementiert hat.

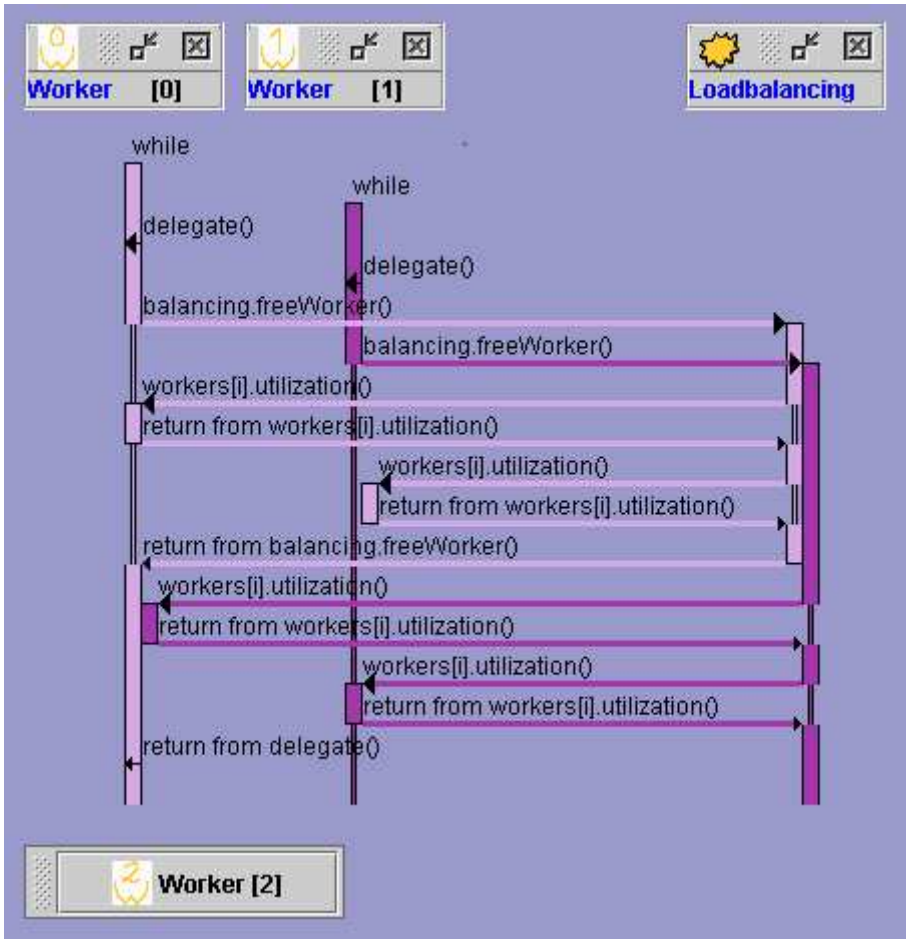
aufgerufen wurde. Die Objektangabe besteht aus der voll qualifizierten Bezeichnung der lokalen Variable oder Klassen Variable, die das aufgerufene Objekt enthält, oder einer anderen im Quelltext verwendeten Notation (z.B. `obj.getX()`). Das Beenden eines Methodenaufrufs wird mit einem Pfeil mit kleinerer Spitze in entgegengesetzter Richtung notiert.

Schleifen werden ohne Pfeil durch Angabe der Bezeichnung der Schleife bei Eintritt und Verlassen gekennzeichnet.

Der Zeitraum zwischen Beginn eines Methodenaufrufs bzw. eines Schleifeneintritts und dem Ende wird auf der senkrechten Zeitleiste durch einen Balken dargestellt. Wenn kein Balken vorhanden ist, bedeutet dies, dass das Objekt in diesem Thread in Bezug auf die gemeldeten Ereignisse nicht aktiv ist. Es ist zu bemerken, dass das Objekt im Programm dennoch aktiv sein kann, jedoch aufgrund der Annotationen im Quelltext diese Aktivität nicht gemeldet wird und somit nicht berücksichtigt werden soll.

Im Gegensatz zu anderen üblichen Darstellungen von UML Ablaufdiagrammen werden Schleifen und Methodenaufrufe, die geschachtelt innerhalb eines Objekts ablaufen, nicht als Balken nebeneinander, sondern innerhalb eines Balkens dargestellt, um die Darstellung kompakt zu halten. Die Darstellung von mehreren Balken nebeneinander auf der Zeitschiene eines Objekts symbolisiert hier mehrere Threads, die gleichzeitig auf dasselbe Objekt einwirken. Jeder Thread hat also bei jedem Objekt einen eigenen Balken, sofern er dort aktiv ist. Diese Darstellung wird dann interessant, wenn mehrere Threads parallel Methoden desselben Objekts aufrufen und diese Methoden parallel ausgeführt werden.

Die Diagramme von Objekten können mitsamt ihrer Zeitbalken im Diagramm mit der Maus verschoben werden. Sie können außerdem zur Ausblendung aus dem Diagramm minimiert oder geschlossen werden. Ein geschlossenes Objekt kann wieder geöffnet werden, indem in der Stackdarstellung auf einen Aufruf doppelt geklickt wird. Beim Schließen oder Minimieren eines Objekts wird das Ablaufdiagramm ohne die Aufrufe, die von dem Objekt ausgehen oder das Objekt treffen, neu gezeichnet. Das Ablaufdiagramm erscheint dann in der Höhe gestaucht (**Abbildung 22**).



4.7 Ändern von Grundeinstellungen

Änderungen an den Grundeinstellungen können im laufenden System und auch dauerhaft in Property Dateien vorgenommen werden. Änderungen über das Property-Menü bei laufendem System sind nicht persistent und nur solange gültig, bis die Applikation beendet wird.

Es gibt ein vielfältiges Repertoire an Grundeinstellungen, die nur in seltenen Fällen geändert werden müssen. Diese Einstellungen befinden sich in Textdateien mit der Endung '*.properties*' im Ordner '*JAN\Properties*'. Der mögliche Inhalt ist in den **Tabellen 4, 5, 6** dargestellt. In der Datei '*path.properties*' sind Pfadangaben aufgeführt. Es wird empfohlen, relative Pfade zu verwenden, da diese nicht von den individuellen Gegebenheiten des verwendeten Dateisystems abhängen. Absolute Pfade können aber auch angegeben werden. Ein absoluter Pfad beginnt in Windows mit einer Laufwerksangabe (z.B. '*D:/Java/jdk1.4/jre/lib/rt.jar*'). Während die Anwendung ausgeführt wird, sollten die Property Dateien nicht editiert werden.

path.properties	
javapath = Java/jdk1.4/jre/lib/rt.jar;Java/jdk1.4/lib/tools.jar	Pfad zur Java Laufzeitumgebung.
baratpath = Extern/Barat/Barat.jar;Extern/Barat/BCEL.jar	Pfad zu *Barat*.
toolpath = Program/visualization.jar;Properties	Pfad zu den Klassen des Visualisierungssystems.
sourcepath = Sandbox/Source	Pfad zu den originalen Quellen der Sandbox.
classpath = Sandbox/Classes	Pfad zu den originalen Klassen der Sandbox.
gensourcepath = Sandbox/Generated/Source	Pfad zu den generierten Quellen.
genclasspath = Sandbox/Generated/Classes	Pfad zu den generierten Klassen.

Tabelle 4 Properties zum Definieren der Pfade.

trace.properties	
modifieriscomponent = 0	Bestimmung von Komponenten
defaultshowobject = true	Objekte anzeigen.
defaultshowlocalobject = false	Lokale Objekte nicht anzeigen.
defaultshowcall = false	Methodenaufrufe nicht anzeigen.
defaultshowloop = false	Schleifen nicht anzeigen.
traceclass = visualization.tracing.modificationtracing.Trace	Klasse der Ablaufverfolgung
traceclassforsystemhistory = visualization.communication.channel.TraceControlProxy	Für die `SystemHistory` sichtbare Klasse der Ablaufverfolgung
generatorclass = visualization.tracing.modificationtracing.Generator	Klasse des Generators.
generatorinownprocess = true	Generator in eigenem Prozess.

Tabelle 5 Properties für die Ablaufsteuerung.

Das Property `modifieriscomponent` bestimmt, ob Felder mit bestimmten Eigenschaften als Komponenten dargestellt werden sollen. Das Property kann folgende Werte haben:

- public=0x0001
- protected=0x0004
- package=0x1000
- private=0x0002

- static=0x0008
- final=0x0010
- volatile=0x0040
- transient=0x0080

Die Eigenschaft package ist vorhanden, wenn kein Zugriffsrecht an einem Feld steht. Die Werte des Properties können durch Addition kombiniert werden. So wird z.B. für die Eigenschaft final und protected der Wert 0x0010+0x0004=0x0014 verwendet. Der Wert 0 wird verwendet, wenn keine Interpretation von Komponenten vorgenommen werden soll, wenn also kein Modifier berücksichtigt werden soll.

gui.properties	
animationperiod = 1000	Zeitspanne zwischen zwei Schritten der Animation in ms.
showlastchange = true	Letzte Änderung im Objektdiagramm markieren.
autoopendiagrams = true	Diagramme automatisch öffnen.
shortprint = true	Ausgabe von kurzen Typbezeichnungen ohne Paketstruktur.
printthis = false	Ausgabe von „this.“
widthofcodeframes = 500	Automatische Begrenzung der Breite von Quelltext Fenstern.
heightofcodeframes = 250	Automatische Begrenzung der Höhe von Quelltext Fenstern.
maxwidthofframes = 400	Automatische Begrenzung der Breite von Fenstern.
maxheightofframes = 200	Automatische Begrenzung der Höhe von Fenstern.
autofitdimofcompframes = true	Komponenten Fenster automatisch in der Größe Anpassen.
autoscroll = false	Ablaufdiagramm automatisch rollen.
Object = Object.gif	Bild für Gruppenbezeichnung.
Loadbalancing = Object.gif	Bild für Gruppenbezeichnung.
Worker = Object.gif	Bild für Gruppenbezeichnung.
Array = Array.gif	Bild für Gruppenbezeichnung.
Set = Set.gif	Bild für Gruppenbezeichnung.
List = List.gif	Bild für Gruppenbezeichnung.
Map = Map.gif	Bild für Gruppenbezeichnung.
Object0 = Object0.gif	Bild für Gruppenbezeichnung.
Object1 = Object1.gif	Bild für Gruppenbezeichnung.
Object2 = Object2.gif	Bild für Gruppenbezeichnung.
Object3 = Object3.gif	Bild für Gruppenbezeichnung.
Worker0 = w0.gif	Bild für Gruppenbezeichnung.
Worker1 = w1.gif	Bild für Gruppenbezeichnung.
Worker2 = w2.gif	Bild für Gruppenbezeichnung.
Worker3 = w3.gif	Bild für Gruppenbezeichnung.
Book = book.gif	Bild für Gruppenbezeichnung.
User = user.gif	Bild für Gruppenbezeichnung.

Tabelle 6 Properties für die Darstellung.

5 Tags: Annotationen im Quelltext

Im Quelltext können spezielle Kommentare, sogenannte Tags angebracht werden, die vom Visualisierungstool automatisch ausgewertet werden. Tags können vor Klassen-, Feld- und Variablendeklarationen, sowie vor Anweisungen stehen. Abhängig von den untersuchten Programmen kann u.U. auch auf Tags verzichtet werden.

Es ist immer wieder von Feldern und Variablen die Rede. Ein Feld ist in diesem Dokument eine Variable, die auf Klassenebene deklariert ist. Es gibt Felder, die zum Objekt einer Klasse gehören und statische Felder, die allen Objekten einer Klasse gemeinsam gehören. Eine Variable ist lokal innerhalb einer Methode deklariert.

Es werden Tags im Java Doc Format verwendet, um mit anderen Java typischen Kommentaren kompatibel zu sein. Kommentare mit der Notation `/**@<Kommentarname> <Kommentarwert>*/` werden bei der Erzeugung einer Java Doc Dokumentation aus dem Quelltext automatisch mit in die Dokumentation aufgenommen.

Die Kommentarform `/**@<Kommentarname> <Kommentarwert>*/` ist mit der Form

```
/**
 * @<Kommentarname> <Kommentarwert>
 */
```

gleichbedeutend.[18] Bei letzterer sind die einzelnen Kommentare durch Zeilenumbrüche getrennt. Wenn der Kommentarwert Leerzeichen enthält, werden die einzelnen Bestandteile als Elemente einer Aufzählung behandelt.

Es werden zwei zueinander orthogonale Gruppen von Tags unterschieden, zum einen die Tags zur Semantikergänzung und zum anderen die Tags zur Auswahl und Beschreibung der darzustellenden Elemente.

5.1 Tags zur Semantikergänzung

Es gibt gewisse semantische Inhalte wie den Komponentenbegriff, die mit Java nicht ausgedrückt werden können, aber für die Visualisierung interessant sind. Diese Semantik kann durch entsprechende Tags ergänzt werden. Da es sich bei dem Visualisierungstool weder um einen Compiler noch um eine Java Laufzeitumgebung handelt, ist allerdings keine automatische Überprüfung der in einem Tag benannten Semantik vorgesehen. Wenn beispielsweise in einem Tag ein Definitionsbereich für ein Integer Feld angegeben ist, wird nicht garantiert, ob im Programm dieser Bereich auch wirklich eingehalten wird. Der Definitionsbereich wird lediglich die Darstellung angegeben. Ebenso bezeichnet die Kennzeichnung eines Feldes als Komponente zwar eine semantische Eigenschaft, diese wird von dem Visualisierungssystem aber auch nur für die Darstellung verwendet. Die Tags zur Semantikergänzung definieren also gewissermaßen, *wie* ein Feld (abhängig von seiner Semantik) dargestellt werden soll.

Tag	**Markiertes Objekt**	**Position im Code**
`/**@range <untere Grenze>..<obere Grenze>*/`	Primitives Zahlenattribut	Felddeklaration
`/**@component*/`	Objekt	Felddeklaration

Tabelle 7 Tags zur Semantikergänzung.

18 **Achtung**: Aufgrund eines Fehlers in Barat sollte vor den Tags eine Zeile mit mindestens einem Zeichen stehen. Ansonsten kann es vorkommen, dass nicht alle Tags gelesen werden. Beispiel:

```
/**
 * -
 * @tag1 value1
 * @tag2
 */
```

5.1.1 Definitionsbereich

Der Definitionsbereich definiert die obere und untere Grenze für die Werte eines Feldes. Diese Angaben können für die Darstellung in einem Balkendiagramm verwendet werden. Das Tag wird nur an Felddeklarationen ausgewertet, die einen primitiven Zahlentyp definieren (`short, int, long, float, double`). Die Angaben der unteren und oberen Grenze sind durch ein Leerzeichen vom Tag Bezeichner und durch zwei Punkte voneinander getrennt. Es können Zahlen und Variablen verwendet werden. Variablen müssen an dieser Stelle im Programm aber schon bekannt sein.

5.1.2 Komponentensemantik

Eine Komponente ist ein *Unterobjekt*, d.h. ein Objekt, welches als *Teil eines größeren* zu verstehen ist. Eine Komponente wird i.d.R. von seinem Besitzer erzeugt. Wenn der Besitzer vernichtet wird, verschwinden auch seine Komponenten. Eine Komponente ist also von seinem Inhaber *existenzabhängig*. Der hier verwendete Komponentenbegriff soll nicht mit dem in der komponentenbasierten Softwarearchitektur verwechselt werden, wie sie beispielsweise bei Enterprise-Java-Beans (EJB) Verwendung findet.

Die Existenzabhängigkeit kann in Java nicht mit Mitteln der Sprache ausgedrückt werden.[19] Ob eine Komponente in einer Programmiersprache in by-reference oder by-value Semantik (Schlüsselwort `expanded` in Eiffel) gehalten wird, soll hier deshalb ohne Bedeutung sein.[20] Ob eine Komponente existenzabhängig oder beständig ist, wird vom System nicht überprüft. Die Auszeichnung als Komponente soll hier die Darstellung beeinflussen.

Ebenso ohne Bedeutung soll die Frage sein, ob eine Komponente nur für den Besitzer sichtbar ist, oder vom Besitzer nach außen gegeben werden kann. Im letzten Falle können also Referenzen von beliebigen Objekten auf die (Unter-) Komponente bestehen. Es ist sinnvoll, dass eine Komponente genau dann sichtbar für außen stehende Objekte ist, wenn sie `public` deklariert ist. Komponenten, die nach außen nicht sichtbar sind, könnten aber durch *Hüllobjekte* (*Wrapper*) sichtbar werden. Wrapper können die Zugriffsmöglichkeiten auf ein Originalobjekt einschränken.

Komponenten werden im Objektdiagramm innerhalb ihres Besitzers dargestellt, d.h. innerhalb des Kästchens des Besitzers als eigenes Kästchen. Es kann zwischen komplexen und primitiven Komponenten unterschieden werden: Komplexe Komponenten sind beliebige Objekte, die ihrerseits wiederum Unterkomponenten haben können. Primitive Komponenten sind Attribute wie Strings oder Zahlen, die eine komplexe Komponente beschreiben.

Der Verweis zu einer Komponente wird in einem Feld[21] einer Klasse gehalten und dort mit dem semantischen Tag `/**@component*/` verdeutlicht. Im Klassendiagramm entspricht solch ein Verweis einer besonderen Assoziation, der *Aggregation*. Komplexe Objekte werden nicht als Komponente aufgefasst und nicht als solche visualisiert, wenn sie nicht das Tag `/**@component*/` tragen. In diesem Fall entspricht der Verweis im Klassendiagramm einer einfachen *Assoziation*. Hier wird in diesem Falle von *Referenz* gesprochen. Im Objektdiagramm werden dafür Pfeile zwischen den Kästchen der Objekte verwendet.

Die Kennzeichnung an einer lokalen Variable mit dem Tag `/**@component*/` ist nicht gültig, da eine Unterkomponente zu ihrer Oberkomponente gehören soll. Lokale Variablen enthalten referenzierte Objekte und bezeichnen keine Unterkomponenten. Eine Ausnahme wird mit Behältern und primitiven Typen gemacht. Diese können nicht referenziert werden und müssen daher als Komponenten gelten. Die Eigenschaft als „lokale Komponente“ wird in der Visualisierung durch

19 Unterobjekte werden in Java immer als Verweise repräsentiert. Die Semantik des Enthaltenseins bzw. der Existenzabhängigkeit müsste durch entsprechenden Code nachgebaut werden. Der Compiler und das Laufzeitsystem können diese Semantik nicht überprüfen.

20 In UML wird zwischen Komponenten und Kompositen unterschieden. Dort sind Kompositen existenzabhängige Unterobjekte, Komponenten jedoch Unterobjekte, die auch ohne ihren Besitzer leben können. Kompositen sollten sinnvoller Weise in by-value Semantik, Komponenten in by-reference Semantik von einer Programmiersprache behandelt werden.

21 Ein Feld ist eine Variable, die auf Klassenebene deklariert ist. Es gibt Felder, die zum Objekt einer Klasse gehören und statische Felder, die allen Objekten einer Klasse gemeinsam gehören. Felder können auch als Attribute bezeichnet werden, da sie ein Objekt näher beschreiben.

einen Variablennamen in runden Klammern gekennzeichnet. Bei primitiven Attributen ist ferner eine Kennzeichnung mit dem Tag `/**@component*/` auch an einer Felddeklaration nicht notwendig, da sie immer als Komponenten aufgefasst werden.

Eine Komponente darf nur an einer Stelle definiert sein, eine Kennzeichnung mehrerer Felder in einer oder in verschiedenen Klassen als Komponente ist also nur erlaubt, wenn den Feldern *nicht* dasselbe komplexe Objekt zugewiesen wird![22] Es gibt jedoch statische Komponenten, die allen Objekten einer Klasse gehören, wenn das Feld `static` deklariert ist.

Falls eine Komponente nicht wiederum Teil einer anderen Komponente ist, so wird sie als Wurzel eines Komponentenbaums behandelt.

Für Arrays und Collections gelten besondere Regeln. Arrays werden grundsätzlich nur als Komponenten verwendet und enthalten nur Komponenten. Eine Kennzeichnung des Arrays mit dem Tag `/**@component*/` kann daher entfallen. Collections werden auch nur als Komponenten verwendet, können jedoch selbst Referenzen und Komponenten enthalten. Eine Collection muss also nicht als Komponente ausgezeichnet werden. Jedoch müssen die Elemente gekennzeichnet werden. Da in der Klasse der Collection keine Tags angebracht werden können, wird hierzu die Felddeklaration der Collection in ihrer Oberkomponente verwendet. Eine Kennzeichnung einer Collection mit dem Tag `/**@component*/` bedeutet, dass ihre Elemente Komponenten sein sollen. Ohne dieses Tag enthält die Collection Referenzen.

Ein Array oder eine Collection kann nicht Wurzel eines Komponentenbaums sein (siehe Abschnitt) oder referenziert werden. Daher werden Behälter, die in lokalen Variablen deklariert sind, auch als Komponenten behandelt. Dies ist insofern eine Besonderheit, als dass solche Objekte nur temporär auftreten.[23] Die Elemente einer lokalen Collection sind immer Referenzen, da das Tag `/**@component*/` an einer lokalen Variable ungültig ist.

	Tag `/**@component*/` an Felddekaration verwendet	**Ohne Tag**
Beliebiges Objekt (außer s.u.)	Komponente	Referenz
Attribut primitiven Typs (`int`,...) einschließlich Hülltypen (`Integer`,...) und `String`	Komponente	Komponente
Array	Array und seine Elemente sind Komponenten.	Array und seine Elemente sind Komponenten.
Collection	Collection und ihre Elemente sind Komponenten.	Collection ist Komponente, ihre Elemente sind Referenzen

Tabelle 8 Verwendung und Bedeutung des Tags `/**@component*/`.

Es ist zu bemerken, dass eine inkonsistente Verwendung des Tags `/**@component*/` zu Inkonsistenzen in der Darstellung des inspizierten Programmes führt. Es muss sichergestellt sein, dass ein und dasselbe komplexe Objekt nicht verschiedenen Feldern zugewiesen wird, die das Tag `/**@component*/` tragen! Bei Collections mit dem Tag `/**@component*/` muss sichergestellt sein, dass ihnen nicht ein Objekt hinzugefügt wird, welches als Komponente gekennzeichnet ist oder zu einem Array gehört![24] Auch darf dieselbe Komponente in einem oder mehreren Arrays nicht mehrfach abgelegt werden. Das gilt nicht für primitive Komponenten, da diese bei mehrfachem Eintrag als verschiedene Objekte gewertet werden.

Wie in Abschnitt beschrieben, kann ein Behälter des weiteren keine Behälter enthalten. Die Ablaufverfolgung sorgt hier automatische dafür, dass diese Operationen nicht gemeldet werden.[25]

22 Eine Komponente ist nicht *shared*.

23 „Lokale Komponenten“ werden in den Diagrammen durch einen Variablennamen in runden Klammern gekennzeichnet.

24 Bei Verstoß gegen diese Regelungen wird der Variablenname der Komponente während des Programmlaufs verändert, bzw. gelöscht. Wie bereits geschrieben, kann die Semantik der Verwendung des Komponentenbegriffs vom Compiler oder der JVM nicht automatisch überprüft werden.

5.1.2.1 Automatische Erkennung von Komponenten

Es ist wahlweise auch eine automatische Erkennung von Komponenten vorgesehen. Eine Heuristik, die Komponenten beschreibt, kann (über das Property-Menü des Visualisierungssystems) angegeben werden. So ist es z.B. möglich, Felder, die `final` deklariert sind oder ein bestimmtes Zugriffsrecht haben, als Komponenten zu behandeln. Bei Angabe eines Zugriffsrechts werden alle Felder mit schwächerem Recht auch als Komponente behandelt. Wenn ein Feld z.B. `protected` ist, so werden alle `private`, `package` und `protected` Felder als Komponenten dargestellt. Es sei angemerkt, dass diese Heuristiken nur Sinn machen, wenn der Programmierer des zu visualisierenden Programmes sich an einen entsprechenden Programmierstil hält. So kann es sinnvoll sein, `final` oder `private` Felder als Komponenten zu betrachten, weil die referenzierten Objekte nicht verändert bzw. für andere Objekte nicht sichtbar sind. Bei Verwendung dieser Heuristiken kann es genauso wie bei falscher Verwendung des Tags `/**@component*/` bei der Darstellung zu Inkonsistenzen kommen, wenn dasselbe Objekt von mehreren Feldern referenziert wird, da auch hierbei keine Überprüfung der Semantik vorgenommen wird.

5.2 Tags zur Auswahl und Beschreibung der darzustellenden Elemente

Die Tags zur Auswahl und Beschreibung der darzustellenden Elemente definieren, *was* dargestellt werden soll. Felder, einzelne Referenzierungen eines Feldes oder einer Variable, Schleifen und Methodenaufrufe können explizit in die Visualisierung aufgenommen oder von ihr ausgeschlossen werden. Desweiteren können Klassen und einzelnen Objekten Gruppen zugeordnet werden, die an bestimmte Symbole oder Bilder für ihre Darstellung gebunden werden können.

Tag	**Markiertes Objekt**	**Position im Code**
`/**@hide*/`, `/**@hideobject*/`	Unterobjekt, Referenz	Felddeklaration, referenzierendes Statement
`/**@show*/`, `/**@showobject*/`	Unterobjekt, Referenz	Felddeklaration, referenzierendes Statement
`/**@hide*/, /**@hidecall*/`	Methodenaufruf	Statement mit Methodenaufruf
`/**@show*/, /**@showcall*/`	Methodenaufruf	Statement mit Methodenaufruf
`/**@hide*/, /**@hideloop*/`	Schleife	Statement einer Schleife
`/**@show*/, /**@showloop*/`	Schleife	Statement einer Schleife
`/**@group <group>*/`	Klasse, Objekt	Klassendefinition, Feld-/ Variablendeklaration mit Objekterzeugung

Tabelle 9 Tags zur Auswahl und Beschreibung der zu visualisierenden Elemente.

5.2.1 Anzeige von Objekten

Die Tags `/**@showobject*/` bzw. `/**@hideobject*/` bestimmen, ob der Inhalt einer Variable oder eines Feldes, d.h. eine primitive oder komplexe Komponente oder eine Referenz visualisiert, bzw. von der Visualisierung ausgeschlossen werden soll. Primitive Typen werden als Attribute dargestellt, komplexe Typen (Objekte), die nicht mit `/**@component*/` markiert sind, werden durch Verweise dargestellt.

Je nach Voreinstellung braucht entweder nur `/**@hideobject*/` oder nur `/**@showobject*/` verwendet zu werden. Wenn die Grundeinstellung z.B. vorsieht, Variablen anzuzeigen, dann müssen unerwünschte Variablen mit `/**@hideobject*/` versteckt werden. `/**@hideobject*/` bzw. `/**@showobject*/` können direkt vor Felddefinitionen und Anweisungen eingefügt werden.

Erste Möglichkeit: Ein Tag steht vor der Definition eines Feldes einer Klasse, wobei auch statische Felder zugelassen sind. In diesem Falle kann angewiesen werden, alle Änderungen des Feldes zu überwachen.

25 Im inspizierten Programm kann es solche Konstrukte sehr wohl geben, sie werden aber nicht visualisiert.

Zweite Möglichkeit: Ein Tag steht vor einer schreibenden Anweisung mit einer Referenzierung eines Feldes oder einer lokalen Variable. In diesem Falle wird nur diese Anweisung überwacht. Es wird angemerkt, dass eine markierte Variable nicht `public` deklariert sein muss.

Beide Tags können durch ihre Kurzformen `/**@hide*/` bzw. `/**@show*/` ersetzt werden.

5.2.2 Anzeige von Methodenaufrufen

Die Tags `/**@showcall*/` bzw. `/**@hidecall*/` bestimmen, ob eine Nachricht, d.h. eine aufgerufene Methode dargestellt werden soll. Je nach Voreinstellung braucht entweder nur `/**@showcall*/` oder nur `/**@hidecall*/` verwendet zu werden. Wenn die Grundeinstellung z.B. vorsieht, Methodenaufrufe nicht anzuzeigen, dann müssen erwünschte Aufrufe mit `/**@showcall*/` gekennzeichnet werden.

Die Tags werden direkt vor der Anweisung eingefügt, die den Methodenaufruf enthält. Beide Tags können durch ihre Kurzformen `/**@show*/` bzw. `/**@hide*/` ersetzt werden.

5.2.3 Anzeige von Schleifen

Die Tags `/**@showloop*/` bzw. `/**@hideloop*/` bestimmen, ob eine Schleife[26] oder bedingte Anweisung dargestellt werden soll. Je nach Voreinstellung braucht entweder nur `/**@showloop*/` oder nur `/**@hideloop*/` verwendet zu werden. Wenn die Grundeinstellung z.B. vorsieht, Schleifen nicht anzuzeigen, dann müssen erwünschte Schleifen mit `/**@showloop*/` gekennzeichnet werden.

Die Tags werden direkt vor der Schleife eingefügt. Beide Tags können durch ihre Kurzformen `/**@show*/` bzw. `/**@hide*/` ersetzt werden.

5.2.4 Gruppenzugehörigkeit

Das Tag `/**@group <group>*/` weist einer Klasse oder einem Objekt eine Gruppe zu. Gruppen können wiederum mit bestimmten Bildern in Verbindung gebracht werden, die dann im Diagramm zu sehen sind. Die Bilder müssen im *gif* oder *jpg* Format im Ordner '*JAN\Extern\graphics*' abgelegt sein. Die Zuordnung zwischen der Gruppenbezeichnung `<group>` und dem Dateinamen des Bildes wird in der Property Datei '*gui.properties*' vorgenommen.[27] Beispielsweise steht dort

```
Object = Object.gif
```

wenn die Gruppenbezeichnung wie folgt definiert wurde:

```
/**
 * @group "Object"
 */
class MyClass {...}
```

Das Tag kann vor Klassendeklarationen stehen, sowie vor Feld-/ und Variablendeklarationen, wenn diese eine Objekterzeugung (z.B. `Object o = new Object();`) beinhalten. Für die Gruppenbezeichnung `<group>` wird normalerweise eine Zeichenkette verwendet, die in Anführungszeichen eingeschlossen sein muss. Bei Feld-/ und Variablendeklarationen kann außerdem (ohne Anführungszeichen) eine Variable verwendet werden, sofern diese an dieser Stelle im Programm bekannt ist. Es sind auch Kombinationen aus einer Variable und einer Zeichenkette zulässig, Variable und Zeichenkette müssen dann durch ein Pluszeichen ohne Leerzeichen voneinander getrennt sein. Diese Möglichkeit kann verwendet werden, wenn innerhalb einer Schleife ähnliche Objekte in eine Menge gefüllt werden, die aber unterschiedliche Symbole tragen sollen.

---

26 Es wird der Einfachheit halber hier oft nur von 'Schleife' gesprochen, Die bedingte Anweisung `if` ist i.d.R. auch damit gemeint.

27 Die Datei '*gui.properties*' steht im Ordner '*JAN\Properties*'. Die Einstellungen müssen vor dem Starten der Applikation vorgenommen werden.

Eine Gruppenbezeichnung vor einer Objekterzeugung überschreibt eine Bezeichnung an der Klasse des Objekts. Es soll angemerkt werden, dass sich deshalb das Bild eines Objekts während des Programmlaufs ändern kann.[28]

Es sollte darauf geachtet werden, Bilder einer sinnvollen Größe (z.B. 20 mal 20 Pixel) zu verwenden, da diese nicht skaliert werden.

5.2.5 Anmerkung

In einigen Fällen ist es erforderlich, mehrere Tags an einer Anweisung unterzubringen. In diesen Fällen sollten nicht die Kurzformen `/**@hide*/` oder `/**@show*/` verwendet werden, um Mehrdeutigkeiten zu vermeiden. Wird z.B. statt mehrerer Tags nur ein `/**@show*/` verwendet, so werden alle möglichen Einzelheiten der Anweisung verfolgt.

Es sei darauf hingewiesen, dass nicht beliebig viele Vorgänge innerhalb einer Anweisung verfolgt werden können, wie es in folgender Anweisung der Fall wäre:

```
/**
 * -
 * @showloop
 * @showcall
 */
if (object.condition1() && object.condition2()) {...}
```

Die erlaubte Struktur von Anweisungen ist im folgenden Abschnitt beschrieben. Es ist daher gegebenenfalls sinnvoll, ein zu inspizierendes Programm entsprechend abzuändern!

Einer einzelnen Struktur im Programm können aber i.A. *mehrere* Eigenschaften zugewiesen werden, wie in folgendem Beispiel:

```
/**
 * -
 * @show
 * @component
 * @group „somegroup“
 */
Object obj;
```

5.2.5.1Erlaubte Syntax zusammengesetzter Anweisungen

Anweisungen[29] im inspizierten Programm mit mehreren zu berichtenden Details müssen in mehrere Anweisungen mit jeweils nur einem Aufruf der Visualisierung zerlegt werden, wenn alle Details verfolgt werden sollen. Die Zerlegung wird für Anweisungen bestimmter Struktur automatisch vorgenommen.

Nicht unterstützt wird z.B. das Berichten über die Variable n in folgender Anweisung:

```
for(this.n=3; n<100; n++) {...}
```

Desweiteren wird z.B. das Berichten über die Methodenaufrufe in der Bedingung der folgenden Anweisung nicht unterstützt:

```
if(cond1() < cond2()) {...}
```

28 Eine Objekterzeugung wird erst gemeldet, wenn der Konstruktor vollständig abgearbeitet ist. Während der Abarbeitung wird aber u.U. schon eine Unterkomponente an das neue Objekt gehängt. Dieser Vorgang wird dem System gemeldet und visualisiert. Dazu wird die beinhaltende Komponente auch schon dargestellt, weil von einer Erzeugung bereits ausgegangen werden kann. Die Nachricht über die Gruppe dieser Komponente ist aber noch nicht eingegangen.

29 Eine Anweisung ist eine Struktur im Programm, die mit einem Semikolon abschließt oder eine `if`, `for` `while` oder `do` Anweisung.

Struktur in Barat	**Beispiel**
Assignment+VariableAccess	`this.name=n;`
Assignment+Literal	`this.name=“Hans“;`
Assignment+ObjectAllocation	`this.name=new String(“Hans“);`
InstanceMethodCall+VariableAccess	`this.names.add(n);`
InstanceMethodCall+Literal	`this.names.add(“Hans“);`
InstanceMethodCall+ObjectAllocation	`this.names.add(new String(“Hans“));`

Tabelle 10 Erlaubte Anweisungen für Komponenten und Referenzen.

Struktur in Barat	**Beispiel**
AMethodCall	`obj.putName(<p>);`
Assignment+AMethodCall	`this.name=obj.getName(<p>);`
VariableDeclaration+Assignment +AMethodCall	`String name=obj.getName(<p>);`
If+AMethodCall	`if(cond(<p>)){...}`

Tabelle 11 Erlaubte Anweisungen für die Verfolgung von Methodenaufrufen.

<p> kann ein oder mehrere Variablen (VariableAccess) oder Literale (Literal) enthalten.

5.3 Beispiele

- Die Bedingung und der Aufruf der Methode innerhalb der Bedingung sollen verfolgt werden.

```
/**
 * -
 * @showloop
 * @showcall
 */
if (object.method()) {...}
```

- Ein Objekt und der erzeugende Methodenaufruf sollen dargestellt werden. Ob es sich um eine Komponente handelt, steht gegebenenfalls bei der Deklaration der Variable.

```
/**
 * -
 * @showobject
 * @showcall
 */
obj = object.method();
```

- Ein Objekt wird als Komponente deklariert. Diese soll auch visualisiert werden. Desweiteren wird der Komponente eine Gruppe zugeordnet.

```
/**
 * -
 * @show
 * @component
 * @group „somegroup“
 */
Object obj;
```

- Ein Attribut primitiven Zahlentyps soll dargestellt werden. Ein Definitionsbereich wird angegeben.

```
int max=10;
/**
 * -
 * @show
 * @range 0..max
 */
int range;
```

- Eine Klasse wird mit einer Gruppe in Verbindung gebracht.

```
/**
 * -
 * @group „somegroup“
 */
class MyClass {...}
```

- Innerhalb einer Schleife werden mehrere Objekte erzeugt und in einem Array abgelegt. Jedes Objekt erhält eine eigene Gruppenzugehörigkeit.

```
for(int i = 0; i < numberOfWorkers; i++) {
  /**@group "Worker"+i*/workers[i] = new Worker(this);
}
```

6 Skalierbarkeit

JAN zeigt unter bestimmten Voraussetzungen eine gute Skalierbarkeit, was die Größe des visualisierten Programmes betrifft. Die Generierung großer Klassen dauert etwas länger als die von kleinen Klassen, was aber kein Problem darstellen dürfte, da die Generierung kein zeitkritischer Vorgang ist. Viel interessanter ist die Ausführung des instrumentierten Programmes.

Natürlich benötigt die Visualisierung jeden Schrittes eine gewisse Zeit, so dass das visualisierte Programm grundsätzlich wesentlich langsamer läuft als das nicht visualisierte. Das ist ist aber gewünscht, da der Beobachter die Visualisierung in einer benutzerfreundlichen Geschwindigkeit verfolgen möchte.

Die kritische Frage ist, ob die Visualisierung einer bestimmten Anzahl von Schritten bei großen Programmen länger dauert als bei kleinen Programmen. Die Ausführungsgeschwindigkeit eines einzelnen Programmschritts, d.h. des Codes zwischen zwei visualisierten Programmelementen hängt grundsätzlich nicht von der Größe des untersuchten Programmes ab, da dieses wie gewohnt in seiner eigenen JVM läuft. Daher ist die Größe des visualisierten Programmes nicht kritisch für die Visualisierungsgeschwindigkeit.

Eine andere Frage bezieht sich auf Programme, die sehr lange laufen. In diesem Zusammenhang ist nicht die Anzahl der tatsächlichen Programmschritte von Bedeutung, sondern nur die Zahl der Ereignisse, die an das Visualisierungssystem gesendet werden. JAN kann auf Rechnern üblicher Größe einige Hundert Schritte visualisieren, ohne dass die Ausführungsgeschwindigkeit eines einzelnen Schrittes ein anwenderfreundliches Maß unterschreitet. Die Visualisierung lang laufender Programme kann mit dem Beispiel 'Loadbalancing' getestet werden.

Von Bedeutung für die Skalierbarkeit ist also weniger die Größe bzw. die Anzahl der Programmschritte des visualisierten Programmes, sondern eher die Anzahl der zu visualisierenden Ereignisse. Bei großen Programmen und auch bei kleineren Programmen mit vielen Programmschritten sollte deshalb schon zum Zwecke der Übersichtlichkeit von den möglichen Annotationen Gebrauch gemacht werden. Es empfiehlt sich, die Grundeinstellung für alle darzustellenden Strukturen vor dem Generieren auf *hide* zu stellen und nur wenige Strukturen im Programmtext mit `/**@show*/` zu kennzeichnen.

Während des Programmlaufs kann durch die Auswahl der darzustellenden Informationen Einfluss auf die Ausführungsgeschwindigkeit genommen werden. Insbesondere die Darstellung von Listen benötigt viel Zeit, was in der internen Verwaltung von Swing begründet ist. Daher empfiehlt es sich, große Listen nicht anzeigen zu lassen. Zu diesem Zweck kann von den Reitern 'history', 'stack' und 'objects' derjenige mit den wenigsten Zeilen ausgewählt werden oder z.B. der Reiter 'stack' ausgewählt und kein Thread aus der Liste der Threads gewählt werden. Die Auswahl eines Threads kann gelöscht werden, indem die Taste 'Steuerung' auf der Tastatur und gleichzeitig die linke Maustaste über dem gewählten Threads gedrückt wird. Analog kann in der Liste 'component-roots' der ausgewählte Baum in der Anzeige gelöscht werden.

6.1Visualisierung von JAN unter Anwendung von JAN

Ein Beispiel für ein großes Programm ist JAN. Daher wird im folgenden die Visualisierung von JAN mit JAN untersucht. Dazu muss der Quelltext der interessierenden Klassen in die Sandbox ('Sandbox/Source') kopiert werden. Desweiteren sind die Graphiken und Property-Dateien dort notwendig. Die Class-Dateien der übrigen Klassen müssen nicht in der Sandbox ('Sandbox/Classes') vorhanden sein, da sie schon im Klassenpfad von JAN stehen. Zum Ausführen wird in der Entwickler-Ansicht die Klasse 'visualization/gui/Main' geladen und in der Ausführung-Ansicht ausgewählt. Je nach Annotation öffnet sich nach einigen Schritten eine zweite JAN-GUI, die hier Client genannt werden soll und in der nun ein beliebiges Programm visualisiert werden kann. Die Vorgänge im Client werden im Server dargestellt. Es ist zu bemerken, dass der Server und der Client sich naturgemäß immer wieder gegenseitig blockieren (erkennbar an gedrückt markierten Buttons). Der Client reagiert erst auf eine Benutzereingabe, wenn im Server der 'Step' Button gedrückt wurde. Andersherum ist eine Betätigung des Clients notwendig, um Schritte im Server ausführen zu können.

Ein sinnvolles Szenario ist die Visualisierung aller Methodenaufrufe von Methoden, die in den Interfaces stehen, die zwischen den einzelnen Schichten von JAN vermitteln (`TraceConsumer`, `TraceHistory`, `TraceEventHandler`). Es werden nun die verhandelnden Objekte und nicht die

Interfaces visualisiert, wodurch der Datenfluss auch über Interfaces hinweg gut zu erkennen ist. Mit den vorhandenen Mitteln kann auf diese Weise der JAN-Server vom `TraceControlProxy` über die zentrale Verwaltung (`SystemHistory`) bis zur GUI (`Main`) visualisiert werden.

Eine Visualisierung der Anteile von JAN im Client (`Trace`, `TraceEventHandlerProxy`) ist nicht ohne weiteres möglich. Dazu sei folgendes Szenario gegeben:

- (1) Server für die Visualisierung von JAN
- (2) Server für die Visualisierung eines Testlings
- (3) Testling auf Client

(1) visualisiert (2) und (2) visualisiert (3).

Die Visualisierung von (2) in (1) ist kein Problem, es kann also der JAN-Server (in einem anderen JAN-Server) visualisiert werden. Dabei lässt der visualisierte JAN-Server (2) einen Testling (3) laufen. Für die Visualisierung der JAN-bezogenen Vorgänge in (3) müssten die entsprechenden Daten (die Aufrufe in der Klasse `Trace` und dem Client-seitigen Proxy `TraceEventHandlerProxy`) aber nicht an (2) sondern an (1) geliefert werden. Der Client (3) müsste also zwischen zwei Servern unterscheiden, nämlich den Server (2) für die Verarbeitung der Daten des Testlings und dem Server (1) für die JAN-Daten. Das Szenario wäre relativ leicht umzusetzen, wenn der Server (1) mehrere Clients unterstützen würde (leichte Erweiterung von `SystemHistory` notwendig). Desweiteren müsste der Client (3) eine Klasse `Trace` für die Testlings-Daten und eine identische Klasse `Trace2` für die JAN-Daten haben, da ansonsten `Trace` sich rekursiv aufrufen würde. (Eventuell sollte der Generator so modifiziert werden, dass die `Trace`-Klasse für die generierten Aufrufe einstellbar ist.) `Trace` und `Trace2` könnten dann denselben Proxy `TraceEventHandlerProxy` benutzen, der die Daten entsprechend von `Trace2` an (1) und von `Trace` an (2) weiterleitet (geringfügige problembezogene Modifikation des Proxys notwendig). Anmerkung: Würden die genannten Sachverhalte nicht berücksichtigt, käme es zu folgendem Phänomen: (1) würde die Server-seitigen Vorgänge der Visualisierung des Testlings und zugleich die Client-seitigen Vorgänge der Visualisierung des JAN-Servers darstellen, was nicht zusammen gehört.

Im Beispiel sind fünf Threads vorhanden (Abbildung 23). '*Trace$2 invoke_program*' ist ein Thread, der für die Ausführung des visualisierten Programmes (d.h. des visualisierten JAN) erzeugt wird. Er macht das visualisierte Programm von JAN's Anteil (d.h. dem beobachtenden JAN) auf dem Client unabhängig. '*Trace$2 invoke_program*' gehört also zur Visualisierung; alle weiteren Threads gehören zum visualisierten Programm. '*Main$67 experiment1*' ist ein Thread, der für die Ausführung des Testlings (auf dem beobachteten JAN) erzeugt wird. '*Main$67 experiment1*' macht diese Ausführung unabhängig von der GUI. Würde dieser Thread nicht verwendet werden, so würde der 'step' Button gedrückt bleiben. '*Trace$2 invoke_program*' und '*Main$67 experiment1*' sind für die Initialisierung zuständig und im abgebildeten Ablaufdiagramm ausgeblendet.

'*EventDispatchThread AWT-EventQueue-0*' bearbeitet die Betätigung des 'step' Buttons. Die beiden Threads '*Thread RMI TCP Connection*' beinhalten den Aufruf des Servers vom Client aus.

Abbildung 23 Einige an der Visualisierung beteiligte Threads.

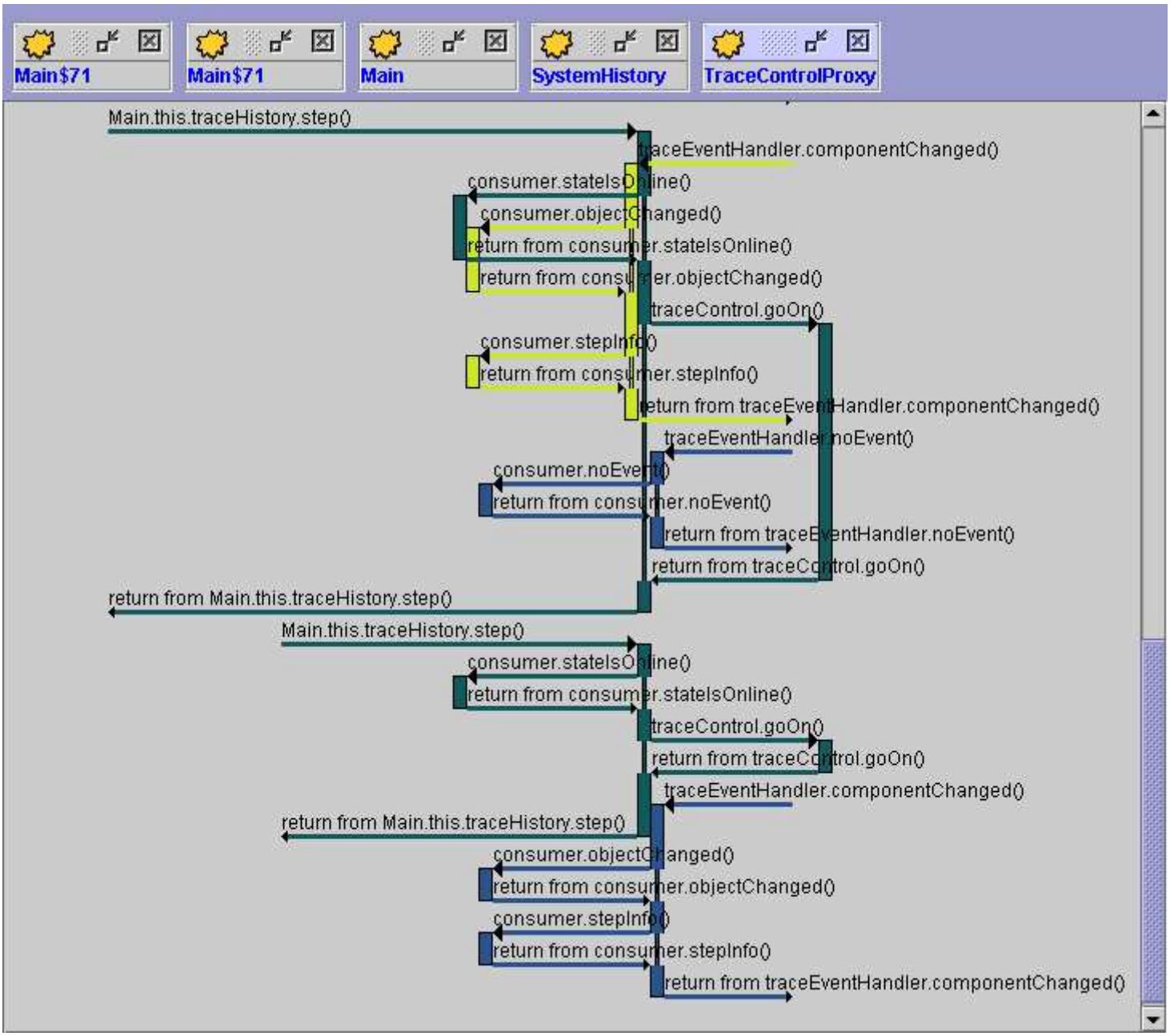


Abbildung 24 Ein Ausschnitt eines Ablaufdiagramms von der Serverseite von JAN.

7 Ausblick

Mit dem Visualisierungssystem ist eine Anwendung entstanden, die eine visuelle Einsicht in die Datenstruktur und den Ablauf eines Java Programmes geben kann. Die geschachtelten Komponentendiagramme vermitteln die Intuition einer Komponentenstruktur aus sich beinhaltenden Objekten. Dabei hat der Benutzer vielfältige Möglichkeiten, die Darstellung interaktiv seinen Wünschen anzupassen.

Der Fokus liegt nicht in der Darstellung statischer Inhalte, sondern in der Vermittlung eines Ablaufs. Das Ablaufdiagramm verschmilzt die Vorgänge in unterschiedlichen Threads, die möglicher Weise um die selben Ressourcen konkurrieren. Durch die Darstellung mit verschiedenen Zeitbalken in unterschiedlichen Farben kann trotz der hohen Komplexität auch schon einfacher Beispiele ein leicht verständlicher Überblick über den Ablauf gewonnen werden.

Es können *die* Objekte ausgewählt werden, deren Interaktionen und Struktur beobachtet werden soll. Diese Auswahl kann durch Aus- und Einblenden einzelner Fenster geschehen. Es kann aber auch schon vorher in der Grundeinstellung des Generators oder in sehr feinkörniger Weise durch Annotationen im Quelltext Einfluss auf die darzustellenden Details genommen werden. Die Annotationen beeinflussen die Lauffähigkeit und Semantik des originalen Programmes nicht, da es sich syntaktisch um Kommentare handelt.

Neben Annotationen für die Darstellung kann das untersuchte Programm mit dem Begriff der Aggregation bzw. Komponente zum Zweck der Visualisierung auch um Semantik erweitert werden. Komponenten werden im Objektdiagramm in einer geschachtelten Struktur dargestellt.

Ein besonderes Merkmal ist die Möglichkeit, in vergangene Zustände zurück zuspringen. Dadurch kann auch der Eindruck eines rückwärts laufenden Programmes geweckt werden.

Das Visualisierungssystem modifiziert den Quelltext des zu inspizierenden Programmes, in dem im Wesentlichen Aufrufe des eigenen Systems eingebaut werden. Diese Aufrufe verändern die Semantik nicht und schleusen die benötigten Informationen nach außen. Das zu untersuchende Programm muss zunächst eine Art Vorübersetzer passieren, der wieder lauffähigen Java Quelltext erzeugt. Dann erfolgt das Übersetzen mit einem handelsüblichen Compiler und schließlich die Visualisierung.

Während der Visualisierung laufen das inspizierte Programm und das Visualisierungssystem parallel in zwei getrennten Prozessen ab, die über Java RMI kommunizieren. Damit kann eine maximale Entkopplung beider Vorgänge sicher gestellt werden.

Literaturverzeichnis

[JSWAT] Blue Marsh Softworks, JSwat - Graphical Java Debugger, www.bluemarsh.com/java/jswat/

[JBUILDER] Borland, Borland JBuilder, www.borland.com/jbuilder/

[TOGETHER] TogetherSoft, TogetherSoft - application modeling, round trip engineering for Java and C++, www.togethersoft.com

[BARAT] Boris Bokowski, Barat, www.boris.bokowski.de

[JAVA] Sun microsystems, Inc, The Source for Java(TM) Technology, java.sun.com

[BOOCH] Grady Booch, James Rumbaugh, Ivar Jacobsen, The unified modeling language user guide, 1999

[OESTEREICH] Bernd Oestereich, Objektorientierte Softwareentwicklung: Analyse und Design, 1997

[CLARKE] David G. Clarke, James Noble, John M. Potter,
Simple Ownership Types for Object Containment, 2001, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, clad@cs.uu.nl, School of Mathematical and Computing Sciences, Victoria University, Wellington, New Zealand, kjx@mcs.vuw.ac.nz, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, potter@cse.unsw.edu.au

[ECOOP] J. Lindskov Knudsen, University of Aarhus, Denmark, ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, 2001

[NOBLE] James Noble, Jan Vitek, John Potter, Flexible Alias Protection, 1998,

[POTTER] David G. Clarke, John M. Potter, James Noble, Ownership Types for Flexible Alias Protection, 1998,

[CLARKE2] David Clarke, Object Ownership and Containment, An Object Calculus with Ownership and Containment, www.cs.uu.nl/~dave/abstracts.html