Interaktive graphische Fehlerbehebung

Ausarbeitung zum Vortrag am 28. Juni 2004 von Jan Sebastian Siwy im Rahmen des Seminars „Visualisierung verteilter Systeme“ an der Freien Universität in Berlin

Fehlerbehebung bei der Softwareentwicklung stellt eine außerordentlich schwierige Aufgabe dar, die sowohl ein Verständnis der Software im Ganzen als auch Detailwissen über die Software erfordert. Durch gängige Werkzeuge wird fast ausschließlich eine sehr quellcodenahe Unterstützung während der Fehlbehebung bereitgestellt, so dass kein hinreichender Überblick über die Software erworben werden kann. In dieser Ausarbeitung werden drei Programme vorgestellt, die während der Ausführung eines Programms den Zustand grafisch darstellen und somit den Vorgang der Fehlerbehebung unterstützen.

1 Einleitung

Grafische Unterstützung wird in der Softwareentwicklung bereits seit Längerem erfolgreich eingesetzt. Jedoch endet der Einsatz entsprechender Werkzeuge meist nach der Entwurfsphase. Diese Ausarbeitung diskutiert die Einsatzmöglichkeiten einer grafischen Darstellung einer Software bei der Fehlerbehebung, denn insbesondere dabei ist ein umfassendes Verständnis der Software notwendig. Ohne ein solches besteht die Gefahr, dass aufgrund von sicher geglaubtem, aber falschem Wissen nur die Symptome, aber nicht die Ursachen eines Problems beseitigt werden. Gängige Werkzeuge bieten hierzu nur eine unzureichendes Unterstützung, da sie sehr quellcodenah arbeiten.

Es werden drei Werkzeuge vorgestellt, welche das entwickelte Programm im Betrieb überwachen und Informationen über den Zustand des Programms ausgeben.

2 Hintergrund

Um grafische Werkzeuge zur Fehlerbehebung in Software zu beurteilten, ist es zuerst notwendig Kriterien für Verfahren zur Darstellung von Software zu definieren. Roman und Cox haben im Jahre 1993 fünf Kriterien vorgeschlagen:

- Das Werkzeug soll einen möglichst großen *Umfang* der entwickelten Software darstellen. Das heißt, es sollte sowohl eine Sicht auf den statischen Teil des Programms (z.B. Quellcode oder Klassendiagramm) als auch auf den dynamischen Teil (z.B. Zustand der Daten) geben.
- Für diese Sichten sollte es verschiedene Stufen der *Abstraktion* geben. Unter einer niedrigen Abstraktionsstufe ist z.B. der Quellcode, unter einer hohen z.B. ein Klassendiagramm zu verstehen.
- Das Werkzeug sollte eine *Methode zur Spezifikation* der Darstellung beinhalten. Das heißt, dass der Benutzer mit möglichst geringem Aufwand einstellen können sollte, welche Informationen ihm durch das Werkzeug bereitgestellt werden.
- Die *Benutzerschrittstelle* sollte einfach und intuitiv zu bedienen sind. Das heißt, dass z.B. sichtbare Elemente über die Maus ausgewählt werden können und dies nicht erst durch eine Kommandozeile notwendig ist.
- Die *Darstellung* muss die Komplexität der Information in geeigneter Weise verringern. Das heißt, es muss verhindert werden, dass der Benutzer des Werkzeuges durch die Menge oder den Detailgrad der Informationen überfordert ist.

Gängige Werkzeuge wie Visual C++ aus dem Jahre 1998 oder ProDev Workshop aus dem Jahre 2002 erfüllen diese Kriterien kaum. Die Abstraktion ist sehr gering, denn der Benutzer hat nur die Möglichkeit auf der Quellcodeebene zu arbeiten; die Methoden zur Spezifikation der Darstellung sind auch sehr eingeschränkt; die Benutzerschnitte bietet aufgrund der geringen Abstraktion ebenfalls kaum Möglichkeiten, die über die Bearbeitung des Quellcodes hinausgehen.

Einige Forschungsprojekte bieten eine höhere Abstraktion. Sie stellen z.B. Zustandsänderungen, den Fluss von Nachrichten oder die Erzeugung von Exemplaren grafisch dar (z.B. Baecker im Jahre 1997 und Jarding sowie Stasko im Jahre 1994 und 1996). Andere Forschungsprojekte befassen sich mit verschiedenen Möglichkeiten ausgewählte Aspekte eines Programms darzustellen, vernachlässigen aber die anderen Kriterien.

UML erscheint aufgrund der weiten Verbreitung und der bekannten Syntax und Semantik als geeignete Grundlage für ein grafisches Fehlerbehebungswerkzeug. Der Umfang der Darstellungsmöglichten ist bei UML sehr groß. Es können z.B. Klassendiagramme für eine statische Sicht, Objekt- oder Sequenzdiagramme für eine dynamische Sicht auf das Programm benutzt werden; es existieren zudem verschiedene Abstraktionsstufen, so dass auch die Komplexität der

Darstellung nach den Bedürfnissen angepasst werden kann. Die verbleibenden Kriterien lassen sich auf UML nicht anwenden, weil sowohl für eine Methode zur Spezifikation sowie für eine Benutzerschnittstelle ein Werkzeug und keine Modellierung betrachtet werden muss.

3 Anwendungen

Das erste Werkzeug arbeitet mit UML-Objektdiagrammen, die dynamisch erstellt und angepasst werden, wobei jedoch der Platzbedarf für diesen Diagrammtyp verbessert worden ist.

Das zweite Werkzeug, Java Animation (JAN), arbeitet mit einer eigenen Notation für Objektdiagramme, die verbesserte Darstellung der Aggregation unterstützt. Außerdem wird neben dem Objektdiagramm auch das Sequenzdiagramm unterstützt.

Das dritte Programm, Data Display Debugger (DDD) ist lediglich eine grafische Benutzerschnittstelle für Fehlerbehebungswerkzeuge und nutzt ebenfalls eine eigene Notation.

Wie bei allen Überwachungsprogrammen darf sich auch durch diese Werkzeuge das Verhalten der Software durch die Überwachung nicht verändern, und der Einfluss auf die Leistung der Software soll möglichst gering gehalten werden.

3.1 Anwendung mit UML

Aus der obigen Betrachtung ist ersichtlich, dass UML drei der fünf Kriterien in geeigneter Weise erfüllt. Weiterhin ist UML in der Softwareentwicklung in der Entwurfsphase weit verbreitet. So werden für den Entwurf von Programmen z.B. so genannte CASE-Werkzeuge (computer-aided software engineering) benutzt. Trotz der weiten Verbreitung von UML in der Entwurfsphase werden die Diagramme bei der Fehlerbehebung meistens nicht mehr verwendet. Der Grund dafür ist, dass in der Implementierungsphase häufig die Verbindung zum Entwurf verloren geht. Bei Änderungen des Entwurfs in dieser Phase wäre eine manuelle Pflege der Diagramme notwendig, worauf aufgrund des zusätzlichen Aufwands meist verzichtet wird. Daher ist das Ziel ein Werkzeug vorzustellen, dass zur Laufzeit automatisch ein UML-Objekt–diagramm erstellt.

Grundlage ist das Programm ArgoUML aus dem Jahre 2002, ein Open-Source-Programm zur Erstellung und Bearbeitung von UML-Diagrammen. Erweitert wird es mit Hilfe von Java Platform Debug Architecture (JPDA), einer Schnittstellen der virtuellen Maschine von Java zur Realisierung von Werkzeugen zur Fehlerbehebung.

Durch JPDA werden Informationen über die einzelnen Exemplare bestimmt. Wenn ein Exemplar erzeugt wird, eine Veränderung an einem Exemplar stattfindet oder wenn eine Verbindung zwischen zwei Exemplaren entsteht oder gelöst wird, so wird diese Information auf ein Objektdiagramm übertragen. Der Platzbedarf durch folgendes Verfahren namens *focus+context* verringert.

3.1.1Detailgrad

Um größere Objektdiagramme effizienter darzustellen, wurde UML angepasst. Dabei wurde die Syntax und Semantik von UML weitestgehend beibehalten, jedoch wurde der Platzbedarf verringert.

Dafür werden die Exemplare nicht immer mit allen Informationen angezeigt, sondern es wird für jedes Exemplar ein Wert berechnet, der Auskunft über die Bedeutung des Exemplars gibt (degree of interest, DOI):

$$DOI \approx \log_2(freq) + (max\_visible\_doi - dist)$$

In dieser Formel ist *freq* die Häufigkeit, mit der auf ein Exemplar zugegriffen worden ist, *max_visible_doi* ist der größtmögliche DOI, bei dem ein Exemplar noch angezeigt werden soll, und *dist* ist die Entfernung eines Exemplars von dem aktuell ausgewählten Exemplar, wobei die Entfernung die minimale Anzahl der Verbindungen ist, die durchlaufen werden müssen, um das ausgewählte Exemplar zu erreichen.

Der Benutzer hat die Möglichkeit ein Exemplar in den Vordergrund zu holen. Damit bestimmt er, welches Exemplar auf jeden Fall mit allen Details angezeigt wird und als Grundlage für die Berechnung der DOIs der übrigen Exemplare dient. Damit entsteht im Grunde ein Fischaugen-Effekt um das Exemplar im Vordergrund.

Aufgrund des DOI wird der Detailgrad (level of detail, LOD) für jedes Exemplar bestimmt.

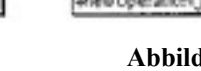
- LOD 0: Es werden alle in einem UML-Objektdiagramm vorgesehenen Informationen über ein Exemplar angeblendet (Name, Liste der Attribute mit ihren Datentypen und Methoden mit ihren Signaturen).

Exemplar 0
Attribut: int
Methode(): void

- LOD 1: Wie LOD 0, jedoch ohne die Angabe der Datentypen der Attribute sowie der Rückgabedatentypen der Methoden.

Exemplar 1
Attribut
Methode()

- LOD 2: Es wird nur der Name des Exemplars mit einer verringerten Schriftgröße angezeigt.



- LOD 3: Es wird auf alle Textinformationen verzichtet, d.h. ein Quadrat symbolisiert nur die Existenz eines Exemplars.

- LOD 4: Wie LOD 3, jedoch mit einem kleineren Quadrat.

- LOD 5: Das Exemplar wird vollständig ausgeblendet.

Unabhängig vom errechneten DOI erhält ein Exemplar den kurzzeitig LOD 1, wenn es einen LOD 2 bis 5 hat, wenn aus das Exemplar zugegriffen wird.

Die Abbildungen 1a und 1b zeigen, wie sich das Objektdiagramm verändert, wenn die Exemplare gemäß ihrem DOI in entsprechender Detailsausprägung angezeigt werden. In dem Beispiel wird das Exemplar mit dem Namen *Class0* (sic!) am häufigsten benutzt und hat damit den höchsten DOI und LOD 0. Das Exemplar *Class1* ist über eine Verbindung zu *Class0* erreichbar und besitzt damit einen geringeren DOI und somit LOD 1. Dieses Prinzip setzt sich nun bis *Class4* fort.

Eine weitere Möglichkeit, den Platzbedarf eines UML-Objektdiagramms zu reduzieren, bietet sich bei der Aggregation an. Wenn bestimmte Exemplare Bestandteile anderer Exemplare sind, dann man die Bestandteile ausblenden und nur die Hauptkomponente zeigen. Die Existenz von Bestandteilen wird durch die Aggregationspfeile der UML-Notation nur angedeutet. Es würde z.B. ausreichen, wenn man einer Hauptkomponente Auto mit den Bestandteilen Reifen lediglich das Auto darstellt.

Bei dieser Technik wird die Annahme gemacht, dass die Bestandteile meist eine geringere Bedeutung haben als die Hauptkomponente, so dass diese trotz einem anderen errechneten DOI ausgeblendet werden

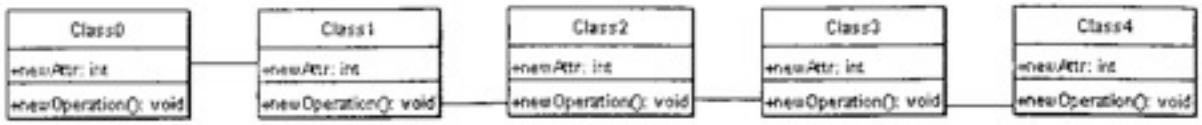


Abbildung 1a: Klassisches UML-Objektdiagramm

Abbildung 1b: UML-Objektdiagramm nach der Verringerung des Detailgrads

können.

Die Abbildungen 2a und 2b zeigen wie der Platzbedarf durch die Erkennung von Aggregationen reduziert werden kann. In diesen Abbildungen wird auch eine Vererbung verwendet, die in einem Objektdiagramm keinen Sinn ergibt. In der Quelle werden die Diagramme zwar als Klassendiagramme bezeichnet, jedoch ergibt dies im Zusammenhang mit der Arbeitsweise der Anwendung ebenfalls keinen Sinn.

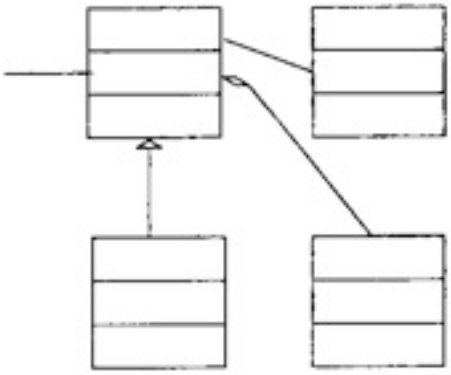


Abbildung 2a: Klassisches UML

Abbildung 2b: Erkennung von Aggregation

Die Quelle lässt leider offen, wie eine Aggregation ohne eine Modifikation des Quellcodes von einer normalen Verbindung unterschieden werden kann.

3.1.2Anordnung

Abgesehen von der Reduktion des Detailgrads kann der Platzbedarf auch durch geschickte Anordnung der Exemplare reduziert werden. Bei der automatischen

Positionierung der Exemplare hat die Hierarchie die oberste Priorität. Was genau eine Hierarchieebene ist, kann aus der Quelle nicht genau bestimmt werden. Anhand der Beispiele scheint dies die Vererbungstiefe zu sein, wobei dies bei Objektdiagrammen natürlich auch hier keinen Sinn ergibt.

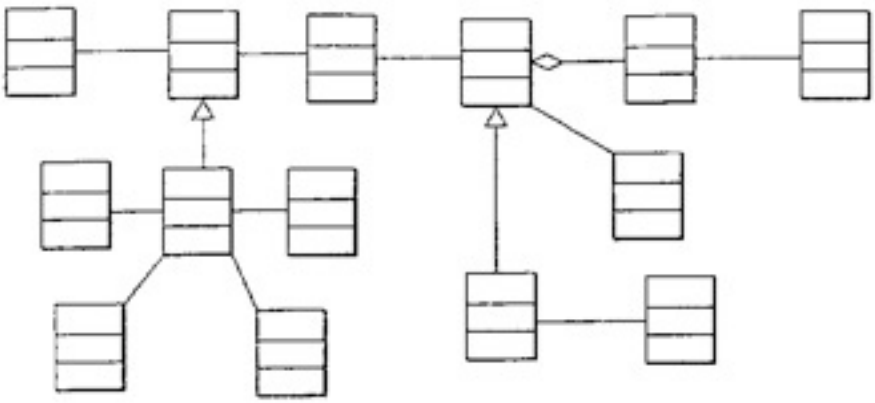


Abbildung 3a: Klassisches UML

Wenn erkannt wird, dass eine Anpassung des Objektdiagramms notwendig ist, so wird der alte Zustand durch eine Animation in den neuen Zustand überführt, so dass es dem Benutzer möglich ist zu verfolgen, wie die Änderung stattgefunden hat.

Die Abbildung 3a und 3b zeigen noch mal, die Auswirkung auf UML-Diagramm durch die Verringerung des Detailgrads.

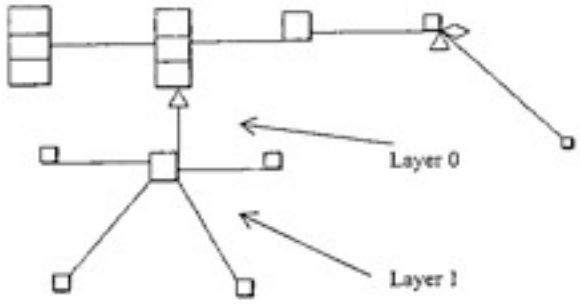


Abbildung 3b: Verringerung des Detailgrads

Abbildung 3c zeigt, wie die hierarchische Anordnung der Exemplare aussehen würde.

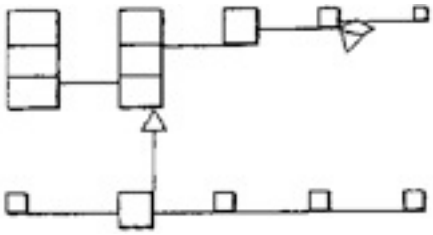


Abbildung 3c: Hierarchische Anordnung

Im letzten Schritt wird die Anordnung der Exemplare optimiert. Das heißt, dass Exemplare mit einem geringen DOI und die nur zu einem anderen Exemplar in Verbindung stehen, kreisförmig um das Exemplar angeordnet werden. Dies ist in Abbildung 3d zu sehen.

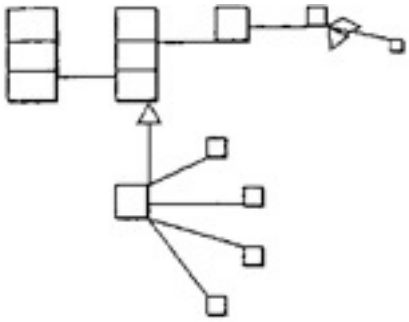


Abbildung 3d: Optimierte Anordnung

3.2 JAN – Java Animation

JAN verfolgt das gleiche Ziel, die Exemplare eines Programms zur Laufzeit anzuzeigen. Neben einem Objektdiagramm wird zusätzlich noch ein Sequenzdiagramm angeboten. JAN benutzt jedoch nicht die klassische UML-Notation. So werden Aggregationen nicht durch Verbindungen zwischen der Hauptkomponente und derer Bestandteile dargestellt, sondern die Bestandteile werden in der Hauptkomponente dargestellt. Dies ist in der Abbildung 4 bei der Sammlung *buecher* der Fall: Die einzelnen Exemplare des Typs *Buch* erscheinen hier im gleichen Fenster wie die Sammlung *buecher*.

Zusätzlich ist es möglich über Anmerkungen in Kommentaren die Animation zu beeinflussen. Man kann z.B. explizit festlegen, welche Verbindungen als Aggregationen aufzufassen sind. Die Anmerkungen werden durch eine Vorübersetzung in eine Kopie des Java-Quellcodes eingearbeitet.

3.3 DDD – Data Display Debugger

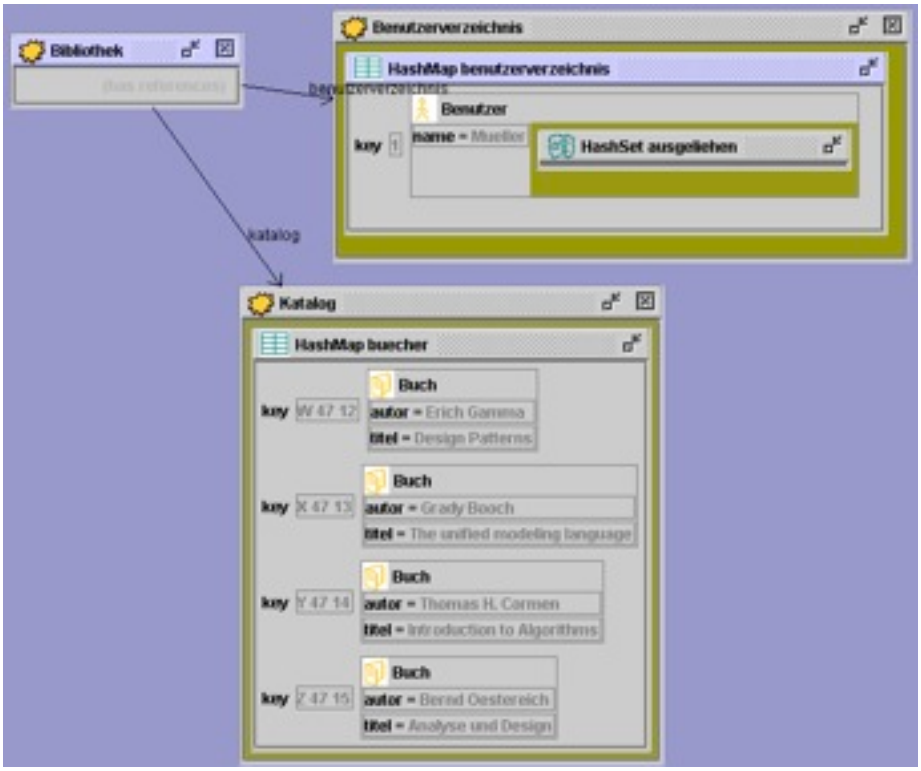


Abbildung 4: JAN mit dem Beispiel *Bibliothek*

DDD ist kein eigenständiges Werkzeug, sondern lediglich eine grafische Benutzerschnittstelle für Fehlerbehebungswerkzeuge. Unterstützt werden z.B. Werkzeuge wie GNU debugger (GDB), DBX, Ladebug und XDB. Allein mit GBD besteht damit eine Unterstützung für sehr viele verschiedene Programmiersprachen: C, C++, Java und viele Andere.

Es werden verschiedene Darstellungsformen unterstützt. Abbildung 5a zeigt beispielsweise, wie Verweise dargestellt werden. Auch wird keine UML-Notation benutzt.

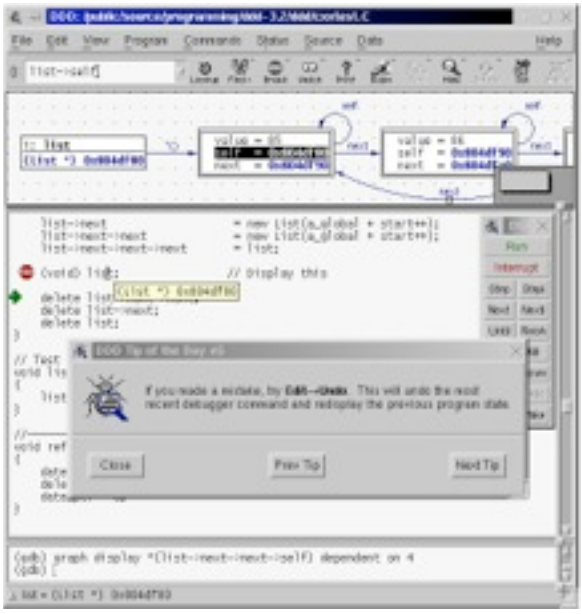


Abbildung 5a: Darstellung von Verweisen

In Abbildung 5b wird der Inhalt von numerischen Feldern dargestellt. Dabei besteht nicht nur die Möglichkeit eindimensionale Felder als Balkendiagramm darzustellen, sondern es gibt auch Unterstützung für zweidimensionale Felder als Flächen.

Abbildung 6: Objektdiagramm mit großen Exemplaren im Vordergrund

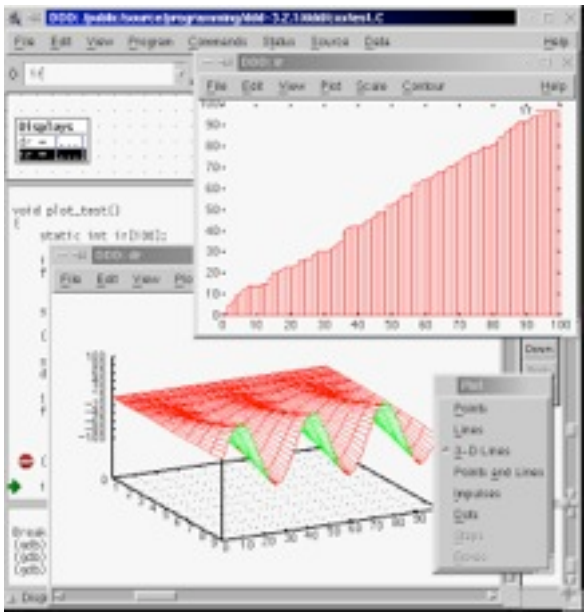


Abbildung 5b: Darstellung von Feldern

4

Diskussion

4.1 Nützlichkeit von *focus+context*

Bei recht einfachen Klassen kann man feststellen, dass bei normaler UML-Notation ungefähr 24 Exemplare auf dem Bildschirm Platz finden. Nach Anwendung der Algorithmen zur Verbesserung des Platzbedarfs (focus+context) können ungefähr 75 Exemplare angezeigt werden.

Der Zugriff auf die einzelnen Exemplare ist bei der modifizierten Variante von UML erleichtert, weil die Notwendigkeit für das Rollen verringert werden kann. Obwohl die Exemplare teilweise nicht in allen Details angezeigt werden, ist es für den Benutzer einfacher ein Exemplar ausfindig zu machen als durch das Rollen, weil die Verbindungen zwischen den Exemplaren ersichtlich sind. Weiterhin fällt die Auswahl eines Exemplars einfacher, weil nur ein Arbeitsvorgang – nämlich das Anklicken – notwendig ist, während beim klassischen UML zuerst das Rollen durchgeführt werden muss.

Der Zugriff bleibt aber dennoch problematisch, wenn Exemplare mit vielen Attributen und vielen Methoden in den Vordergrund geholt werden. Da Exemplare im Vorgrund mit allen Details angezeigt werden, kann ein solches Exemplar alleine durchaus den ganzen Bildschirm einnehmen, wie in Abbildung 6 zu sehen.

Überschneidungen, die durch die hierarchische Anordnung der Exemplare entstehen können, werden durch das focus+context-Verfahren nicht beseitigt. Durch leidet die Übersichtlichkeit der Objektdiagramme. Auch dies ist in Abbildung 6 zu sehen.

Bei der diesem Verfahren ist auch nicht geklärt, welcher Faden aktuell in einem Exemplar aktiv ist. Verwendete Exemplare werden zwar hervorgehoben und kurzzeitig mit mindestens LOD 1 angezeigt, jedoch wird dadurch nicht ersichtlich, welcher Faden das

Exemplar verwendet. Aus der Quelle wird nicht einmal ersichtlich, ob es überhaupt möglich ist, sich alle Fäden anzeigen zu lassen.

4.2 Vergleich der Werkzeuge

Alle drei Werkzeuge verfolgen sehr ähnliche Ziele. Während das UML-Werkzeug und JAN ausschließlich für Java-Programme konzipiert sind, kann DDD für jede Programmiersprache eingesetzt werden, sofern ein entsprechendes Fehlerbehebungswerkzeug unterstützt wird.

Bei dem UML-Werkzeug wurde der großen Wert darauf gelegt, dass die UML-Syntax von Objektdiagrammen nahezu vollständig beibehalten wurden ist. Insbesondere bei Aggregationen erscheint dies nicht sinnvoll, weil die Notation weniger intuitiv ist. Die von JAN verwendete Notation ist hierbei vorteilhafter.

5 Kommentar zur Quelle

In der Quelle zu dieser Ausarbeitung behaupten die Autoren zu Beginn von Abschnitt 3, dass deren Abhandlung von der grafischen Darstellung von objektorientierten Anwendungen in einem verteilten System handelt. Dies ist eine Irreführung, da das vorgestellte Programm keine Unterstützung für verteilte Systeme bereitstellt. Es existiert zwar ein Abschnitt über die Analyse verteilter Systeme (Abschnitt 8), jedoch ist die Kernaussage nur, dass die Autoren sich in früheren Arbeiten mit verteilten Systemen beschäftigt haben und erst für die Zukunft beabsichtigen diese Arbeiten auf eine nicht näher spezifizierte Art und Weise mit dem vorgestellten Programm zusammenzuführen.

Weiterhin benutzen die Autoren in ihrer Abhandlung die Begriffe Objekt- und Klassendiagramm scheinbar willkürlich. Obwohl im dritten Absatz von Abschnitt 3 der Unterschied zwischen beiden Diagrammtypen erläutert wird, halten sich die Autoren in spätern Absätzen nicht mehr an diese Unterscheidung. Zu Beginn von Abschnitt 3 wird explizit gesagt, dass in der vorgestellten Anwendung Objektdiagramme benutzt werden, doch bereits in Unterabschnitt 3.1 wird bereits von Klassen gesprochen. Auch die vorgestellten Grafiken, die angeblich Objektdiagramme zeigen, benutzen Symbole, die Klassendiagramme vorbehalten sind (z.B. existiert bei Grafik 7 eine Vererbung).

Leider ist das in der Abhandlung vorgestellte Programm nicht verfügbar. Da es auch keine empirische Untersuchung über die Nützlichkeit der vorgestellten *focus+context*-Methode gibt, hat der Leser keine Möglichkeit die präsentierte Schlussfolgerung zu überprüfen.

6 Einordnung in das Seminar

Das Thema der Abhandlung passt nur bedingt zu den anderen Vorträgen. Einerseits handelt es sich bei dem vorgestellter Software um ein Werkzeug zur grafischen Darstellung, andererseits hat dieses Werkzeug – auch wenn in der Abhandlung das Gegenteil behauptet wird – nichts mit verteilten Systemen zu tun.

Mit dem Sequenzdiagramm stellt JAN eine Unterstützung für Nebenläufigkeit bereit, jedoch existiert auch hier keine Darstellung von Nachrichten basierenden Systemen, die für verteilte Systeme häufig verwendet wird.

7 Quellen

JACOBS, TIMOTHY UND MUSIAL, BENJAMIN,
„Interactive Visual Debugging with UML“,
Air Force Institute of Technology, *ACM Synposium of Software Visualization, San Diago, CA,* 2003, Seite 115-122

LÖHR, KLAUS-PETER UND VRATISLAVSKY, ANDRÉ,
Object-Oriented Program Animation Using JAN,
Freie Universität Berlin, 2003,
http://page.mi.fu-berlin.de/~vratisla

Data Display Debugger,
http://www.gnu.org/software/ddd