Fachhochschule Gelsenkirchen
Fachbereich Informatik
Medieninformatik

KARTINA

EIN FRAMEWORK ZUR ALGORITHMENVISUALISIERUNG

BACHELORTHESIS

Verfasser:

Malte Schwerhoff

200425206

Betreuer:

Prof. Dr. Marcel Luis

Datum und Unterschrift des Betreuers

Hiermit versichere ich, die Arbeit selbständig angefertigt und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt zu haben.

Datum und Unterschrift des Verfassers

Malte Schwerhoff: *Kartina – ein Framework zur Algorithmenvisualisierung*
Bachelorthesis, © Februar 2008

ZUSAMMENFASSUNG

Diese Thesis beschreibt die Entwicklung von Kartina, einem Framework zur Visualisierung von Algorithmen. Unter Algorithmenvisualisierung versteht man die schrittweise grafische Darstellung des Ablaufs eines Algorithmus, genutzt wird dies insbesondere im Rahmen der Lehre.

Nach einer kurzen Einführung in das Thema der Algorithmenvisualisierung werden die der Entwicklung zugrunde liegenden Motive aufgezeigt. Daran anschließend wird die Architektur von Kartina beschrieben, die auf dem Java Debugger Framework aufsetzt, sowie die für den Visualisierungsprozess benötigten Klassen. Dabei werden zuerst Ideen bzw. Anforderungen beschrieben, die sich ggf. in Schnittstellen abbilden lassen, bevor auf die eigentliche Implementierung eingegangen wird.

Abschließend werden zwei Beispielvisualisierungen (Selection- und Quicksort) vorgestellt, die als Machbarkeitsstudie einen Eindruck davon vermitteln, welche Möglichkeiten Kartina dem Visualisierungsentwickler bietet.

ABSTRACT

This thesis describes the development of Kartina, a framework for algorithm visualisation, that is to say the graphical representation of a stepwise executed algorithm. This technique is especially used for educational purposes.

After a short introduction into the topic of algorithm visualisation, the motives for the development of Kartina will be pointed out, followed by the introduction of the systems architecture, which bases on the Java Debugger Interface. Afterwards the important classes of the visualisation process will be described, going from ideas specified in interfaces to actual implementations.

Finally two visualisations (Selection- and Quicksort) are presented as a proof of concept to give an idea of the possibilities that Kartina provides to developers of algorithm visualisations.

KONVENTIONEN

In dieser Thesis werden folgende Konventionen verwendet:

- Quelltextbezüge, z.B. Methodennamen und Ausdrücke sowie Verzeichnisangaben werden in `nichtproportionaler` Schrift gesetzt.
- Klassennamen werden nur bei ihrer erstmaligen Erwähnung in `nichtproportionaler` Schrift gesetzt, da übermäßiger Gebrauch nichtproportionaler Schrift den Lesefluss stört. Ausnahmen werden nur gemacht, falls es sonst eventuell unklar ist, ob es sich um einen Klassennamen oder um etwas anderes handelt.
- Die im Rahmen dieser Thesis erstellten Klassen befinden sich alle im Paket `de.oakgrove.kartina` sowie in dessen Unterpaketen. Klassennamen in dieser Thesis, die mit dem Paketpräfix `kartina` anfangen, beziehen sich daher immer auf `de.oakgrove.kartina`, d.h. mit `kartina.debugger.DelayedVMResumer` ist die Klasse `de.oakgrove.kartina.debugger.DelayedVMResumer` gemeint.
- Methodennamen werden immer mit Klammern angegeben, zum Beispiel `sleep()`. Die Angabe der Parameter entfällt, es sei denn, es wird näher auf sie eingegangen oder die Signatur ist zur eindeutigen Identifizierung einer bestimmten Methode zwingend nötig.
- Begriffe, die entweder in dieser Thesis geprägt werden oder die im Zusammenhang mit Algorithmenvisualisierung eine besondere Bedeutung haben, werden in *kursiver* Schrift gesetzt.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

LISTINGSVERZEICHNIS

TABELLENVERZEICHNIS

Teil I

EINLEITUNG

1 ALGORITHMENVISUALISIERUNG

1.1 Einsatzgebiete

Die Visualisierung von Abläufen in der Informatik unter Fokussierung auf unterschiedliche Aspekte ist ein Standardvorgehen bei der Entwicklung von Software, bei der Erforschung von Algorithmen und bei der didaktischen Aufbereitung im Rahmen der Lehre.

Zu den bekanntesten Vertretern zählen *UML* für die Visualisierung (hier konkret die Modellierung) verschiedener Aspekte in der objektorientierten Softwareentwicklung, *Petrinetzen* (bzw. deren grafische Repräsentation) für die Modellierung paralleler Abläufe und *Programmablaufpläne*[1] für die Darstellung von linearen Programmflüssen.

Unter *Algorithmenvisualisierung* lässt sich im Allgemeinen jede grafische Repräsentation verschiedener Aspekte eines Algorithmus auffassen, z.B. die statische Struktur, der Ressourcenverbrauch oder Variablenbelegungstabellen.

Für die didaktische Aufbereitung eines Algorithmus zu Lehrzwecken sind jedoch besonders folgende Punkte von Interesse:

Ablauf Wie geht der Algorithmus vor, welche Entscheidungen werden zu einem Zeitpunkt getroffen und welche Aktionen werden daraufhin ausgeführt?

Ideen Welche Ideen stecken hinter einem Algorithmus? Für das Verständnis eines Algorithmus ist die ihm zugrunde liegende Idee von zentraler Bedeutung, da sie der Grund dafür ist, warum bestimmte Entscheidungen getroffen werden.

Interaktionen Welche Operationen führt der Algorithmus auf seinen Datenstrukturen aus und welche Auswirkungen haben sie auf die Datenstruktur im lokalen und im globalen Kontext?

Algorithmenvisualisierung ist beileibe kein neues Thema. Nach Fleischer und Kučera [10] reicht ihre Geschichte von Goldsteins und von Neumanns Repräsentation von Programmen als Programmablaufplänen über das erste,1981 von Baecker zu Lehrzwecken erstellte Video „Sorting out Sorting“bis zu aktuellen Werkzeugen zur Algorithmenvisualisierung wie z.B. Jeliot.

1.2 Statische und dynamische Visualisierungen

statische Visualisierung: grafisch sehr flexibel aber szenariogebunden

Das genannte Video fällt dabei in die Kategorie der *statischen* Visualisierungen. Statisch meint hierbei nicht die Visualisierung als solches, denn deren Flexibilität ist gerade bei statischen Visualisierungen besonders hoch, da bei ihrer Erstellung auf Techniken aus der Videoproduktion zurückgegriffen werden kann. Statisch bezieht sich stattdessen auf den Umstand, dass die Visualisierung für einen bestimmten Ablauf des Algorithmus, d.h. für einen

1 DIN 66001

bestimmten Datensatz, erstellt wurde. Um den Ablauf des Algorithmus auf einem anderen Datensatz zu visualisieren, müssen weitere, separate Visualisierungen erstellt werden.

dynamische Visualisierung: grafisch eingeschränkt, aber szenariounabhängig

Aktuelle Werkzeuge zur Algorithmenvisualisierung fallen mehrheitlich in die Kategorie der *dynamischen* Visualisierungen, d.h. sie ermöglichen das Operieren auf unterschiedlichen Datensätzen, ohne dass dafür eine neue Visualisierung erstellt werden muss. Mitunter wird dies jedoch nur dadurch erreicht, dass das Werkzeug die Visualisierung jedes mal neu generiert, so dass solche Werkzeuge eigentlich in beide Kategorien eingeordnet werden können.

Im Rahmen der dynamischen Visualisierung wird versucht, einen hohen Grad an Automatisierung zu erreichen. Eine *automatische* Visualisierung nimmt dem Ersteller der Visualisierung viel Arbeit ab, indem sie bestimmte Aspekte des Algorithmus eigenständig darstellt, z.B. die Darstellung eines Arrays als Reihe unterschiedlich hoher Balken. Leider sinkt mit steigendem Automatisierungsgrad die Flexibilität einer Visualisierung, so dass der Grad der Automatisierung in der Praxis beschränkt ist.

Als Spezialisierung des Begriffs der automatischen Visualisierung kann das *Paradigma der Selbstvisualisierung* aufgefasst werden (siehe Kap. 3.1). Hierbei wird die Visualisierung eines Objektes durch seinen Datentyp bestimmt, z.B. wird ein Integer immer als Zahl in einem Kästchen dargestellt, eine Warteschlange als Aneinanderreihung von stilisierten Personen. Dieser Ansatz reduziert die Flexibilität bei der Visualisierung zwar auf ein Minimum, dafür lassen sich mit sehr geringem Aufwand Visualisierungen erstellen.

1.3 Deklarative und imperative Visualisierungen

Bei der Visualisierung des Ablaufs[2] eines Algorithmus kann generell zwischen *deklarativem* und *imperativem Vorgehen* unterschieden werden [10].

Beim deklarativen oder *datengetriebenen* (*data-driven*) Vorgehen wird jedem Zustand des ablaufenden Programms ein Bild zugeordnet und jeder Zustandsübergang führt zu einer automatischen Änderung dieser grafischen Repräsentation. Theoretisch hat ein Beobachter beim deklarativen Vorgehen immer den kompletten aktuellen Zustand vor sich, ohne dass er explizit mitgeteilt bekommt, welche Änderungen sich gerade ergeben haben. Diese lassen sich jedoch aus der Differenz zwischen dem vorherigen und dem aktuellen Bild rekonstruieren.

imperative Visualisierung: mehr Aufwand, mehr Möglichkeiten

Beim imperativen oder *ereignisgesteuerten* (*event-driven*) Vorgehen führt die Ausführung bestimmter Operationen im Ablauf des Algorithmus zur Generierung von *Ereignissen* (*Events*), aufgrund derer eine Steuerungseinheit die Aktualisierung der grafischen Repräsentation vornimmt. Der Steuerungseinheit wird dabei nur die Änderung mitgeteilt, den Gesamtzustand muss sie selbst ermitteln.

Eine passable Analogie ergeben zwei Briefschachspieler, die sich entweder abwechselnd ein Foto ihres Spielbretts schicken (deklarativ) oder sich den jeweils getätigten Zug („Springer von b8 nach c6“) zukommen lassen (imperativ).

2 Die Unterscheidung zwischen deklarativem und imperativem Vorgehen bietet sich eigentlich nur im Rahmen der dynamischen Visualisierung an. Bei einer statischen Visualisierung kann eventuell von deklarativem Vorgehen gesprochen werden, wenn man die einzelnen Szenen der Visualisierung als abgebildete Zustände auffasst.

Nach Demetrescu u. a. [7] resultiert das imperative Vorgehen zwar in mehr Aufwand für den *Visualisierer* (siehe Kap. 1.6.3) und es erfordert genauere Kenntnisse über die konkrete Implementierung des zu visualisierenden Algorithmus, dafür ist es jedoch flexibler und bietet dem Visualisierer mehr Freiraum in der grafischen Umsetzung.

1.4 Granularität

Für beide Vorgehensweisen gilt, dass sich die einzelnen Operationen eines Algorithmus - bei einer konkreten Implementierung also die Anweisungen der verwendeten Programmiersprache – auf unterschiedlichem Niveau betrachtet werden können. [7] nennt dies die *Granularität der Animation* und gibt als Beispiel einen Tauschvorgang während eines typischen vergleichsbasierten Sortierverfahrens an:

Granularität der Visualisierung: bei Ideen hoch, bei Implementierungen niedrig

- eine eng an der konkreten Implementierung orientierte Visualisierung stellt den Tauschvorgang in drei Schritten dar:
  1. Sichern des Wertes an Position A in der Liste in einer temporären Variable
  2. Position A erhält den Wert von Position B
  3. Position B erhält den Wert aus temporären Variable

  Nach [7] hat diese Darstellung des Tauschvorgangs eine hohe Granularität, da der Tauschvorgang schrittweise dargestellt wird.
- eine abstraktere Darstellung, die den Tauschvorgang als ganzes, unabhängig von seiner konkreten Implementierung visualisiert, kann die Vertauschung der Werte an Position A und B gleichzeitig ablaufen lassen und hätte somit eine niedrigere Granularität als die erste Darstellung
- Falls der Tauschvorgang Teil einer größeren Operation ist und als eigener Vorgang nur von geringem Interesse ist, kann er in der Visualisierung auch komplett fehlen oder derart visualisiert werden, dass nicht das WIE – der Tauschvorgang – sondern das WARUM – die Idee dahinter – zum Ausdruck gebracht wird.

1.5 Beobachten der Programmausführung

Unabhängig davon, ob die deklarative oder die imperative Vorgehensweise gewählt wird, muss es in jedem Fall eine Instanz geben, die den Ablauf des Programms beobachtet und die entweder die Abbildung vom Zustands- in den Bildraum vornimmt (deklarativ) oder Ereignisse generiert, die eine Rekonstruktion des Zustands erlauben.

Hierfür existieren im Allgemeinen zwei Ansätze: *Instrumentierung* sowie das Andocken an die Laufzeitumgebung (nach Kreft und Langer [17]).

1.5.1 *Instrumentierung*

Instrumentierung bedeutet in diesem Zusammenhang das Anreichern des auszuführenden Codes um Anweisungen, die letztendlich den Ablauf des

Codes beobachtbar machen. Grundsätzlich können dabei alle im Zuge einer Programmausführung vorkommenden Arten von Code instrumentiert werden, d.h. der Quelltext des Programms, der finale Maschinencode sowie jeglicher Zwischen- oder Bytecode.

Quelltext-instrumentierung verschmutzt selbigen, ist aber immer möglich

Die Instrumentierung des bereits erwähnten Tauschvorgangs im Quelltext kann z.B. wie folgt aussehen:

```
1 // An dieser Stelle kann ein Ereignis generiert werden,
2 // das Verweise auf die am Tausch beteiligten Variablen
3 // enthält, z.B.
4 //
5 // generiereTauschereignis(array, i, j)
6
7 int tmp = array[i]
8 array[i] = array[j]
9 array[j] = tmp}
```

Listing 1: Imperative Instrumentierung eines Tauschvorgangs

Die Positionierung der Visualisierungsanweisungen relativ zum Tauschprozess (davor oder dahinter) ist frei wählbar, allerdings ergeben sich aus den unterschiedlichen Positionierungen unterschiedliche Anforderungen an die Daten, die den Visualisierungsanweisungen übergeben werden.

Der obige Quelltextauszug ist ein Beispiel für ein imperatives Vorgehen bei der Visualisierung. Beispiele für ein deklaratives Vorgehen finden sich z.B. in [7].

Eine manuelle Instrumentierung von Byte- oder Maschinencode wird in der Regel nicht durchgeführt, da es für einen Programmierer einfacher ist, die zusätzlichen Anweisungen im Quelltext unterzubringen, als sie in ein Kompilat einzufügen, dessen Lesbarkeit im Allgemeinen stark eingeschränkt ist.

Das Beschreiben manueller Instrumentierung legt nahe, dass es auch eine automatische Instrumentierung gibt. Diese kann jedoch im Rahmen der Algorithmenvisualisierung nicht oder nur stark eingeschränkt angewandt werden, da bisher kein Algorithmus bekannt ist, der Visualisierungsanweisungen an die richtigen Stellen beliebiger Algorithmenimplementierungen einfügen kann.

Automatische Bytecodeinstrumentierung ist jedoch in anderen Einsatzgebieten von praktischem Nutzen, z.B. bei der Analyse von Unittests (siehe Kap. 7.4, Cobertura), beim Profiling (siehe Kap. 7.3, Profiling) oder bei der aspektorientierten Programmierung unter dem Stichwort *Weaving* ([18]).

1.5.2 *Andocken an die Laufzeitumgebung*

Andocken an die Laufzeitumgebung muss von dieser unterstützt werden

Eine weitere Möglichkeit, den Ablauf eines Programms zu beobachten, bietet das Ausnutzen von dafür vorgesehen Schnittstellen der Laufzeitumgebung. Die Laufzeitumgebung kann dabei z.B. eine virtuelle Maschine oder das Betriebssystem sein, auf dem das Programm läuft.

Falls die Laufzeitumgebung geeignete Schnittstellen definiert, werden diese i.d.R. auf das ereignisgesteuerte Vorgehen zugeschnitten sein, d.h. die Laufzeitumgebung wird Interessenten über bestimmte auftretende Ereignisse während des Programmablaufs informieren.

Als Ereignis kann prinzipiell jede Operation, die zu einer Zustandsänderung führt, aufgefasst werden. Da jedoch nicht immer alle Operationen von Interesse sind, bietet es sich an, dass sich der Interessent explizit für bestimmte Ereignisse anmeldet, z.B. für den Zugriff auf eine bestimmte Variable.

1.6 Rollenaufteilung

Im Rahmen der Algorithmenvisualisierung kann zwischen den im folgenden beschriebenen Rollen unterschieden werden, die bestimmte anfallende Tätigkeiten bündeln. Obwohl die Rollen hier getrennt beschrieben werden, wird es in der Praxis häufig so sein, dass zwei unterschiedliche Rollen mit ein und derselben Person bzw. Gruppe besetzt werden.

Rollenname an MVC angelehnt

Eine Analogie zum Architekturmuster MVC ([12]) in der Softwareentwicklung ziehend, kann

- der *Algorithmenimplementierer* als zuständig für das *Model* ansehen werden, dass er durch seine konkrete Implementierung vorgibt.
- der *Kontrollvisualisierer* als zuständig für den *Controller* angesehen werden, da er Modifikationen an der Visualisierung aufgrund von Ereignissen im Modell vornimmt.
- der Visualisierer als zuständig für die *View* angesehen werden, da er die letztendliche grafische Darstellung vornimmt.

1.6.1 *Frameworkentwickler*

Entwickelt ein Werkzeug wie Kartina

Der Frameworkentwickler implementiert ein Werkzeug, dass die Erstellung von Algorithmenvisualisierungen ermöglicht bzw. vereinfacht. Das Ziel des Frameworkentwicklers muss es sein, dem Visualisierer ein Maximum an automatischer Visualisierung zukommen zu lassen und ihn dabei in der grafischen Darstellung so wenig wie möglich zu begrenzen.

Sieht man Adobe Flash[3] einmal als Werkzeug zur Erstellung von Algorithmenvisualisierungen und stellt es einem System wie dem im Rahmen dieser Thesis erstellten Framework gegenüber, so wird klar, dass Flash zwar maximalen Freiraum in der grafischen Darstellung lässt, aber wenig Unterstützung für eine automatische Visualisierung bietet, während Kartina die Implementierung automatischer Visualisierung stark vereinfacht, dafür jedoch die grafische Darstellung einschränkt.

1.6.2 *Algorithmenimplementierer*

Implementiert zu visualisierende Algorithmen

Der Algorithmenimplementierer setzt einen Algorithmus in einer Programmiersprache um und erstellt so eine konkrete Implementierung, die im Rahmen der dynamischen Visualisierung verwendet werden kann. Im Rahmen einer statischen Visualisierung kann die Rolle des Algorithmenimplementierers unbesetzt bleiben, da hierbei keine konkrete Implementierung des Algorithmus benötigt wird.

Im Idealfall braucht der Algorithmenimplementierer keine Kenntnisse über das zum Einsatz kommende Visualisierungswerkzeug zu besitzen, d.h.

3 http://www.adobe.com/products/flash/

er kann die Implementierung des Algorithmus frei gestalten und unterliegt keinerlei Einschränkungen, z.B. bei der

- eingesetzten Programmiersprache
- Vergabe von Variablennamen
- Verwendung von Kontrollstrukturen
- Benutzung von externen Bibliotheken.

In der Praxis wird dem Algorithmenimplementierer zumeist jedoch die Programmiersprache vorgegeben. Dabei kann es sich um eine populäre und allgemein verwendete Sprache handeln, z.B. Java oder C oder um eine speziell zum Zweck der Visualisierung entworfene Sprache.

1.6.3 *Visualisierer*

Der Visualisierer ist für die grafische Darstellung des ablaufenden Algorithmus zuständig. Seine Rolle kann noch weiter in die beiden im folgenden beschriebenen Rollen unterteilt werden, deren Namen erneut an das MVC-Entwurfsmuster angelehnt wurden.

View

Erstellt die grafische Darstellung

Der ausschließlich an der grafischen Darstellung von Datenstrukturen oder Operationen arbeitende Visualisierer erstellt allgemein verwendbare Repräsentationen, deren Fokus auf Wiederverwendbarkeit liegt und die vor allem zur automatischen Visualisierung genutzt werden können.

Ein bekanntes Beispiel hierfür ist die Darstellung einer Liste mit ganzzahligen Werten als Reihe von Balken, wobei die Länge eines Balkens die Höhe des Werts widerspiegelt, den er repräsentiert. Zeiger auf bestimmte Felder der Liste können in der grafischen Darstellung als Pfeile wiedergegeben werden, die auf den entsprechenden Balken zeigen.

Diese Form der Darstellung ist für die meisten auf Listen arbeitenden Sortierverfahren ausreichend, so dass bei der Erstellung einer Visualisierung des Quicksort-Algorithmus auf die Balkendarstellung zurückgegriffen werden kann.

Für einen Algorithmus, der eingehende Nachrichten auf festen `m` Plätzen verwaltet und der bei Überlauf die älteste Nachricht mit der zum Überlauf führenden Nachricht überschreibt, kann jedoch die Darstellung der intern verwendeten Liste als geschlossener Ring mit `m` Plätzen besser geeignet sein, wenn die dem Algorithmus zugrunde liegende Idee visualisiert werden soll.

Controller

Steuert die grafische Darstellung

Die zweite Rolle eines Visualisierers – die des *Kontrollvisualisierers* - kommt vor allem bei der ereignisgesteuerten Visualisierung vor. Hierbei hat der Visualisierer die Aufgabe, auftretende Ereignisse – i.d.R. also die Ausführung bestimmter Quelltextanweisungen – in entsprechende visualisierungsbezogene Anweisungen umzusetzen, d.h. er kontrolliert somit die Visualisierung einer konkreten Implementierung.

Im Rahmen des in Kap. 1.4 erwähnten Tauschvorgangs kann dies die Darstellung der zum Tausch auszuführenden drei Schritte in einem Animationsschritt sein. Stellt man sich den Vorgang der Algorithmenvisualisierung als Schichtenmodell vor (siehe Abb. 1), so ist es auch denkbar, dass der Visualisierer einer niedrigen Schichten mehrere *quelltextbezogene* Ereignisse zusammenfasst bzw. aufbereitet und anschließend ein *ideenbezogenes* Ereignis an den Visualisierer einer höheren Stufe weiterreicht, der dieses dann mittels einer bestehenden Standardvisualisierung darstellt.

Verglichen mit quelltextbezogenen Ereignissen, die für ausgeführte Anweisungen, z.B. das Schreiben einer Variable oder das Betreten einer Methode generiert werden, liegen ideenbezogene Ereignisse auf einem höheren Abstraktionsniveau. Sie spiegeln Ideen wider, die hinter dem ablaufenden Algorithmus stehen.

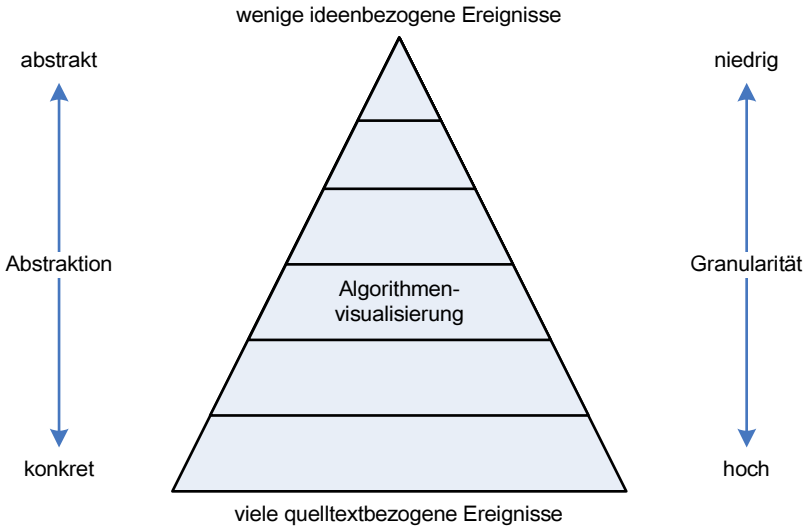

Abbildung 1: Abstraktion im Rahmen von ereignisgesteuerter Visualisierung

Im Kontext des Tauschvorgangs entspricht je ein quelltextbezogenes Ereignis der Ausführung einer der drei nötigen Anweisungen für die Durchführung des Tauschens, wobei beim Eintreten jedes Ereignisses der aktuelle Zustand des Programms abgefragt werden kann. Eine Bündelung dieser drei Ereignisse in einem ideenbezogenen Ereignis heißt, dass

- ein einziges Ereignis für die nächsthöhere Schicht generiert wird, welches den gesamten Tauschvorgang repräsentiert.
- die das ideenbezogene Ereignis verarbeitende Schicht den Zustand des Programms nach Ausführung des Tauschvorgangs abfragen kann. Der Tauschvorgang an sich muss jedoch auf Quelltextebene noch nicht abgeschlossen worden sein.

Für eine ansprechende Visualisierung des Tauschvorgangs, z.B. durch eine Animation, ist es zudem notwendig, dass die visualisierende Schicht Zugriff auf Informationen darüber erhält, welche Felder vertauscht wurden.

1.6.4 *Vorführer*

Der *Vorführer* ist diejenige Person, die die Visualisierung eines bestimmten Algorithmus abschließend durchführt, z.B. im Rahmen einer Vorlesung oder einer Unterrichtsstunde.

Für ihn steht die einfache Auswahl, Parametrisierung und Steuerung des Algorithmus bzw. der Visualisierung im Vordergrund.

2 FUNKTIONALE ANFORDERUNGEN

Die anfängliche Idee, die den Anstoß zu dieser Thesis gab und daher letztendlich auch den Grundstein für Kartina legte, war ebenso einfach wie unspezifisch: es sollte ein Werkzeug bereit gestellt werden, mit dem in Java implementierte Algorithmen mit möglichst geringem Aufwand visualisiert werden können.

Da das Werkzeug im Rahmen der Ausbildung von Studenten der Informatik in den unteren Semestern genutzt werden soll, ist es wichtig, dass viel Spielraum bei der grafischen Gestaltung der Visualisierung zur Verfügung steht.

Im weiteren Verlauf kristallisierten sich weitere im folgenden beschriebene Anforderungen heraus:

1. Die Java-Implementierung muss frei von visualisierungsbezogenen Anweisungen sein, d.h. es kann keine Quelltextinstrumentierung durchgeführt werden.

2. Eventuelle Einschränkungen bei der Verwendung von Java sollen so gering wie möglich ausfallen. Generell muss es möglich sein, sämtliche Sprachkonstrukte und Bibliotheken zu nutzen.

   Allgemeiner formuliert soll der Algorithmenimplementierer möglichst nicht wissen müssen, dass seine Implementierung später visualisiert werden soll.

3. Iterative und rekursive Algorithmen müssen visualisierbar sein.

4. Das Werkzeug muss eine einfache Auswahl der bestehenden Visualisierungen ermöglichen.

5. Die aktuelle Position innerhalb des Quelltextes der Algorithmusimplementierung muss ersichtlich sein.

6. Es muss möglich sein, den Algorithmus auf unterschiedlichen Datensätzen operieren zu lassen. Bei der Visualisierung eines Sortieralgorithmus kann es z.B. von besonderem Interesse sein, den Algorithmus auf einer bereits sortierten Liste arbeiten zu lassen.

7. Die Ablaufgeschwindigkeit des Algorithmus bzw. der Visualisierung muss eingestellt werden können, der Ablauf muss angehalten und fortgeführt werden können.

8. Zu einer Visualisierung gehören
   a) der Algorithmus
   b) eine Eingabemaske für die Daten, auf denen der Algorithmus operiert
   c) die für die Steuerung der Visualisierung benötigte Einheit
   d) die Visualisierung, also die grafische Repräsentation des ablaufenden Algorithmus

Der Ausschluss der Quelltextinstrumentierung führt leider dazu, dass vermehrt mit den Zeilennummern zur Identifikation von Quelltextstellen gearbeitet werden muss. Daher können Änderungen im Quelltext Änderungen in der Visualisierung nach sich ziehen, falls sich die Zeilennummern durch selbige referenzierter Quelltextstellen geändert haben.

Da jedoch im Regelfall der Quelltext eines Algorithmus bereits komplett feststeht, bevor mit seiner Visualisierung begonnen wird, schränkt dieser Umstand die Flexibilität des Visualisierers nur minimal ein.

3 VERWANDTE ARBEITEN

Da es sich bei der Algorithmen- oder Softwarevisualisierung nicht um ein neues Themengebiet handelt, gibt es bereits zahlreiche Werkzeuge, die die Entwicklung diesbezüglicher Animationen unterstützen. Einen Überblick darüber bietet u.a. Diehl [8].

Unter Beachtung der in Kap. 2 aufgestellten Anforderungen scheiden jedoch bereits zahlreiche dort erwähnte Werkzeuge aus, da sie z.B. nicht dafür ausgelegt sind, Java-Implementierungen zu visualisieren. Außerdem scheiden viele Werkzeuge aus, da sie seit Jahren nicht mehr weiterentwickelt wurden und oftmals nur noch unvollständig, z.B. ohne Dokumentation, zu beziehen sind.

Auch die vielen Einzelanimationen, die z.B. als Java-Applet vorliegen, werden an dieser Stelle nicht behandelt, da sie kein Werkzeug zur Visualisierung beliebiger Algorithmen darstellen.

Die Liste der im folgenden vorgestellten Werkzeuge zur Algorithmenvisualisierung erhebt keinen Anspruch auf Vollständigkeit, sondern gibt einen Überblick über bekannte oder vielversprechende Programme aus dem Themengebiet der Softwarevisualisierung.

3.1 Jeliot

setzt auf Selbstvisualisierung

Bei Jeliot[1] handelt es sich um eine Familie von Java-Programmen (Jeliot I, Jeliot2000, Jeliot 3), die die automatische Visualisierung nach dem sogenannten *self-animation paradigm* ermöglicht.

Während Jeliot I das Erstellen eigener Visualisierungen ansatzweise unterstützt (aber seit 1997 nicht mehr weiterentwickelt wurde), bietet Jeliot 3 nur noch eine fixe Sicht auf das ablaufende Programm. Dies resultiert aus der Fokussierung auf absolute Anfänger als Zielgruppe.

Im Handbuch von Jeliot wird Jeliot 3 als Animationssystem für den Einsatz im Rahmen des Anfängerunterrichts beschrieben, das Programme automatisch und ohne zusätzliche Modifikationen durch den Benutzer animiert. Dies schränkt zwar den Spielraum für die Animierungen ein, führt aber dazu, dass Jeliot sehr einfach zu bedienen ist, so dass es nicht nur von den Anfängern schnell akzeptiert wird, sondern auch von den Lehrkräften, die sich nicht erst in die Erstellung von Animationen einarbeiten müssen [9].

Aufgrund der Zielgruppe und des Selbstvisualisierungsparadigmas visualisiert Jeliot 3 das ablaufende Programm immer auf der Quelltextebene. So wird z.B. beim Zugriff auf eine Konstante eben diese aus einer Konstantenbox, die sich links unten im Programmfenster befindet, geholt und auf die Bühne gezogen; algebraische Operationen werden schrittweise dargestellt und ausgeführt. Für die Visualisierungen im Rahmen dieser Thesis werden jedoch weitaus abstraktere Darstellungen des aktuellen Programmzustandes benötigt.

1 http://cs.joensuu.fi/jeliot/

Zur Ausführung des zu visualisierenden Java-Codes verwendet Jeliot 3 den Java Interpreter DynamicJava[2]. Da die letzte Nachricht auf der Homepage von Juni 2002 ist, liegt die Vermutung nahe, dass DynamicJava nicht mehr weiterentwickelt wird. Des Weiteren bietet es daher keine Unterstützung der mit Java 1.5 eingeführten Neuerungen.

3.2 Animal

statische oder dynamische Visualisierung mit Quelltextinstrumentierung

Animal[3] ist ein Java-Programm, dass es ermöglicht, Algorithmenanimationen über eine Drag & Drop unterstützende Oberfläche, ähnlich einem GUI Builder[4], zu erstellen. Mit AnimalScript bietet es zudem eine eigene Skriptsprache, mit der die einzelnen Objekte auf der Bühne animiert werden können.

Animals GUI Builder ähnelt Programmen wie Adobe Flash oder Microsoft Powerpoint – auf die es auch unter dem Menüpunkt „verwandte Systeme“verweist – und bietet keine Möglichkeit, implementierte Algorithmen auf unterschiedlichen Datensätzen operieren zu lassen.

Die AnimalScript API ermöglicht jedoch das Generieren von AnimalScript-Skripten aus Java heraus (so genannte Generatoren) und erlaubt so den Umgang mit unterschiedlichen Eingabedaten.

Auf der Homepage des Projekts finden sich bereits über 40 Einzelvisualisierungen, darunter Standardalgorithmen aus den Gebieten Sortieren und Suchen, aber auch das 0/1-Rucksackproblem, die LZW-Kodierung, die arithmetische Kodierung (siehe jeweils [22]) sowie Generatoren aus verschiedenen Themengebieten. Besonders umfangreich ist der Generator für Graphen und Graphalgorithmen, der das Erstellen von Graphen über textuelle Eingaben (z.B. über eine Adjazenzmatrix) sowie über eine grafische Komponente ermöglicht.

Leider ist es nicht bei jeder Visualisierung möglich, den Quelltext des Algorithmus anzuzeigen, da es in der Verantwortung des Visualisierers liegt, für einen Quelltextansicht zu sorgen. Die Algorithmen, bei denen der Quelltext angezeigt wird, bieten zudem kein Syntaxhighlighting.

Die Tatsache, dass der letzte Eintrag auf der Projektwebseite von Juli 2007 ist, lässt den Schluss zu, dass Animal noch weiterentwickelt wird.

3.3 j-Algo

dynamische Visualisierung mit Quelltextinstrumentierung

j-Algo[5] ist ebenfalls nicht dafür ausgelegt, in Java implementierte Algorithmen ablaufen zu lassen und diesen Ablauf dann zu visualisieren. Stattdessen nimmt es so genannte Module entgegen, die in Java geschriebene Visualisierungen von Algorithmen beinhalten. Im Grunde genommen handelt es sich dabei um eine Implementierung des Algorithmus, die mit visualisierungsbezogenen Anweisungen versehen wurden.

Die bereits vorliegenden Visualisierungen sind liebevoll gestaltet und sehr gut in das Hauptprogramm integriert. Beispielsweise erwähnt seien an dieser

2 http://koala.ilog.fr/djava/
3 http://www.animal.ahrgr.de/index.php3
4 http://en.wikipedia.org/wiki/Graphical_user_interface_builder
5 http://j-algo.binaervarianz.de/

- das Modul Dijkstra [22], das eine ähnliche Eingabekomponente für Graphen bietet wie Animal. Diese ist zwar nicht ganz so umfangreich, dafür ist aber die grafische Eingabekomponente komfortabler zu bedienen.
- das Modul Knuth-Morris-Pratt [22], das Protokoll über die im Algorithmus getroffenen Entscheidungen und die daraus resultierenden Aktionen führt.

Leider bieten auch bei j-Algo nicht alle Module die Anzeige des Quelltextes an und wenn, dann wiederum ohne Syntaxhighlighting, da es auch bei j-Algo dem Visualisierer überlassen bleibt, eine Quelltextanzeige zu implementieren.

Der frei verfügbare Quelltext von j-Algo weißt eine klare Struktur auf und ist ausreichend dokumentiert. Den Neuigkeiten der Projektwebseite lässt sich entnehmen, dass intensiv an j-Algo bzw. an Modulen dafür gearbeitet wird.

3.4 Publikationen

Animal und insbesondere Jeliot haben den Vorteil, dass es bereits zahlreiche Publikationen über sie gibt. Von besonderem Interesse sind die in Ben-Ari u. a. [3] erwähnten Arbeiten von Stasko und anderen, die der Frage nachgehen, inwieweit Visualisierung für das Verständnis von Algorithmen förderlich ist.

Ebenfalls interessant in diesem Zusammenhang sind die in [10] aufgestellten Anforderungen an Algorithmenvisualisierungen sowie die dort genannten Anregungen.

Teil II

KARTINA

4 ENTWURFSPRINZIPIEN

Entwickeln gegen Schnittstellen Alle[1] Klassen von Kartina wurden gegen Schnittstellen entwickelt und können daher ohne großen Aufwand ausgetauscht werden. Komplexe Schnittstellen setzen sich dabei über Vererbung aus einfacheren Schnittstellen zusammen, so dass eine klare Aufteilung der Zuständigkeiten gewährleistet bleibt.

Dependency Injection Der konsequente Einsatz von Dependency Injection[2] erhöht das Maß, in dem Klassen über Modultests mittels Mocking getestet werden können und fördert so die Qualität der Software [21/Kap. 6, 11].

Hierbei stützt sich der zu testende Dienstnehmer nur auf Schnittstellen ab, die konkreten Dienste bekommt er von außen injiziert.

Kommt der Dienstnehmer im System zum Einsatz, werden ihm die benötigten Dienste als konkrete Implementierungen zur Verfügung gestellt. Wird der Dienstnehmer stattdessen im Rahmen eines Modultests eingesetzt, so bekommt er statt der richtigen Dienste nur Mockobjekte (siehe Kap. 7.4) injiziert, so dass der zu testende Dienstnehmer unabhängig von den in Anspruch genommenen Diensten getestet werden kann.

Dependency Injection ist mit dem Konzept der Fabrikmethoden verwandt und kann als dessen Verallgemeinerung angesehen werden.

Lose Kopplung Lose Kopplung von Softwarekomponenten, u.a. erreicht durch das Beobachter-Entwurfsmuster, das Beachten des Gesetzes von Demeter (in [12/S. 265]) passender als *Principle of Least Knowledge* bezeichnet) und den Einsatz von Dependency Injection, resultieren in einer flexibleren, komponentenbasierten Architektur und somit letztendlich in flexiblerem Code [12/S. 53, 4/Kap. 17, 19/Kap. 17].

Änderungen im Rahmen einer Weiterentwicklung von Kartina wirken sich daher i.d.R. nur lokal aus und ziehen keine Änderungen an weiteren Teilen des Programms nach sich.

1 Ausnahmen bilden lediglich Hilfsklassen und die visualisierungsunabhängigen Klassen des GUIs

2 Manuelle DI mittels setter-Methoden. Dem Einsatz eines DI-Frameworks steht jedoch nichts im Wege.

5 FRAMEWORK

5.1 Grundlage JDI

5.1.1 *Begrifflichkeiten*

Vor der Einführung des *Java Debugger Interfaces* seien an dieser Stelle einige grundlegende Begriffe erklärt, die im Allgemeinen entweder gar nicht gebräuchlich sind oder nur in einem bestimmten Sinn benutzt werden.

der Debugger überwacht und steuert die Ausführung des Debuggees

Im Rahmen des JDI wird mit *Debugger* ein Programm bezeichnet, das auf das JDI aufsetzt und über selbiges die Ausführung eines anderen Programms – des *Debuggees* – beobachtet. Es sei an dieser Stelle darauf hingewiesen, dass dieser Debugger zwar auch die Aufgaben eines klassischen Debuggers übernehmen kann (z.B. das schrittweise Ausführen des Debuggees), dass dies aber letztendlich nicht mit der Bezeichnung „Debugger“einhergeht.

In späteren Kapiteln dieser Thesis gesellen sich zu dem Debuggee noch die *Zielklasse* sowie die *Zielmethode*. Während unter Debuggee das gesamte Programm zu verstehen ist, dessen Ausführung über das JDI beobachtet wird, bezeichnet die Zielklasse nur die Klasse, die den zu visualisierenden Algorithmus beinhaltet. Analog dazu ist unter der Zielmethode diejenige Methode zu verstehen, die letztendlich den Algorithmus zur Ausführung bringt.

Angenommen, es existieren die in folgendem Quelltextauszug aufgeführten Klassen und Methoden

```
1  public class SchlechterStil {
2     public static void main(String[] argv) {
3        if (argv[0] == "a") {
4           new A().sortiere();
5        } else if {argv[0] == "b" {
6           new B().suche();
7        } else {
8           new X().teile();
9        }
10    }
11 }
```

Listing 2: Debuggees vs. Zielklasse

und aus Sicht des Vorführers soll `B.suche()` visualisiert werden, wofür die Ausführung von `SchlechterStil` beobachtet werden muss. Der Begriff des Debuggees umfasst nun alle beteiligten Klassen, also `SchlechterStil`, `A`, `B` und `X`. Die Zielklasse bezieht sich hingegen ausschließlich auf B und die Zielmethode ist naheliegenderweise `B.suche()`.

Die Einführung der beiden Begriffe erscheint an dieser Stelle eventuell umständlich oder überflüssig. Sie helfen jedoch an späterer Stelle der Thesis, die einzelnen Klassen, die den Debuggee ausmachen, besser auseinander zu halten.

5.1.2 *Einleitung*

Aufgrund der in Kap. 2 aufgeführten Anforderungen 1, 2, 5 und 7 lag es nahe, das Visualisierungsframework mittels eines Debuggers zu implementieren. Übliche Debugger wie z.B. der in der Netbeans IDE[1] integrierte Debugger ermöglichen es, den Programmcode schrittweise auszuführen, den aktuellen Zustand den Programms, sprich die Werte der sichtbaren Variablen abzufragen und den Ablauf an bestimmten Stellen (Ausführen einer Zeile, Betreten/Verlassen einer Methode) zu unterbrechen, ohne dafür den Quelltext des Programms durch besondere Anweisungen verschmutzen zu müssen.

Andocken an die Java-VM über das JDI

Da die Algorithmen in Java implementiert werden sollen, konnte Kartina auf dem *Java Debug Interface* (JDI) aufgebaut werden.

Hierbei handelt es sich um eine[2] Schnittstelle, die es einem Java-Programmierer erlauben, einen Debugger für Java-Programme in Java zu schreiben, ohne dabei auf ein bestimmtes Wirtssystem achten zu müssen. Der Begriff Wirtssystem umfasst hierbei nicht nur wie üblich die Systemhardware und das Betriebssystem, sondern auch die Implementierung der virtuellen Java Maschine, auf der sowohl der Debugger als auch das zu debuggende Programm laufen.

Streng genommen bildet nicht das JDI allein, sondern die *Java Platform Debugger Architecture*[3] (JPDA) die Grundlage für die Implementierung eines Java-Debuggers. Da der Entwickler eines Debuggers jedoch i.d.R. nur mit dem JDI arbeitet, wird in dieser Thesis nur auf selbiges verwiesen.

Die in diesem Zusammenhang spezifizierten Schnittstellen sind

JVM TI Das Java VM Tool Interface definiert die im Rahmen einer Debuggingsitzung benötigten Dienste, die eine virtuelle Maschine bereitstellt. Die Implementierung dieser Schnittstelle durch eine virtuelle Maschine ermöglicht den einfachen Wechsel der Debuggerarchitektur, z.B. eine Änderung des Kommunikationskanals.

JDWP Das Java Debug Wire Protocol definiert das Format, in dem die Kommunikation stattfindet, jedoch nicht den Kanal, auf dem sie durchgeführt wird. Die Definition des Protokolls ermöglicht es u.a., dass Debuggee und Debugger nicht nur in getrennten virtuellen Maschinen, sondern auch auf getrennten Systemen laufen. Mit dem JDWP kann auch auch ein nicht in Java implementierter Debugger mit dem Debuggee kommunizieren.

JDI Das Java Debug Interface vereinfacht den debuggerseitigen Zugriff auf den Debuggee, der ansonsten auch direkt über das JDWP erfolgen kann. Aufgrund der Komplexität des Protokolls empfiehlt Sun jedoch ausdrücklich die Nutzung des JDI für die Entwicklung von Debuggern. Das JDI stellt daher die Schnittstelle dar, gegen die der Debuggerentwickler letztendlich (ausschließlich) entwickelt.

Da Debugger und Debuggee über ein Protokoll miteinander kommunizieren und da sie jeweils in ihrer eigenen Umgebung – hier die virtuelle

1 http://www.netbeans.org/
2 genauer gesagt mehrere, siehe nächsten Absatz
3 http://java.sun.com/javase/technologies/core/toolsapis/jpda/

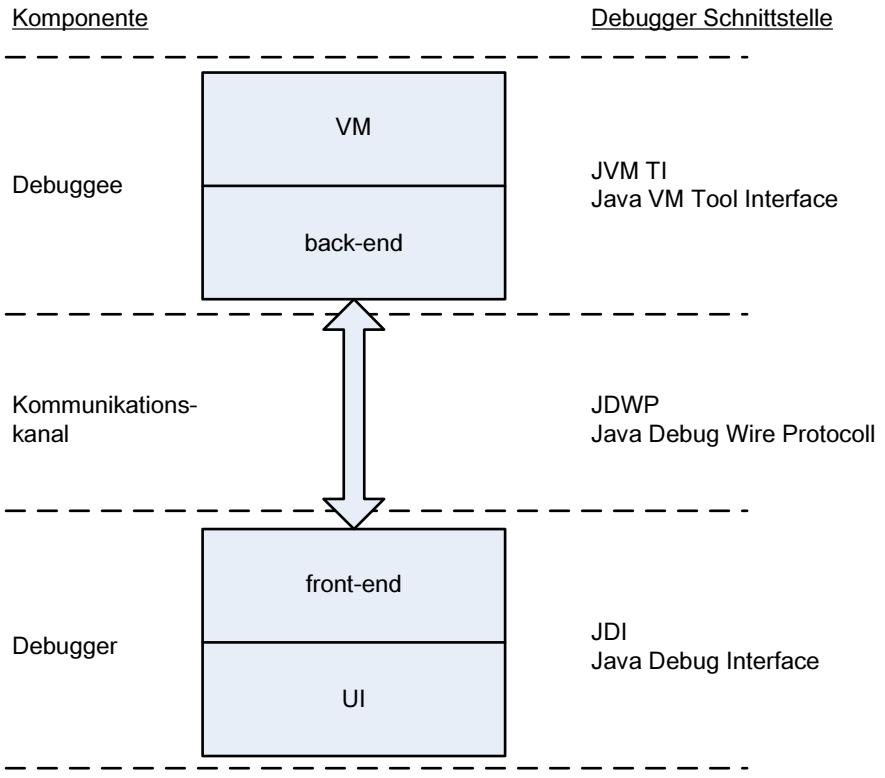

Abbildung 2: Die Architektur des JDI

Maschine – laufen, lässt sich ein Vergleich zu Rechnernetzen ziehen: der Debugger stellt den lokalen Rechner dar, der Debuggee den entfernten Rechner. Diese Analogie liegt auch der Benennung des in Kap. 5.4.10 vorgestellten RemoteSupporters zu Grunde.

JDK beinhaltet bereits eine Implementierung des JDI

Mit dem Java Development Kit (JDK) liefert Sun Microsystems auch eine Referenzimplementierung des JDI aus, die sich im Hauptverzeichnis des JDKs in `lib/tools.jar` befindet. Die API-Dokumentation kann wie üblich auf der Java-Webseite[4] eingesehen werden.

Zusätzlich dazu befindet sich in `demo/jpda/examples.jar` bereits ein einfacher Debugger, der den Einstieg in das JDI erheblich erleichtert. Insbesondere die im Rahmen dieser Thesis entwickelten Klassen `kartina.debugger.VMLauncher`, `kartina.debugger.StreamRedirectThread` und `kartina.debugger.events.DefaultEventReader` wurden maßgeblich von diesem Debugger beeinflusst.

5.1.3 *Requests und Events*

Das JDI ermöglicht das ereignisbasierte Debuggen eines Java-Programms. Von besonderer Bedeutung sind hierbei die beiden im folgenden näher beschriebenen Klassen.

Ein *EventRequest* repräsentiert die Anforderung, beim Eintreten einer bestimmten Situation während der Ausführung des Debuggees ein *Event* zu generieren, das die eingetretene Situation widerspiegelt.

4 http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/index.html

Beispiele für einen Request sind der BreakpointRequest, der zur Generierung von Ereignissen vor der Ausführung einer bestimmten Quelltextzeile führt und der ModificationWatchpointRequest, der der Ereignisgenerierung bei schreibendem Zugriff auf eine Instanzvariable zugrunde liegt.

Ereignisse unterbrechen die Ausführung des Debuggees

Für jeden Request kann festgelegt werden, ob die von ihm generierten Ereignisse für eine Unterbrechung der virtuellen Maschine des Debuggees sorgen, oder nicht. Wird die VM angehalten, muss ihre Ausführung manuell wieder aufgenommen werden. Ist keine Unterbrechung erwünscht, läuft sie nach der Ereignisgenerierung einfach weiter.

In der Regel ist jedoch eine Unterbrechung der VM erforderlich, da die meisten Informationen über den aktuellen Programmzustand, z.B. Variablenbelegungen, nur abrufbar sind, wenn der Ablauf der virtuellen Maschine unterbrochen wird. Dies liegt darin begründet, dass Debugger und Debuggee nebenläufig ausgeführt werden und sich der Zustand des Debuggees während der Auflösung einer seiner Variablen ändern kann, so dass das Ergebnis der Auflösung einen nicht existierenden Zustand widerspiegelt.

Angenommen, der Debuggee führt folgende Schleife aus,

```
1  // k = 0, i = 8, j = 2;
2  while (k < 100) {
3     k = i + j;
4     i = i * 2;
5
6     // ...
7     // ... lesender Zugriff auf j ...
8     // ...
9
10    j = j * 2;
11 }
```

Listing 3: Eine rechnende Schleife

und der Debugger, der über den schreibenden Zugriff auf `k` in Zeile 3 informiert wurde, lässt selbige Variable auflösen. Da die virtuelle Maschine des Debuggees nicht unterbrochen wurde, werden `i` und `j` aufgelöst, während der Debuggee die Zeilen 6 bis 9 ausführt[5]. Dies führt dazu, dass `k` innerhalb der Schleife mit `10, 20, 40, 80` belegt wird, während die Auflösung im Debugger die Werte `18, 36, 72, 144` ergibt.

Wird die VM des Debuggees nicht angehalten, kann zudem die Situation eintreten, dass eine Variable aufgelöst werden soll, die im dann aktuellen Kontext des Debuggees bereits nicht mehr gültig ist.

Unterbrechungen ermöglichen verzögerte schrittweise Ausführung

Weiterhin erzwingt bereits die Tatsache, dass eine sinnvolle Visualisierung eines Algorithmus langsamer abläuft als der Algorithmus selbst, dass die Ausführung des Algorithmus angehalten und erst nach Ablauf einer gewissen Zeitspanne (siehe Kap. 2, Punkt 7) wieder fortgeführt wird[6].

5 Dies ist natürlich auch nur eine Annnahme, eine genaue Aussage darüber, zu welchem Zeitpunkt die Auflösung stattfindet, kann nicht gemacht werden.

6 Wären immer die richtigen Informationen verfügbar, d.h. würde es die vorher genannten Einschränkung der Auflösbarkeit von Variablen nicht geben, ließe sich auch nur die Visualisierung verlangsamen, so dass die Visualisierung noch läuft, obwohl der Algorithmus bereits komplett ausgeführt wurde.

Requests können mit zusätzlichen Einschränkungen versehen werden, die erfüllt werden müssen, bevor ein Event erzeugt wird. So kann z.B. ein BreakpointRequest derart eingeschränkt werden, dass ein Ereignis nur dann erzeugt wird, wenn die ausgewählte Zeile innerhalb einer bestimmten Instanz der Klasse erreicht wurde.

Ereignisgenerierung kann eingeschränkt werden

Von besonderem Interesse ist das Einschränken auf bestimmte Klassen, bei deren Ausführung Ereignisse generiert werden sollen. In der Regel wird man bei der Visualisierung eines Algorithmus ausschließlich über Ereignisse informiert werden wollen, die in der Klasse des Algorithmus auftreten und nicht über Ereignisse, die in den Klassen des JDK auftreten, zum Beispiel in `java.lang.String`.

Requests können zudem beliebig zur Laufzeit aktiviert und deaktiviert werden, wobei Events nur für aktivierte Requests generiert werden.

Ereignisgenerierung muss angefordert werden

Mit der Ausnahme von VMStartEvents und VMDisconnectEvents, die immer generiert werden, werden alle anderen Events nur dann erzeugt, wenn ein entsprechender aktivierter Request vorliegt.

Einem Event lassen sich i.d.R. detaillierte Informationen über das eingetretene Ereignis entnehmen. Diese können den Ort des Eintretens (Klasse, Methode, Zeilennummer) umfassen sowie weitere, der Ereignisart entsprechende Daten beinhalten, z.B. den neuen Wert einer beschriebenen Variable oder die Parameter einer aufgerufenen Methode.

Der Ablauf des Debuggingprozesses mithilfe von Requests und Events lässt sich wie folgt in Pseudocode angeben:

```
1  // ...
2  // Debuggee starten
3  // ...
4  vmConnected = true
5  while (vmConnected) do {
6     ereignis = warteAufNächstesEreignis()
7
8     if (ereignis = VMDisconnectEvent) {
9        vmConnected = false
10    } else {
11       // Auf das Ereignis reagieren,
12       // Requests absetzen
13       // ...
14    }
15 }
16 // ...
17 // Sitzung beenden
18 // ...
```

Listing 4: Debuggen mit Requests und Events

Das JDI definiert zahlreiche Ereignisse, von denen folgende für das Verständnis dieser Thesis relevant sind:

StepEvent wird generiert, wenn eine neue Quelltextzeile[7] ausgeführt wird und ermöglicht somit das schrittweise Ausführen eines Programms.

7 Die Granularität eines Schrittes kann eingestellt werden, siehe Javadoc des com.sun.jdi.request.EventRequestManagers, Methode createStepRequest. In Kartina wird jedoch immer zeilenweise vorgegangen.

BreakpointEvent wird generiert, wenn ein Breakpoint, d.h. eine bestimmte Zeilennummer, erreicht wird.

MethodEntryEvent, MethodExitEvent wird generiert, wenn eine Methode betreten bzw. verlassen wird.

AccessWatchpointEvent, ModificationWatchpointEvent wird generiert, wenn lesend bzw. schreibend auf eine Instanzvariable zugegriffen wird. Für lokale Variablen existiert leider kein vergleichbares Ereignis.

ClassPrepareRequest wird generiert, wenn der Classloader des Debuggees eine Klasse lädt.

5.2 Systemanforderungen

Um den Einstieg in die Implementierung von Kartina zu erleichtern, werden an dieser Stelle zunächst die Anforderungen erhoben, die sich entweder direkt aus der Verwendung des JDI ergeben oder die aus der Kombination der in Kap. 2 aufgestellten Forderungen mit dem JDI resultieren.

1. Bevor der Ablauf eines Programms überhaupt beobachtet werden kann, muss das Programm erstmal im Kontext des JDI gestartet werden. Der hierfür zuständige Dienst wird in Kap. 5.4.1 eingeführt.

2. Da Requests erstellt werden müssen, damit die virtuelle Maschine des Debuggees Ereignisse generiert, muss es einen Dienst geben, der das Erstellen der Requests ermöglicht. Diese Dienste werden in Kap. 5.4.5 behandelt.

3. Die für die Verteilung der auftretenden Ereignisse zuständigen Dienste werden in Kap. 5.4.2 vorgestellt. Analog dazu wird die Fähigkeit, über Ereignisse informiert zu werden, in Kap. 5.4.3 vorgestellt.

4. Da die virtuelle Maschine des Debuggees angehalten wird, wenn ein Ereignis eintritt, muss sie anschließend wieder fortgeführt werden. Der dafür zuständige Dienst wird in Kap. 5.4.9 beschrieben.

5. Das Auflösen von Variablen zum Zeitpunkt eines Ereignisses und somit die Ermittlung des aktuellen Zustands des zu visualisierenden Algorithmus sowie Punkt 2 der Anforderungserhebung (komplette Java-Syntax soll verwendbar sein) werden in Kap. 5.4.6 behandelt.

6. Punkt 5 der Anforderungserhebung (aktuelle Position im Quelltext) wird in Kap. 5.7 angesprochen.

7. Der Dienst, der unter anderem für Punkt 6 der Anforderungserhebung (Operation auf unterschiedlichen Datensätzen) zuständig ist, wird in Kap. 5.4.10 eingeführt.

8. Auf das letztendliche Erstellen einer Visualisierung wird in Kap. 5.5, Kap. 5.6 und in Kap. 6 eingegangen. Dafür, dass der Visualisierer Zugriff auf die von ihm benötigten Dienste erhält, sorgt der in Kap. 5.4.7 beschriebene Dienst.

5.3 Schichtenmodell

Kartina lässt sich strukturell in die in Abb. 3 abgebildeten, aufeinander aufbauenden Schichten unterteilen. Die in der Grafik aufgeführten Klassen werden in den folgenden Abschnitten beschrieben.

Einordnung einer Klasse zu einer Schicht nicht immer unstrittig

Die Grafik stellt eine abstrakte Sicht auf Kartina dar und spiegelt die konkrete Implementierung nur teilweise wider. Listener und Adapter gehören – aus rein struktureller Sicht – zweifelsohne in die zweite Schicht, da sie entworfen wurden, um das Arbeiten mit dem JDI zu vereinfachen. Durch die Abstraktion des Schichtenmodells wird jedoch verschwiegen, dass z.B. der LineExecutedRequestManager und der Visualisierungscontroller auch Listener sind. Da ihre Zuständigkeiten aber in höheren Schichten liegen und die Eigenschaft ein Listener zu sein, ihre Rolle nicht dominiert[8], kann dies bei der Zuordnung zum Schichtenmodell ignoriert werden.

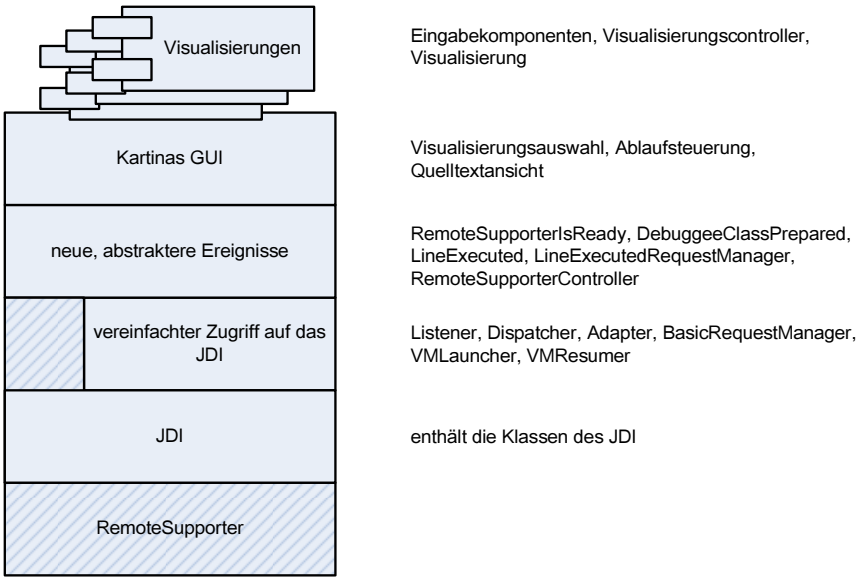

Abbildung 3: Kartinas Architektur als Schichtemodell

Die Visualisierungen sind bewusst als Komponenten abgebildet worden, da sie über die in Kap. 5.6 beschriebene Konfigurationsdatei an Kartina übergeben werden und somit modular[9] sind.

Sonderrole des RemoteSupporters

Die Position des RemoteSupporters innerhalb des Schichtenmodells ist – relativ zu den anderen Klassen bzw. Schichten – nicht eindeutig, weshalb er sich über die unteren beiden Schichten erstreckt. Dieser Entscheidung liegen die Überlegungen zu Grunde, dass

8 Im Gegensatz zur Schnittstelle `Listener`, deren dominierende, da einzige Rolle der Listener ist

9 Durch die in Kap. 4 beschriebenen Entwurfsprinzipien können auch die anderen Schichten als modular angesehen werden, relativ dazu sind die Visualisierungen jedoch weitaus modularer. Zudem würde die Darstellung aller Schichten als Komponenten die Verständlichkeit der Grafik trüben.

- auf den RemoteSupporter über das JDI zugegriffen wird und er deshalb unter selbiger Schicht eingeordnet werden kann.
- aber der RemoteSupporter gleichzeitig eine Vereinfachung des Umgangs mit dem JDI darstellt, weshalb er in der zweiten Schicht eingeordnet werden kann.

Dass der RemoteSupporter eine besondere Rolle innerhalb von Kartina einnimmt, wird daher bereits durch seine schichtenübergreifende Einordnung angedeutet.

5.4 Zugriff auf das JDI

Zugriff vereinfachen und Zuständigkeiten bündeln

Um die Arbeit mit dem JDI zu erleichtern, wurde im Schichtenmodell von Kartina zunächst eine Schicht eingezogen, die den Zugriff auf die Funktionalitäten des JDI vereinfachen und den höheren Schichten als Dienste zur Verfügung stehen. Dies ermöglicht es außerdem, Zuständigkeiten klar zu definieren und zu strukturieren und dem Anwender somit einen einfacheren Zugang zum System zu bieten.

Von besonderem Interesse sind hierbei die im folgenden vorgestellten Klassen.

5.4.1 *VMLauncher*

Der `VMLauncher` startet ein Java-Programm (den Debuggee) in einer eigenen virtuellen Maschine und ermöglicht es dem startenden Programm (dem Debugger), die Ausführung des Debuggees mittels des JDI zu steuern und zu beobachten.

Die Anbindung des Debuggees and den Debugger erfolgt durch einen sogenannten `Connector`. Mit dem JDI liefert Sun drei Konnektoren aus, die

- den Debuggee in einer neuen virtuellen Maschine starten und den Debugger mit dieser verbinden.
- den Debugger mit einer bereits laufenden virtuellen Maschine verbinden.
- auf eine Verbindungsanfrage seitens des Debuggees warten.

Jeder Debuggee wird in seiner eigenen VM gestartet

Der `VMLauncher` nutzt ausschließlich den ersten Konnektor, d.h. der Debuggee wird jeweils in einer eigenen neuen virtuellen Maschine gestartet.

An den Debuggee werden hierbei folgende Anforderungen gemacht:

1. Der Classpath, der die für die erfolgreiche Ausführung des Debuggees benötigten Klassen enthält, muss der neuen virtuellen Maschine bekannt sein.
2. Der Debuggee muss eine startbare Java-Klasse sein, d.h. er muss die `main`-Methode enthalten.

Während die erste Anforderung keine Nachteile mit sich bringt – die den Debuggee startende Methode des `VMLaunchers` nimmt den Classpath des Debuggees als Parameter entgegen – steht die zweite Anforderung im Widerspruch zu Punkt 1 der Anforderungserhebung, dass der Algorithmenimplementierer nicht wissen muss, dass seine Implementierung später visualisiert werden soll.

Diese Einschränkung, vor allem aber die in Kap. 5.4.10 beschriebene Problematik der Übergabe der Datensätze, auf denen der zu visualisierende Algorithmus arbeiten soll, erzwingen daher den Einsatz des in Kap. 5.4.10 beschriebenen RemoteSupporters.

Ereignisse werden sofort in eine Warteschlange eingestellt

Konnte der Debuggee gestartet werden, beginnt sofort der Debuggingprozess, d.h. die Generierung von Events aufgrund von Ereignissen in der virtuellen Maschine des Debuggers. Die auftretenden Ereignisse werden vom JDI in die der virtuellen Maschine zugeordneten `EventQueue` eingestellt, der sie anschließend vom `EventReader` (siehe Kap. 5.4.2) wieder entnommen werden.

5.4.2 *EventDispatcher*

Ein `EventDispatcher` (oder kurz Dispatcher) versendet aufgetretene Ereignisse an `EventListener`, die sich bei ihm registriert haben. Er ist somit dafür zuständig, dass alle an Ereignissen interessierte Objekte auch über selbige informiert werden und auf sie reagieren können.

Beobachter-Entwurfsmuster zur Ereignisverteilung

Mittels EventDispatcher und EventListener wurde das klassische Beobachter-Entwurfsmuster [12] implementiert, um die Verteilung auftretender Ereignisse zu realisieren. Dies hat den Vorteil, dass sich die an Ereignissen interessierten Objekte nicht selber darum kümmern müssen, dass sie auf auftretende Ereignisse reagieren können (abgesehen von einer einmaligen Registrierung am Dispatcher). Außerdem realisiert es das Entwurfsprinzip der losen Kopplung [12/S. 53] zwischen dem Dispatcher und den Listenern.

Die in Kartina verwendete Implementierung des Beobachter-Entwurfsmusters wirft zwei Fragen auf:

1. Mit `java.util.Observable` sowie `java.util.Observer` liegt dem JDK bereits eine Implementierung des Beobachter-Entwurfsmusters bei. Wieso wird dieses nicht verwendet?

2. Beim Beobachter-Entwurfsmuster kann die Übergabe der neuen Daten (hier des aufgetretenen Ereignisses) sowohl vom Dispatcher (Ausliefern) als auch vom Listener (Abholen) initiiert werden. In [12/S. 62f] werden diese Vorgehen mit *push* bzw. *pull* bezeichnet. Warum werden die Ereignisse bei Kartina ausschließlich ausgeliefert?

Zu Punkt 1 führt [12/S. 71] unter anderem an, dass `java.util.Observable` eine Klasse ist und somit von ihr geerbt werden muss. Da in Java nur von einer Klasse geerbt werden kann, ist es zu empfehlen, das Beerben einer Klasse möglichst lange zu vermeiden und stattdessen Schnittstellen zu implementieren. Ob eine Klasse ein Dispatcher (bzw. observable) ist, ist jedoch eine der ersten Entscheidungen während des Entwurfs einer neuen Klasse für Kartina. Daher wurde auf die Verwendung der JDK-Klassen an dieser Stelle verzichtet.

Das in der oben erwähnten Diskussion zu Punkt 2 genannte Argument der Bequemlichkeit war in der Entscheidung im Falle von Kartina ausschlaggebend, da über die hier verwendeten Dispatcher generell nur Ereignisse verschickt werden. Die Ereignisse wiederum bieten den Zugriff auf weitere Informationen, so dass diese nicht über andere Wege ausgetauscht werden müssen. Da die Listener daher nicht auf Getter-Methoden des Dispatchers angewiesen sind, um an diese Informationen zu kommen, ist die Wahl des

push-Verfahrens im Rahmen von Kartina die bessere Wahl. Durch die Verwendung von Filtern kann zudem verhindert werden, dass unnötigerweise Ereignisse an Listener verschickt werden, die daran gar nicht interessiert sind.

Beobachter-Entwurfsmuster kann zu Problemen mit der Garbage Collection führen

Ein Nachteil des Beobachter-Entwurfsmusters im Zusammenhang mit Javas Garbage Collection ist, dass ein eigentlich nicht mehr benötigtes Objekt nicht aus dem Speicher entfernt wird, da es nicht vom Dispatcher abgemeldet wurde und der Dispatcher somit weiterhin eine Referenz auf dieses Objekt hält. Dieses Problem kann aber durch die Verwendung von schwachen Referenzen[10],[11] gelöst werden.
In der derzeitigen Implementierung der Dispatcher werden jedoch keine schwachen Referenzen verwendet, da die Anzahl der Listener nicht so groß ist bzw. seien dürfte, dass es zu Speichermangel kommen könnte. Außerdem werden bei jeder Visualisierung sämtliche Dispatcher neu initialisiert, so dass das Abmelden von Listenern i.d.R. nicht nötig ist.

Es existieren verschiedene Erweiterungen des einfachen Dispatchers, zum Beispiel der `FilteringEventDispatcher`, der es Listenern ermöglicht, mittels `EventFiltern` zu bestimmen, über welche Ereignisse sie informiert werden wollen. Jedem Listener können hierbei beliebig viele Filter zugeordnet werden. Um über ein Ereignis informiert zu werden, reicht es aus, wenn ein einziger Filter das Ereignis durchlässt. U.a. bereits implementiert sind Filter, die die Ereignisse nach Kategorie, nach Klasse oder nach Request filtern.

Eine weitere Erweiterung ist der `PrioritizableEventDispatcher`, der es ermöglicht, die Reihenfolge, in der die Listener benachrichtigt werden, festzulegen. Hierzu wird jedem Listener eine Priorität zugeordnet, nach der die Listener sortiert werden. Dieses Verhalten wird zu Beginn des Visualisierungsprozesses benötigt, da dort die Reihenfolge, in der verschiedene Listener auf Ereignisse reagieren, von vitalem Interesse ist.

EventReader

Entnimmt der Warteschlange die Ereignisse

Der `EventReader` ist ein besonderer Dispatcher, da er zusätzlich für die Entnahme der Ereignisse aus der EventQeue verantwortlich ist. Anhand der auftretenden VM-bezogenen Ereignisse (zum Beispiel VMStartEvent und VMDeathEvent) hält er den Status des Debuggees mit und kann so darüber Auskunft geben, in wie Weit noch mit dem Debuggee zu arbeiten ist.

Weiterhin ermöglicht es der EventReader, die virtuelle Maschine des Debuggees automatisch fortzuführen, nachdem alle Listener benachrichtigt wurden (siehe Kap. 5.4.9).

Der derzeit von Kartina verwendete `DefaultEventReader` ist als Thread implementiert, der der Warteschlange so lange Ereignisse entnimmt, wie die Verbindung zum Debuggee besteht.

PrioritizableFilteringEventDispatcher

Filtert und verteilt Ereignisse, priorisiert Listener

Im Gegensatz zum EventReader, der auftretende Ereignisse an alle registrierten Listener verschickt, bietet der `PrioritizableFilteringEventDispatcher` eine genauere Kontrolle über die Verteilung der Ereignisse. Wie der Name bereits andeutet, handelt es sich bei ihm um eine Implementierung des FilteringEventDispatchers und des PrioritizableEventDispatchers.

10 http://java.sun.com/javase/6/docs/api/java/lang/ref/WeakReference.html
11 http://java.sun.com/javase/6/docs/api/java/util/WeakHashMap.html

Der PrioritizableFilteringEventDispatcher ist dabei gleichzeitig ein EventListener, der als einziger Listener am EventReader angemeldet ist und so über alle auftretenden Ereignisse informiert wird, die er dann wiederum an seine eigenen Listener weitergeben kann.

In Kartina stellt der PrioritizableFilteringEventDispatcher die zentrale Einheit dar, an der sich die Listener anmelden, um über auftretende Ereignisse informiert zu werden. Er verteilt nicht nur die Ereignisse, die direkt vom JDI generiert werden (`com.sun.jdi.event.Event` und deren Unterklassen), sondern auch die Ereignisse, die während des Visualisierungsprozesses von Kartina generiert werden (z.B. `TargetClassPrepareEvent`, `LineExecutedEvent`).

Dekorierer-Entwurfsmuster

Um den Listenern Prioritäten geben zu können, verwendet der PrioritizableFilteringEventDispatcher das Dekorierer-Entwurfsmuster [12], dass die Listener zur Laufzeit mit der Fähigkeit, priorisiert werden zu können, versieht. Hierbei kommen der `kartina.util.GenericDecorator`, die Schnittstelle `kartina.util.Prioritizable` und der `kartina.util.DecoratorIterator` zum Einsatz. Während die ersten beiden Klassen das Dekorierer-Entwurfsmuster realisieren, dient der DecoratorIterator dazu, den Einsatz der Dekorierer für Dritte transparent zu gestalten.

5.4.3 *EventListener*

Die Schnittstelle `EventListener` ermöglicht es Klassen, sich an EventDispatchern zu registrieren und so über auftretende Ereignisse informiert zu werden. Informiert zu werden bedeutet hierbei, dass die Methode `public void eventOccurred(Event event)` mit dem aufgetretenen Ereignis als Parameter aufgerufen wird. Da das Event-Objekt Zugriff auf zahlreiche Informationen bietet, die im Zusammenhang mit dem Ereignis von Interesse sind, hat der Listener so die Möglichkeit, auf das Ereignis reagieren zu können.

Um dies auch differenzierter tun zu können existiert die abstrakte Klasse `ListenerAdapter`, der analog zum von AWT/Swing bekannten MouseAdapter[12] eine Reihe von Methoden mit leeren Rümpfen bereitstellt, die – falls nötig – von der implementierenden Klasse überschrieben werden können. Jede dieser Methoden ist hierbei für die Handhabung eines bestimmten Ereignissen zuständig, z.B. `handleMethodEntryEvent` für das Handhaben des gleichnamigen MethodEntryEvents.

5.4.4 *Klassendiagramm der Dispatcher*

Abb. 4 gibt einen Überblick über die bisher beschriebenen Dispatcher und Listener. Aus Gründen der Übersichtlichkeit werden die Klassenattribute sowie einige Methoden nicht im Diagramm aufgeführt.

Der `AbstractFilteringEventDispatcher` wurde in den vorherigen Abschnitten nicht eingeführt, da er zwar aus Sicht der Softwarearchitektur eine wichtige[13] Klasse darstellt, für das Verständnis der Architektur von Kartina aber uninteressant ist.

12 http://java.sun.com/javase/6/docs/api/java/awt/event/MouseAdapter.html

13 Wichtig in dem Sinne, als dass er die Operationen bündelt, die für die meisten Implementationen eines FilteringEventDispatchers anfallen, so dass diese nicht mehrfach implementiert werden müssen. Durch die Definition abstrakter Methoden bleibt den erweiternden Klassen jedoch noch genug Spielraum, siehe PrioritizableFilteringEventDispatcher.

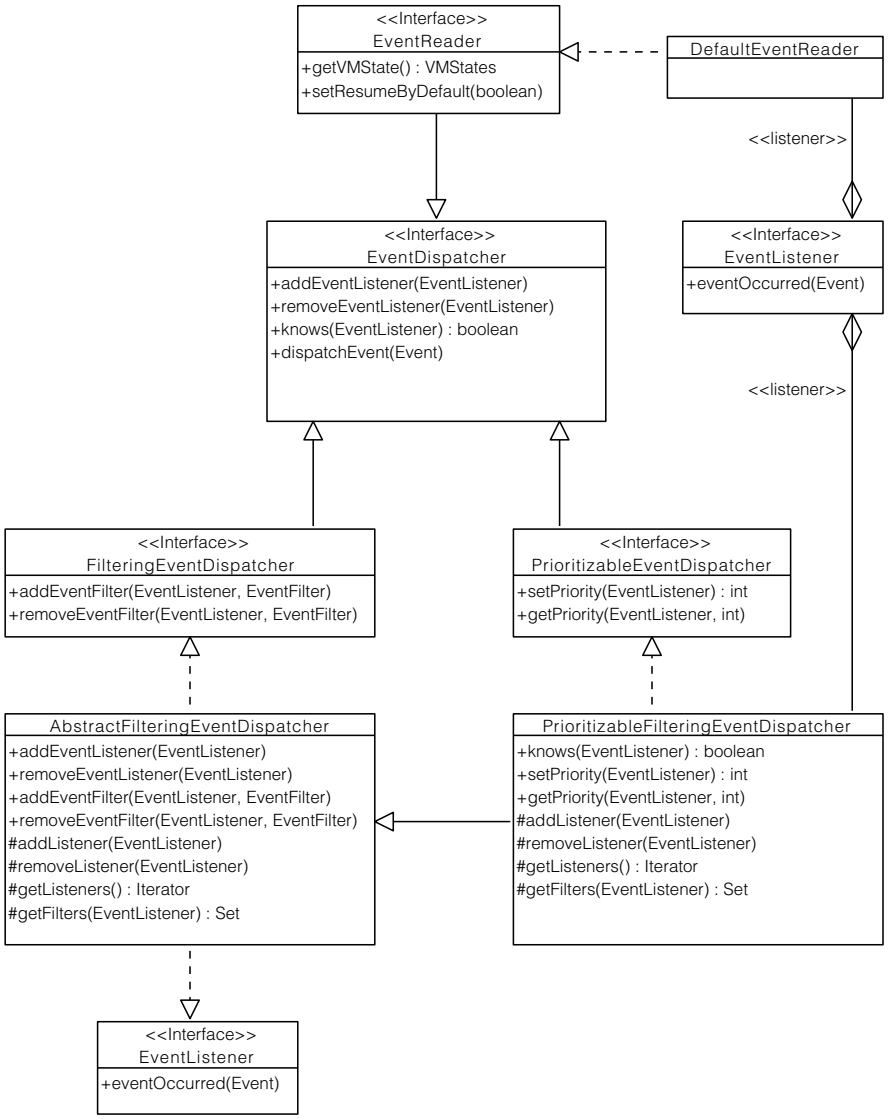

Abbildung 4: Klassendiagramm der Dispatcher, das EventListener-Interface ist nur der Übersichtlichkeit halber doppelt aufgeführt

5.4.5 *RequestManager*

Wie in Kap. 5.1.3 beschrieben, erzeugt das JDI die meisten Ereignisse erst, wenn ein dazu passender Request erzeugt wurde. Dafür stellt das JDI die Klasse `EventRequestManager` bereit, die für alle vom JDI definierten Ereignisse Requests erzeugen kann.

BasicRequestManager

Vereinfacht den Zugriff auf bestehende Funktionen

Da sich der Umgang mit dem EventRequestManager aber nicht einfach gestaltet und der Anwender wiederholt eine Menge zusätzlicher Operationen durchführen muss, um einen Request adäquat zu initialisieren, stellt Kartina den `BasicRequestManager` zur Verfügung, der das anwenderfreundliche Erzeugen von Requests ermöglicht.

Durch die Verwendung eines `ExcludedClassesProviders`, der eine Liste derjeniger Klassen verwaltet, die von der Ereignisgenerierung ausgeschlossen werden sollen, wird insbesondere das Einschränken der Requests auf die im Zusammenhang mit der Visualisierung interessanten Klassen vereinfacht.

Der von Kartina verwendete DefaultBasicRequestManager implementiert – wie auch schon der PrioritizableFilteringEventDispatcher – die Schnittstelle EventListener, da er dem VMStartEvent den EventRequestManager entnehmen muss, den er zur Erzeugung der Requests verwendet. Der DefaultBasicRequestManager wird allerdings nicht mehr am EventReader, sondern mit einer hohen (genauer gesagt sogar mit der höchsten verwendeten) Priorität am PrioritizableFilteringEventDispatcher angemeldet.

Die hohe Priorität ist nötig, da ein anderer Listener, der versuchen würde, mittels des DefaultBasicRequestManagers einen Request zu erzeugen, bevor dieser über den VMStartEvent Zugriff auf den EventRequestManager erhalten hat, eine Exception provozieren würde.

LineExecutedRequestManager

beispielhafte Implementierung eines komplexen künstlichen Ereignisses

Der LineExecutedRequestManager ermöglicht das Erstellen von LineExecutedRequests und er erzeugt die dazu passenden LineExecutedEvents. Diese ähneln den bereits zur Verfügung stehenden BreakpointEvents insoweit, als dass sie generiert werden, sobald eine bestimmte Zeile im Quelltext (identifiziert durch Klassenname und Zeilennummer) ausgeführt wird.

Im Unterschied zum BreakpointEvent, der generiert wird, BEVOR die Zeile ausgeführt wird, wird der LineExecutedEvent generiert, NACHDEM die Zeile ausgeführt wurde. Würden für folgenden Programmausschnitt

```
1 int x = 7;
2 x = 3 * x;
3 int y = -1 * x;
```

Listing 5

jeweils ein Breakpoint- und ein LineExecutedRequest für die Zeilen 2 und 3 erzeugt, so wären die genutzten Variablen zum Eintrittszeitpunkt der jeweiligen Ereignisse wie in Tab. 1 belegt.

Der LineExecutedEvent, der aufgrund eines LineExecutedRequests für eine Zeile `m` generiert wird, wird für den Anwender transparent durch die

im folgenden beschriebene Kombination aus Breakpoint- und StepRequest erzeugt:

1. BreakpointRequest für Zeile `n` erzeugen
2. Beim Auftreten des passenden BreakpointEvents einen StepRequest erzeugen
3. Beim Auftreten des nächsten StepEvents den StepRequest deaktivieren und den LineExecutedRequest erzeugen und vom PrioritizableFilteringEventDispatcher verteilen lassen

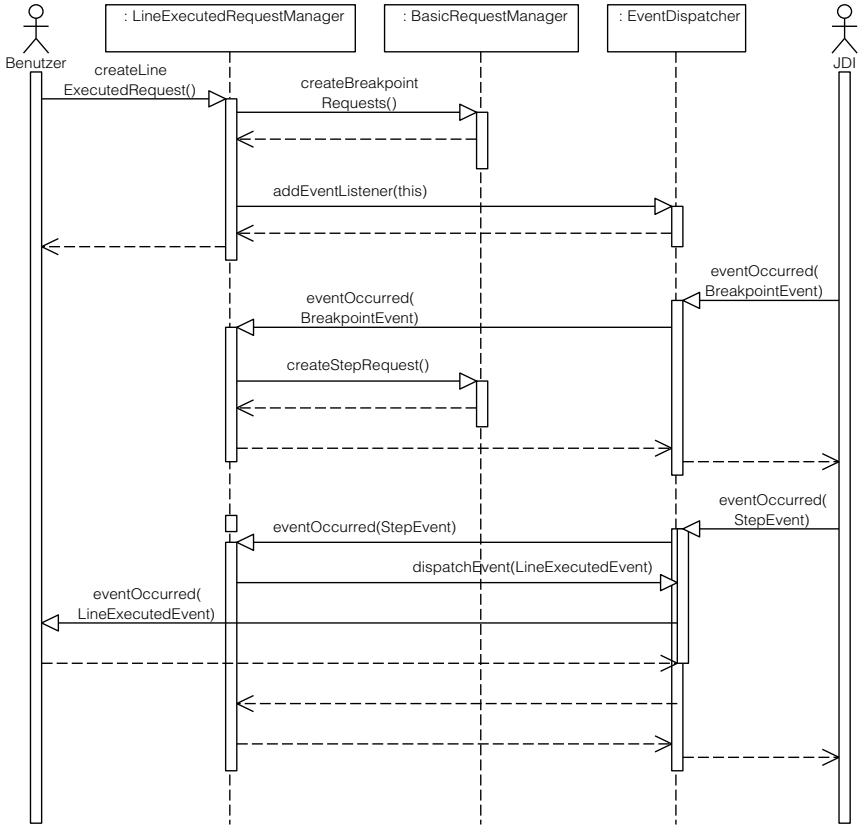

Abbildung 5: Vorgehen des LineExecutedRequestManagers

Ereignis in Zeile	Zustand
Zeile 2, BreakpointEvent	`int x = 7`
Zeile 2, LineExecutedEvent	`x = 21`
Zeile 3, BreakpointEvent	`y unbekannt`
Zeile 3, LineExecutedEvent	`y = -21`

Tabelle 1: Zustände bei unterschiedlichen zeilenbezogenen Ereignissen

Wie dem beschriebenen Ablauf entnommen werden kann, wird der LineExecutedEvent erst erzeugt, wenn die nächste Zeile betreten wurde (aber noch nicht ausgeführt wurde, siehe API-Dokumentation des StepEvents).

Befände sich nun obiger Programmausschnitt am Ende einer Methode, so wäre zum Eintrittszeitpunkt des LineExecutedEvents für Zeile 3 die Methode bereits verlassen worden und die Variable `y` somit unbekannt. Dieser Sachverhalt stellt jedoch keinen Nachteil dar, da eine lokale Variable `y`, die erst am Ende einer Methode gesetzt wird, nicht von Relevanz sein kann (und somit überflüssig ist). Würde es sich bei `y` um eine Instanzvariable handeln, so wäre sie natürlich auch noch nach dem Verlassen der Methode gültig.

Für Kartina selbst wird der LineExecutedEvent nicht benötigt und ob er von großer Bedeutung für konkrete Visualisierungen sein wird, muss an dieser Stelle offen bleiben. Vielmehr dient er als Beispiel für ein neues, auf den Ereignissen des JDI aufbauendes Ereignis und als Beleg dafür, dass sich weitere Ereignisse realisieren und durch den Einsatz des Beobachter-Entwurfsmusters problemlos in die existierende Architektur integrieren lassen.

5.4.6 *ExpressionInterpreter*

wertet Java-Ausdrücke im Kontext eines Debuggees aus

Für die Visualisierung eines Algorithmus muss logischerweise auf den aktuellen Zustand des Algorithmus, d.h. auf die Daten, auf denen er operiert und somit auf seine Variablen, zugegriffen werden können.

Manche Ereignisse ermöglichen den direkten Zugriff auf bestimmte Variablen, auf die sich das Ereignis bezieht. So bietet ein AccessWatchpointEvent Zugriff auf die beobachtete Variable und ein MethodEntryEvent Zugriff auf die Parameter der betretenen Methode.

Vom Umgang mit Debuggern bekannt sind außerdem die folgenden Verfahren zum Zugriff auf den aktuellen Zustand des Debuggees:

1. Auflisten aller zur Zeit gültiger Variablen, meistens realisiert über einen Baum, dessen Wurzel das `this`-Objekt bildet.

2. Das Auswerten von *Watches*, d.h. das Auswerten von Ausdrücken, die aus einer Untermenge der Java-Syntax erstellt werden können und die den Zugriff auf Variablen ermöglichen, beispielsweise `i`, `array[i]`, `this.list.get(i)`, `(foo + bar) * 2`.

Das Kartina zugrunde liegende JDI unterstützt bereits beide Verfahren, wobei im Rahmen von Kartina der Einsatz von Watches vorzuziehen ist. Auf den Zugriff mittels einer Baumstruktur kann verzichtet werden, da

1. keine grafische Exploration des Zustands benötigt wird (anders als in einem Debugger).

2. der Zugriff auf Variablen i.d.R. wissentlich erfolgt, d.h. es ist genau bekannt, auf welche Variable zugegriffen werden soll, somit ist auch der *Zugriffspfad* bekannt.

Die Exploration des Programmzustands über das `this`-Objekt ist im Rahmen von Kartina zwar nicht zu präferieren, kann jedoch bei Bedarf über die vom JDI zur Verfügung gestellten Schnittstellen erfolgen. Es sei an dieser Stelle aber darauf hingewiesen, dass sich dies sehr umständlich gestaltet.

Der `ExpressionInterpreter` ermöglicht das Auswerten von Watches der oben beschriebenen Form im Kontext eines Debuggees und ermöglicht so den Zugriff auf dessen aktuellen Zustand.

Der im Rahmen von Kartina entwickelte `SunExpressionInterpreter` stützt sich auf den dem Java Development Kit beiliegenden `com.sun.tools.example.debug.expr.ExpressionParser` ab, der die Auswertung umfangreicher Ausdrücke ermöglicht. Hierbei handelt es sich sowohl um einen Parser, der mittels JavaCC[14] aus der Java-Grammatik erstellt wurde als auch um einen Interpreter, der den resultierenden Ableitungsbaum im Kontext des Debuggees auswertet.

5.4.7 *DebugEnvironment*

bündelt von Entwicklern benötigte Dienste Kartinas

Der Visualisierer – insbesondere der Kontrollvisualisierer – benötigt für das Erstellen einer Visualisierung Zugriff auf fast alle bisher erwähnten Klassen.

Requests müssen erstellen werden können, es muss sich für daraus resultierende Events registriert werden können und es muss der aktuelle Programmzustand ausgelesen werden können, wenn ein Event auftritt.

Um dem Visualisierer eine zentrale Anlaufstelle für die benötigten Dienste zu bieten, existiert die `DebugEnvironment`. Hierbei handelt es sich um eine einfache, geschäftslogiklose Klasse, die mittels einer Reihe von Getter-Methoden den Zugriff auf die Dienste ermöglicht.

In Kartina existieren zwei Implementierungen der DebugEnvironment. Die erste – die `InternalDebugEnvironment` – ist für den Auf- und Abbau einer Debuggersitzung zuständig. Sie initialisiert die benötigten Dienste, verdrahtet sie untereinander und startet den Debuggee, außerdem ist sie für das Beenden und Aufräumen einer Sitzung zuständig. Die zuvor beschriebenen Aufgaben der DebugEnvironment erfüllt diese Implementierung sozusagen nur nebenbei.

Außerdem instanziiert die InternalDebugEnvironment eine zweite Implementierung der DebugEnvironment, die ausschließlich die vom Interface definierten Getter-Methoden implementiert. Diese `ExternalDebugEnvironment` wird dem Visualisierer mitgegeben, so dass dieser zwar die zur Verfügung stehenden Dienste nutzen, aber keinen (weiteren) Einfluss auf die laufende Debuggersitzung nehmen kann.

5.4.8 *Zusammenspiel der Klassen*

Abb. 7 und Abb. 8 stellen zeitlich aufeinander folgende Abläuf dar

Das Zusammenspiel der im Rahmen von Kartina realisierten Klassen bilden die folgenden Diagramme ab. Das Objektdiagramm in Abb. 6 stellt dabei die Verknüpfungen dar, die zur Laufzeit zwischen den beteiligten Klassen existieren, während die Kommunikationsdiagramme in Abb. 7 und Abb. 8 die Interaktionen zwischen ihnen wiedergeben. Die Kommunikationsdiagramme folgen dabei zeitlich aufeinander, d.h. die Kommunikation, die in Abb. 8 dargestellt wird, beginnt erst, wenn die in Abb. 7 beschriebene Kommunikation stattgefunden hat.

Ebenso wie bisher gilt auch für diese Diagramme, dass sie die reale Situation nicht komplett wiedergeben müssen. Operationen, die für das Verständnis der Abläufe nicht relevant sind, können weggelassen werden.

14 https://javacc.dev.java.net/

Zusätzlich zu den bisher vorgestellten Klassen wurden noch weitere Klassen, die erst in den folgenden Abschnitten erläutert werden, in das Diagramm aufgenommen, um eine Gesamtübersicht über das System geben zu können.

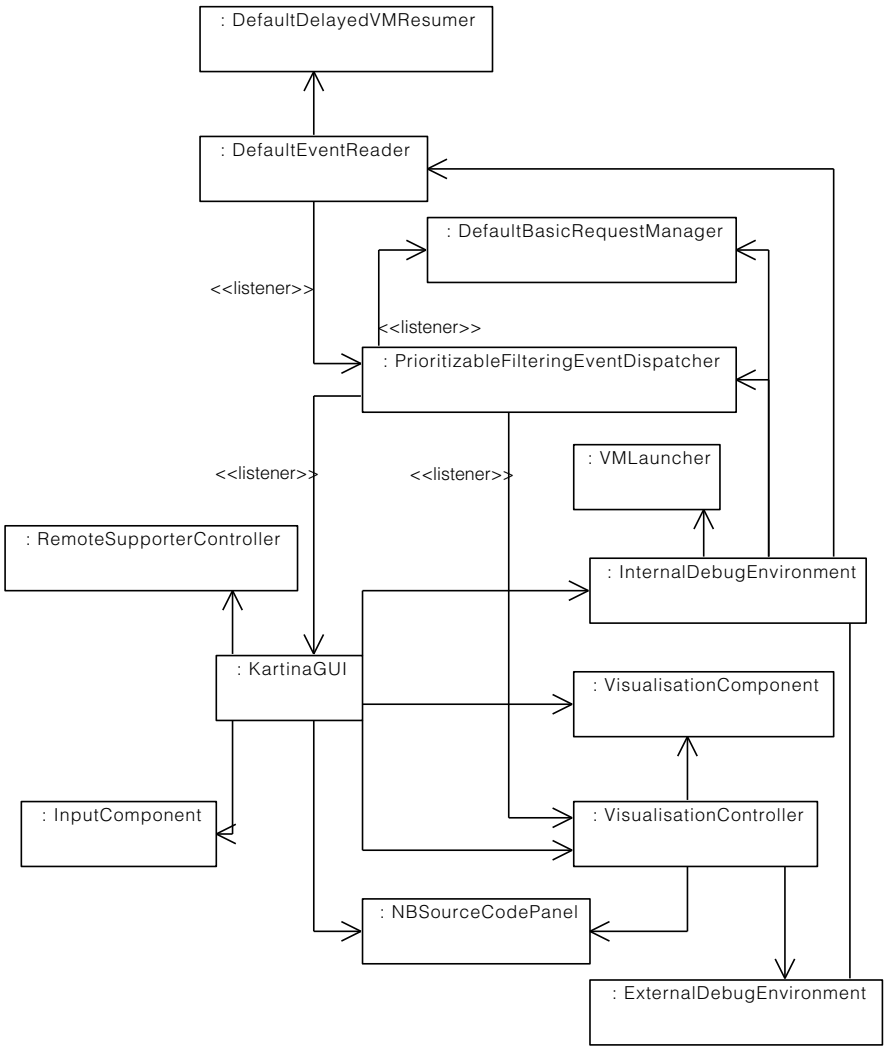

Abbildung 6: Abhängigen zur Laufzeit

In Abb. 7 zeigen die mit «`listener`» stereotypisierten Assoziationen an, dass das Objekt, bei dem die Assoziation beginnt, am anderen Objekt als Listener registriert ist. In Abb. 8 bedeutet der «`listener`»-Stereotyp, dass ein in einem vorherigen Schritt vorkommendes Ereignis mittels Eventdispatching weitergeleitet wird – z.B. in Schritt 2 der ClassPrepareEvent und in Schritt 7 der RemoteSupporterIsReadyEvent.

«`listener`»

Die Schritte 5 und 7 in Abb. 7, die jeweils mit „Erzeugen und registrieren“beschrieben sind, umfassen das Instanziieren und Initialisieren der beteiligten Klassen sowie die Registrierung der neuen Objekte am zuständigen EventDispatcher.

Erzeugen und registrieren

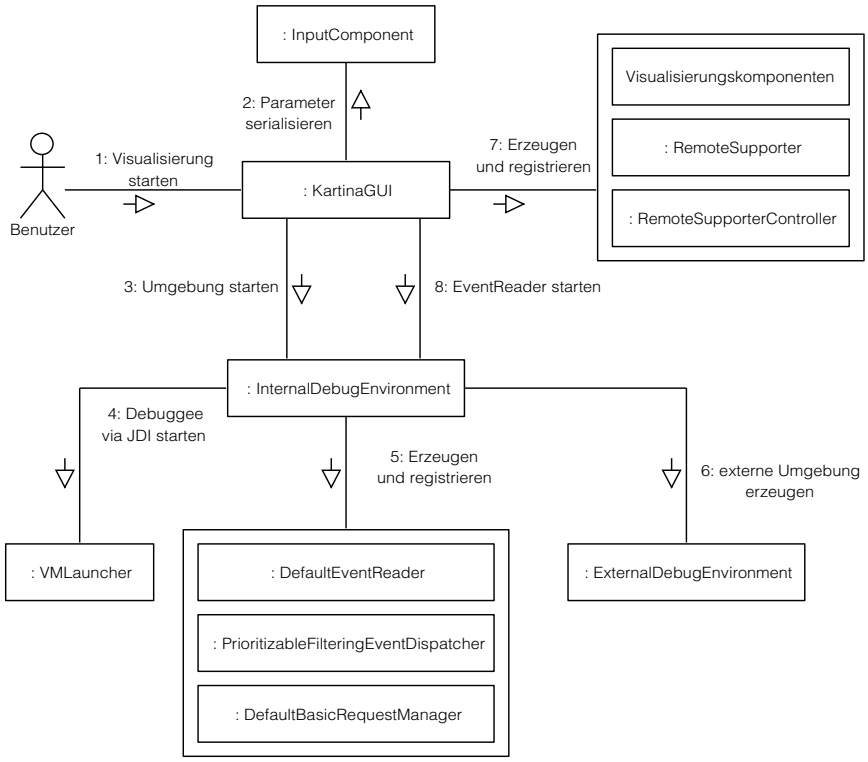

Abbildung 7: Visualisierungsumgebung starten

Das doppelte Vorkommen von Schnitt 12 in Abb. 8 ist kein Fehler, wie sich leicht vermuten ließe, sondern spiegelt den Umstand wieder, dass die Reihenfolge, in der die Listener vom PrioritizableFilteringEventDispatcher benachrichtigt werden, nicht definiert ist, falls die Listener dieselbe Priorität haben.

Im Anschluss an die in Abb. 8 dargestellten Abläufe folgt die eigentliche Visualisierung, d.h. die weitere Kommunikation wird durch den Visualisierungscontroller bestimmt[15].

15 Ausgenommen hiervon ist die Interaktion des Benutzers mit der Visualisierungssteuerung, denn natürlich zieht beispielsweise das Abbrechen der Visualisierung auch wieder interne Kommunikation nach sich.

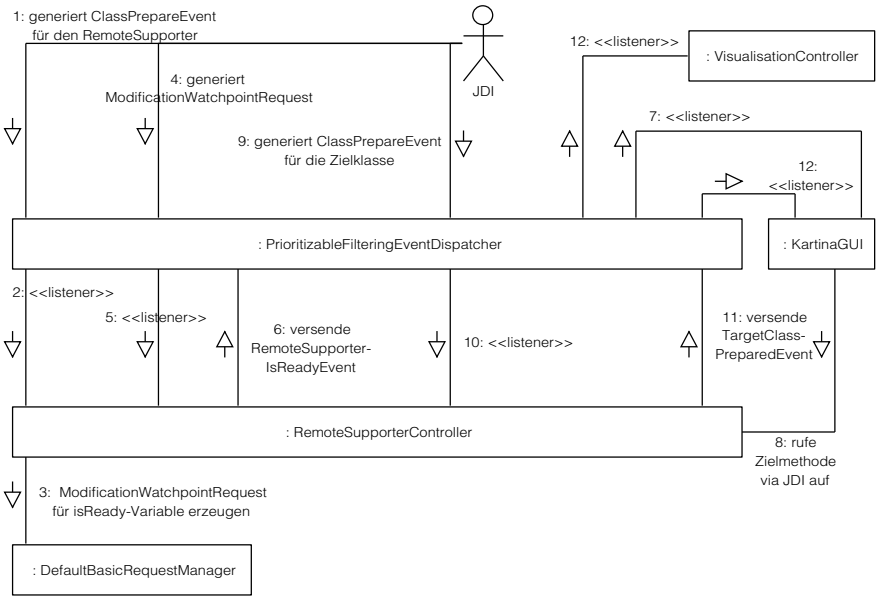

Abbildung 8: Zielmethode ausführen

5.4.9 *DelayedVMResumer*

Wie in Kap. 5.1.3 aufgezeigt wurde, wird die virtuelle Maschine des Debuggees angehalten, wenn ein Ereignis auftritt, um am Ereignis interessierten Objekten die Möglichkeit zu geben, auf dieses zu reagieren.

naives Fortführen des Debuggees birgt Synchronisationsprobleme

Wurde die virtuelle Maschine des Debuggees angehalten, so kann sie über `VirtualMachine.resume()` fortgesetzt werden. Dafür muss es eine eindeutig zuständige Instanz geben, die diesen Aufruf durchführt. Würde es jedem Listener freigestellt, den Aufruf durchzuführen, so könnte die VM unter Umständen gar nicht fortgesetzt werden. Sie könnte aber auch fortgeführt werden, bevor alle Listener über das Ereignis informiert wurden und somit die Gelegenheit hatten, auf es zu reagieren.

Der EventReader könnte den Aufruf tätigen, nachdem er alle seine Listener über das Ereignis informiert hat. Da er der oberste Dispatcher ist und die Ereignisverbreitung synchron erfolgt, ist sichergestellt, dass wirklich alle Listener auf das Ereignis reagieren konnten. Über eine parametrierbaren Verzögerungsfunktion ließe sich auch die Ablaufgeschwindigkeit der Visualisierung einstellen.

Swing bedingt threadsichere Instanz

Leider kommt es bei diesem Vorgehen zu Problemen mit der Swing-Visualisierung, da Swing über einen eigenen Thread verfügt[16]. Bei der Programmierung von Swing-Oberflächen muss die Ausführung von Aktionen mitunter verzögert werden[17], damit sichergestellt ist, dass vorherige Aktionen, die die Oberfläche verändern, bereits ausgeführt wurden. Setzt nun der EventReader die Ausführung des Debuggees fort, so versuchen durch Swing verzögert ausgeführte Aktionen unter Umständen auf Informationen zuzugreifen, die im Debuggee bereits nicht mehr gültig sind.

16 http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html
17 http://java.sun.com/javase/6/docs/api/javax/swing/SwingUtilities.html, Methode invokeLater

Eine Lösung hierfür bietet der Einsatz einer threadsicheren Instanz, die durch den Einsatz von Sperren dafür sorgt, dass die virtuelle Maschine erst dann fortgesetzt wird, wenn sämtliche Zugriffe auf sie abgeschlossen wurden.

Dieses Vorgehen wurde in der Schnittstelle `kartina.debugger.events.DelayedVMResumer` spezifiziert und in der Klasse `DefaultDelayedVMResumer` des selben Pakets umgesetzt. Von letzterem existiert genau ein Objekt, auf das die Listener über die DebuggerEnvironment zugreifen können.

Zugriffskontrolle über Sperren

Jeder Listener kann nun Sperren hinzufügen bzw. entfernen sowie die virtuelle Maschine fortführen. Letzteres wird jedoch nur dann sofort ausgeführt, wenn keine Sperren mehr vorliegen. Andernfalls merkt sich der VMResumer, dass die virtuelle Maschine fortgeführt werden soll, tut dies jedoch erst, wenn die Anzahl der Sperren auf `0` gesenkt wird.

Expliziter formuliert wird der Debuggee fortgeführt, wenn eine der beiden folgenden Situationen eintritt:

A. die Anzahl der Sperren beträgt `0` und `DelayedVMResumer.resume()` wird aufgerufen

B. `DelayedVMResumer.resume()` wurde aufgerufen, als die Anzahl der Sperren noch größer als `0` war und ein Aufruf von `DelayedVMResumer.removeLock()` senkt diese auf `0`

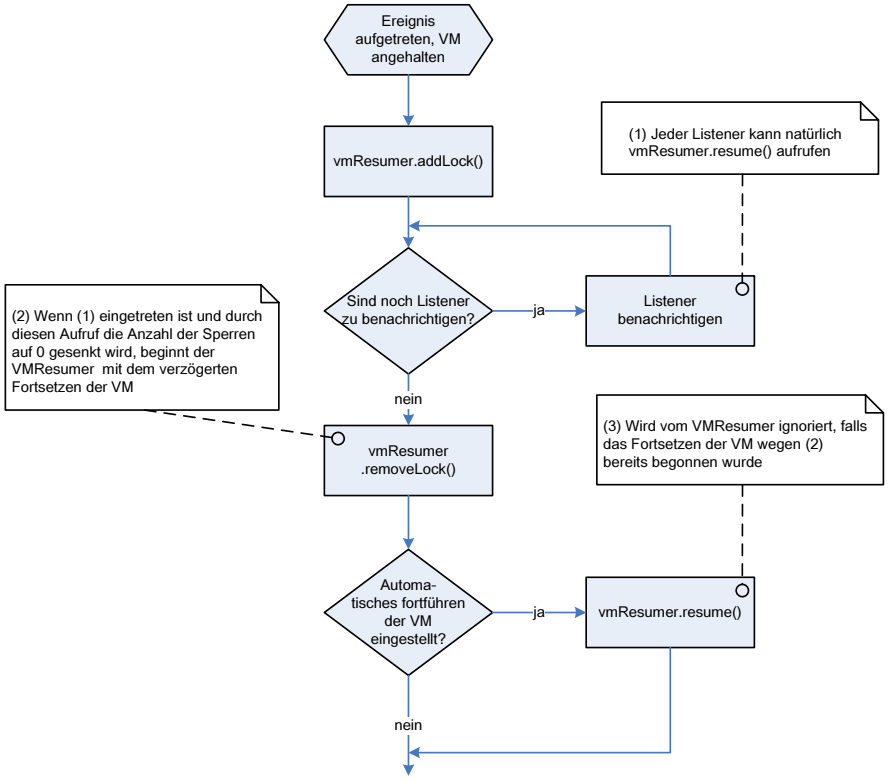

Abbildung 9: Zusammenspiel zwischen DefaultEventReader und DefaultDelayedVMResumer

Dem VMResumer kann zudem eine Zeitdauer mitgegeben werden, um die das Fortsetzen der Ausführung der virtuellen Maschine verzögert wird, nachdem die Anzahl der Sperren auf `0` gesenkt wurde. Der DefaultDelayedVMResumer verfügt dazu über eine private innere Klasse, die von Thread abgeleitet ist und die die virtuelle Maschine des Debuggees erst nach einer durch `Thread.sleep()` erreichten Verzögerung wieder fortsetzt.

zeitverzögertes Fortführen

Wie bereits in Kap. 5.4.2 erwähnt, kann der EventReader angewiesen werden, das Fortsetzen der virtuellen Maschine anzustoßen, wenn er alle seine Listener informiert hat. Dabei wird wie in Abb. 9 dargestellt vorgegangen.

Das Zusammenspiel von EventReader und DelayedVMResumer ermöglicht die in Punkt 7 der Anforderungserhebung genannten Funktionalitäten, z.B. das schrittweise Ausführen des Programms. Für die Steuerung des Ablaufs stehen die in Abb. 10 und Abb. 11 dargestellten Steuerelemente zur Verfügung. Die Schaltfläche „nächster Schritt“ bewirkt dabei, dass der Debuggee bis zum nächsten auftretenden Ereignis ausgeführt wird.

Abbildung 10: Steuerelemente vor dem Start des Debuggees: „nächster Schritt“,„Start“ und „Ablaufgeschwindigkeit“

Abbildung 11: Steuerelemente zur Laufzeit: „Pause“,„Stop“ und „Ablaufgeschwindigkeit“

5.4.10 *Debuggee und RemoteSupporter*

Einschränkung auf `main`-Klassen

Die in Kap. 5.4.1 genannte Einschränkung, dass das JDI nur `main`-Klassen starten kann, hat bei näherer Betrachtung die im Folgenden beschriebenen Nachteile.

Zwang zur `main`-Methode verletzt Anforderung und ist unnötig

Der Algorithmenimplementierer muss wissen, dass der zu implementierende Algorithmus später visualisiert werden muss. Dies steht nicht nur im Widerspruch zu Punkt 2 der Anforderungserhebung für Kartina, sondern führt auch dazu, dass so eine Algorithmenbibliothek entsteht, in der jede Klasse ausführbar ist, obwohl dies aus architektonischer Sicht nicht erforderlich ist.

Ein Punkt der Anforderungserhebung war die Möglichkeit, den zu visualisierenden Algorithmus auf unterschiedlichen Datensätzen operieren zu lassen. Zur visuellen Analyse eines Sortieralgorithmus ist mindestens das Sortieren einer unsortierten und je einer bereits auf- bzw. absteigend sortierten Liste hilfreich. Soll auch die Stabilität des Sortieralgorithmus untersucht werden, kann das Sortieren einer weiteren unsortierten Liste nötig sein.

Da das JDI die Ausführung auf `main`-Klassen beschränkt, könnte eine zu visualisierende Implementierung eines Sortieralgorithmus in etwa wie folgt aussehen:

```
1  public class SortieralgorithmusMitKommandozeile {
2     public void sort(int[] array) {
3        /* Array sortieren. */
4     }
5
6     public static void main(String[] args) {
7        int[] array
8              = konvertiereKommandozeilenparameter(args);
9
10       new SortieralgorithmusMitKommandozeile()
11             .sortiere(array);
12    }
13 }
```

Listing 6: Parameterübergabe über die Kommandozeile

Bei einem Sortieralgorithmus der auf einer Zahlenfolge operiert, ist obige Implementierung mit relativ geringem Aufwand zu erreichen. Da die Zahlenfolge als Kommandozeilenparameter übergeben wird, muss allerdings

1. ein Format definiert werden, in dem der Parameter angegeben wird, z.B. `1,4,7,44,3,0,5`.
2. eine Konvertierungsroutine für dieses Format geschrieben werden.
3. der Vorführer mit diesem Format vertraut sein.

Operationsdaten sind u.U. komplex und zahlreich

Bei der Visualisierung von Algorithmen, die auf komplexeren Datenstrukturen, z.B. Graphen oder Matrizen operieren, gestaltet sich die Übergabe der Parameter über die Kommandozeile weitaus komplizierter.

Ein anderes, auch bei komplexen Datenstrukturen anwendbares Vorgehen wäre das feste Kodieren der Datensätze innerhalb der `main`-Methode:

```
1  public class SortieralgorithmusMitAuswahl {
2     //...
3
4     public static void main(String[] args) {
5        int[][] arrays = new int[][] {
```

```
6           new int[] {6,3,8,9,1,4,5,0},
7           new int[] {6,3,6,9,1,6,5,6},
8           new int[] {1,2,3,4,5},
9           new int[] {5,4,3,2,1}
10       };
11
12       int wahl = Integer.parseInt(args[0]);
13       int[] array = arrays[wahl];
14
15       new SortieralgorithmusMitAuswahl().sortiere(array);
16    }
17 }
```

Listing 7: Parameterauswahl über die Kommandozeile

hartkodierte Daten sind unflexibel

Bei diesem Vorgehen muss der Vorführer nur noch wissen, welche Wahlmöglichkeit zu welchen Daten führt. Es hat allerdings den Nachteil, dass es nicht mehr ohne Neukompilieren des Algorithmus möglich ist, einen zusätzlichen Datensatz zu visualisieren oder einen bestehenden zu modifizieren. Eben dies kann jedoch im Rahmen eines Unterrichts durchaus von Interesse sein, z.B. um auf Fragen aus dem Publikum zu reagieren.

Serialisierung

Eine weitere Möglichkeit, Parameter über die Kommandozeile zu übertragen, bietet die Serialisierung von Objekten in Java. Unter Serialisierung versteht man dabei ganz allgemein das Abbilden von Objekten auf eine sequenzielle Darstellungsform, die z.B. auf der Festplatte abgelegt oder über ein Netzwerk übertragen werden kann [23/Kap. 12]. Da auch ein String als sequenzielle Darstellungsform in Frage kommt, ist es somit möglich, beliebige Objekte über die Kommandozeile zu übertragen. Außerdem liegen dem JDK bereits die Klassen `ObjectOutputStream` und `ObjectInputStream` im Paket `java.io` bei, die eine anwenderfreundliche Objektserialisierung ermöglichen.

Verschiedene Algorithmen

In den vorherigen Beispielen befand sich jeweils nur eine aufzurufende Methode in der Zielklasse. Sind aber z.B. mehrere Sortierverfahren in einer Klasse realisiert, so müsste zusätzlich zur Auswahl der Datensätze noch eine Auswahl der aufzurufenden Methode stattfinden.

Letzteres lässt sich über if-Anweisungen realisieren,

```
1  public class SortieralgorithmenMitSwitch {
2     //...
3
4     public static void main(String[] args) {
5        int[] array = ermittleDatensatz(args);
6        int methode = Integer.parseInt(args[0]);
7
8        SortieralgorithmenMitIfBlock algorithmen
9              = new SortieralgorithmenMitIfBlock();
10
11       switch (methode) {
12          case 0:
13             algorithmen.quicksort(array);
14             break;
```

```
15            case 1:
16               algorithmen.heapsort(array);
17               break;
18            //...
19         }
20      }
21   }
```

Listing 8: Algorithmenauswahl über `switch`

oder – eleganter, da flexibler – über Reflection [23/Kap. 21] lösen

```
1  public class SortieralgorithmenMitReflection {
2     //...
3
4     public static void main(String[] args) {
5        int[] array = ermittleDatensatz(args);
6        int methode = Integer.parseInt(args[0]);
7
8        Class clazz = SortieralgorithmenMitReflection.class;
9
10       /* Die Methode wird über ihren Namen und ihre
11        * Signatur (hier int[]) identifiziert.
12        */
13       Method method = clazz.getDeclaredMethod(
14             args[0], array.getClass());
15
16       Object instance = clazz.newInstance();
17       method.invoke(instance, array);
18    }
19 }
```

Listing 9: Algorithmenauswahl über Reflection

Trägerklasse

Alle bisher vorgestellten Ansätze lösen nicht das Problem (könnten es auch gar nicht), dass sich der zu visualisierende Algorithmus innerhalb einer `main`-Klasse befinden muss, was, wie bereits mehrfach erwähnt, nicht akzeptabel ist.

JDI ermöglicht Objekterzeugung zur Laufzeit, wie Reflection

Da das JDI dem Debugger erlaubt, neue Instanzen innerhalb der VM des Debuggees zu erzeugen[18] sowie Methoden im Kontext der VM aufzurufen[19], ist es möglich, die zu startende `main`-Methode in eine *Trägerklasse* auszulagern, die dann den Debuggee darstellt und vom JDI gestartet wird. Anschließend kann in ihrem Kontext eine Instanz des zu visualisierenden Algorithmus – der Zielklasse – erzeugt und die entsprechende Zielmethode ausgeführt werden. Die eigentlichen Algorithmen kämen somit ohne `main`-Methode aus (siehe Lst. 9).

Leider muss die VM – genauer gesagt der verwendete Classloader – des Debuggees die zu instanziierende Klasse bereits kennen, d.h. die Trägerklas-

18 http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/com/sun/jdi/ClassType.html, Methode `newInstance()`

19 http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/com/sun/jdi/ObjectReference.html, Methode `invokeMethod()`

se muss von jeder zu visualisierenden Klasse bereits eine Instanz erzeugt haben. Erst wenn der Debugger eine JDI-Referenz auf eine Klasse innerhalb der virtuellen Maschine des Debuggees hat, kann er das JDI veranlassen, eine neue Instanz dieser Klasse zu erzeugen.
Eine feste Kodierung aller möglichen Klassen, d.h. aller visualisierbarer Algorithmen, scheidet natürlich wegen mangelnder Flexibilität aus.

Eine Lösung dieses Problems mittels Reflection findet sich in Geiger [15] und wird dort in Abschnitt 3.3.3 in der Realisierung als Klasse `DLauncher` beschrieben.

Funktionsweise der Trägerklasse

Die Idee des dort in Abb. 13 als Sequenzdiagramm dargestellten Ablaufs lässt sich wie folgt beschreiben

1. Die Trägerklasse wird als Debuggee mittels des JDI gestartet.
2. Die Ausführung der Trägerklasse wird an einer bestimmten Stelle unterbrochen (in Abb. 13 mittels eines Breakpoints).
3. Der Debugger führt über das JDI eine Methode des Debuggees (also der Trägerklasse) aus und teilt ihr so mit, welche Klasse zu instanziieren ist.
4. Die Ausführung des Debuggees wird fortgesetzt. Dieser erzeugt nun mittels Reflection eine neue Instanz der angegebenen Klasse.
5. Jetzt kann auch der Debugger mittels JDI mit der nun der VM des Debuggees bekannten Klasse arbeiten.

RemoteSupporter

Die in [15] aufgezeigte Lösung wurde für Kartina adaptiert und nur leicht verändert.

Variable statt Zeilennummer als Unterbrechungspunkt

- Die Ausführung der Trägerklasse wird durch das Setzen einer boolesche Variablen auf `true`, also durch einen `ModificationWatchpointEvent`, unterbrochen.

  Dies hat gegenüber einem Breakpoint den Vorteil, dass Änderungen am Quelltext der Trägerklasse, die ein zeilenbezogenes Verschieben des Unterbrechungspunktes bewirken, keine Änderungen an der Spezifikation des Unterbrechungspunktes, das heißt des `ModificationWatchpointRequests` nach sich ziehen.

  Voraussetzung für einen erfolgreichen Ablauf ist hierbei natürlich, dass die beobachtete boolesche Variable an keiner anderen Stelle der Trägerklasse auf `true` gesetzt wird.

Trägerklasse übernimmt mehr Aufgaben

- In Kartina werden der Trägerklasse nicht nur die zu instanziierende Klasse mitgeteilt, sondern direkt auch die aufzurufende Methode und deren Parameter.

  Wird die Ausführung der Trägerklasse wieder aufgenommen, so erzeugt diese umgehend eine neue Instanz der ihr mitgeteilten Klasse, in deren Kontext sie dann die Zielmethode ausführt.

  Der Debugger muss daher nicht selber via JDI für die Instanziierung der Zielklasse und die Ausführung der gewünschten Methode sorgen. Er kann natürlich, falls nötig, trotzdem über das JDI darauf zugreifen,

so dass diese Änderung keine Einschränkung der Allgemeinheit bewirkt.

In Kartina wurde dieses Vorgehen in den Klassen `RemoteSupporter` und `RemoteSupporterController` implementiert. Der RemoteSupporter ist dabei die über eine `main`-Methode verfügende Trägerklasse, die die Zielmethode entgegen nimmt und diese anschließend mittels Reflection ausführt.

Der RemoteSupporterController übernimmt die Steuerung des Ablaufs. Er erstellt den `ModificationWatchpointRequest` für die zu überwachende boolesche Variable und überträgt zu gegebener Zeit die notwendigen Informationen für die Ausführung der Zielmethode via JDI.

Debuggee muss zu instanziierende Klasse bereits kennen

Leider gilt bei der Ausführung einer Methode mittels JDI eine Einschränkung ähnlich derjenigen, die bereits in Kap. 5.4.10 für das Erzeugen von Klassen beschrieben wurde: die Parameter der auszuführenden Methode müssen bereits in der VM des Debuggees vorhanden sein.

Angenommen, es müsse eine Methode aufgerufen werden, die als Parameter eine HashMap erwartet. Dann

A. muss die als Parameter zu verwendende HashMap bereits vom Debuggee erstellt worden sein

B. oder es wird eine neue, als Parameter zu verwendende HashMap via JDI erzeugt. Hierfür muss jedoch der Classloader der VM des Debuggees die Klasse HashMap bereits kennen, so dass man wieder beim Ausgangsproblem angelangt ist.

serialisiert lassen sich beliebige Objekte übertragen

Eine rettende Ausnahme bilden die primitiven Datentypen von Java (boolean, int, double, etc., aber nicht Boolean, Integer, Double, etc.) sowie Strings, für die mittels `com.sun.jdi.VirtualMachine.mirrorOf()` eine Kopie des Werts einer im Debugger existierenden Variable gemacht werden kann.

Zielklasse und -methode können als Namen, also als Strings, übertragen werden. Für die Übertragung der Parameter der Zielmethode nutzt Kartina die eingangs bereits angesprochene Objektserialisierung. Die Objekte, die der Zielmethode als Parameter zu übergeben sind, werden auf der Seite des Debuggers in einen String serialisiert, danach via JDI an den RemoteSupporter übertragen und dort abschließend wieder deserialisiert.

Zusätzlich zur bereits beschriebenen Kommunikation mit dem RemoteSupporter erzeugt der RemoteSupporterController einen `RemoteSupporterIsReadyEvent`, der ankündigt, dass der RemoteSupporter nun Zielmethoden entgegennimmt, sowie den `TargetClassPrepareEvent`, der angibt, dass der `ClassPrepareEvent` für die Zielklasse aufgetreten ist. Da diese Ereignisse über den normalen Kommunikationskanal verschickt werden, d.h. mittels EventDispatcher, ist es für interessierte Klassen möglich, auf sie zu reagieren.

Wie bereits im vorherigen Absatz angedeutet, ist der RemoteSupporter in der Lage, mehrere Zielmethoden hintereinander entgegen zu nehmen. Wird die Ausführung des RemoteSupporters fortgesetzt, so werden die entgegengenommenen Zielmethoden sequenziell ausgeführt. Von dieser Eigenschaft wird aber im Rahmen von Kartina bisher kein Gebrauch gemacht.

Das folgende Sequenzdiagramm ist an Abb. 13 aus [15] angelehnt und stellt die Idee hinter der Realisierung dar. Während sich Abb. 8 auf das Zusammenspiel der beteiligten Klassen konzentriert, wie es letztendlich in Kartina stattfindet, stellt das Sequenzdiagramm die Idee und somit die Interaktion zwischen RemoteSupporter und RemoteSupporterController in den

Vordergrund. Daher fehlt in diesem Diagramm z.B. ein Dispatcher, der für den Versand der Nachrichten zuständig ist.

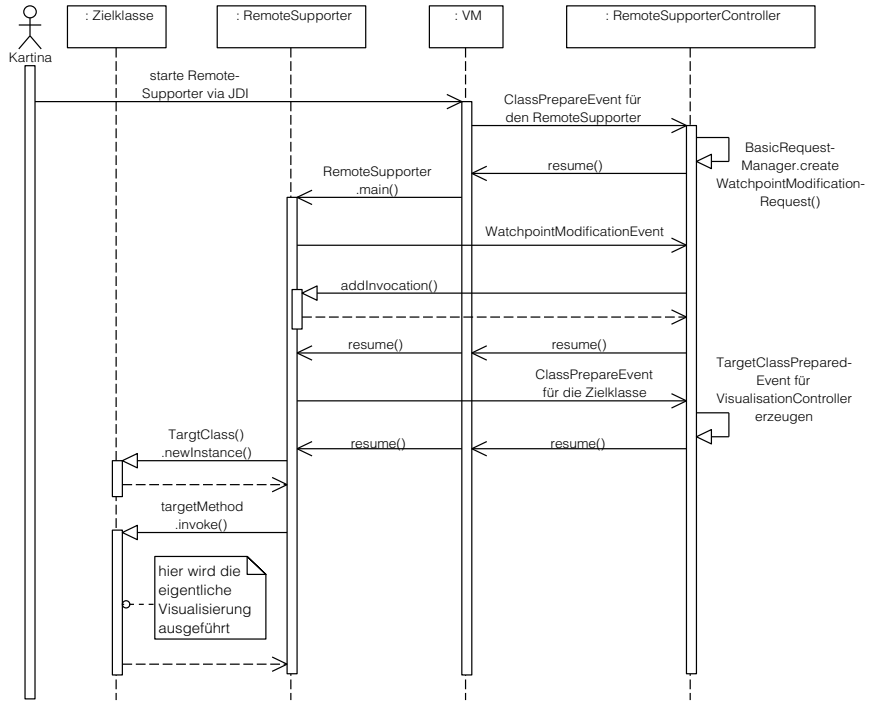

Abbildung 12: Einsatz des RemoteSupporters

Einschränkungen

Die derzeitige Implementierung von RemoteSupporter und RemoteSupporterController hat die folgenden Einschränkungen, die jedoch – falls überhaupt möglich – ohne strukturelle Änderungen aufhebbar sind.

- Die Methode des RemoteSupporters, die die Zielklasse etc. entgegennimmt, muss allein durch den Namen eindeutig identifizierbar sein.

  Um eine JDI-Referenz (`com.sun.jdi.Method`) auf diese Methode zu erhalten, wird der erste Eintrag aus der Liste aller zum Namen passenden, sichtbaren Methoden der Zielklasse genommen.

  Abhilfe können hier die Methoden `com.sun.jdi.ReferenceType .methodsByName()` sowie `com.sun.jdi.ClassType.concreteMethodByName()` schaffen, die neben dem Namen auch die JNI-Signatur[20] der gesuchten Methode beachten.

- Die Zielklasse muss – wie bei einer JavaBean – über einen Standardkonstruktor, d.h. über einen parameterlosen Konstruktor verfügen.

  Zielklasse benötigt Standardkonstruktur

  Abhilfe würde hier das Übertragen der Konstruktorparameter analog zum Übertragen der Zielmethodenparameter schaffen.

20 http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/types.html#wp16432

- Es können nur öffentliche Zielmethoden (auch geerbte) ausgeführt werden.

  Die via Reflection auszuführende Zielmethode wird über `java.lang.Class.getMethod()` ermittelt und mittels `java.lang.reflect.Method.invoke()` ausgeführt.

  Alternativ können auch die Methoden `Class.getDeclaredMethod()` und `Method.setAccessible()` dazu genutzt werden, auch private Methoden auszuführen.

schwerwiegend: keine Interfaces als Parameter

- Die Zielmethode darf keine Interfaces als Parametertypen haben.

  `Class.getMethod()` wird verwendet, um eine Reflection-Referenz auf die Zielmethode zu bekommen. Das Array mit den Typen (`Class<?>... parameterTypes`), die die Methode als Parameter akzeptiert, wird mit den Typen der deserialisierten Parameter erstellt. Da dies immer konkrete Klassen, keine Schnittstellen sind (z.B. `java.util.ArrayList` statt `java.util.List`), liefert `Class.getMethod()` auch nur Methoden zurück, deren Signatur die konkreten Klassen enthält.

  Abhilfe würde hier das Ermitteln der von einer konkreten Klasse implementierten Schnittstellen schaffen, die dann wiederum für die Methodensuche mittels `Class.getMethod()` verwendet werden müssten. Bei einer Weiterentwicklung von Kartina sollten dieser sowie der nächste Punkt höchste Priorität bekommen, da sie im krassen Widerspruch zu Punkt 2 der Anforderungserhebung stehen.

schwerwiegend: keine Primitiven als Parameter

- Primitive Datentypen und Autoboxing [23] werden nicht unterstützt.

  Eine Folge des im vorherigen Punkt beschriebenen Verhaltens von `Class.getMethod()` ist, dass eine Methode mit einem `double` in der Signatur nicht gefunden wird, wenn ein `java.lang.Double` deserialisiert wurde.

  Da z.Z. ausschließlich `java.io.ObjectInputStream.readObject()` zur Deserialisierung verwendet wird, muss auch die entsprechende Methode `ObjectInputStream.writeObject()` zur vorangehenden Serialisierung genutzt werden (wobei primitive Datentypen dank Autoboxing in ein entsprechendes Objekt gehüllt werden). Demzufolge können zur Zeit keine Methoden aufgerufen werden, die primitive Datentypen als Parameter erwarten. Arrays, z.B. `int[]`, werden hingegen unterstützt, da Arrays wie Objekte behandelt werden.

  Diese Einschränkung, die im krassen Widerspruch zu Punkt 2 der Anforderungserhebung steht, lässt sich ähnlich wie die Einschränkung auf konkrete Klassen lösen, indem auch die zu einem Objekt passenden primitiven Datentypen in die Methodensuche einbezogen werden.

5.5 Visualisierung mit Swing

In der im Rahmen dieser Bachelorthesis umgesetzten Version von Kartina erfolgt die letztendliche Visualisierung eines Algorithmus mit Swing-Komponenten. Dies hat den Vorteil, dass auf eine erprobte Technologie zurückgegriffen werden kann, für die eine Vielzahl frei erhältlicher Komponenten verfügbar ist.

Für die Visualisierung notwendig sind hierbei (neben dem Algorithmus selbst) folgende drei Klassen:

1. ein *Visualisierungscontroller* zur Steuerung der Visualisierung, der die Schnittstelle `kartina.visualisations.VisualisationController` implementieren muss.
2. eine von `JPanel` abgeleitete *Visualisierungskomponente*, die vom Controller gesteuert wird.
3. eine *Eingabekomponente* als Eingabemöglichkeit für die Parameter der Zielmethode. Sie muss von `javax.swing.JPanel` abgeleitet sein und `kartina.gui.InputComponent` implementieren.

Visualisierungscontroller

Der Visualisierungscontroller ist typischerweise für das Erstellen von Requests und das Reagieren auf auftretende Events zuständig, entsprechend derer er die Visualisierungskomponente modifiziert. In der Regel wird ein Controller für einen bestimmten Algorithmus, maximal für eine bestimmte Klasse von Algorithmen erstellt werden.

Von Kartina wird der Controller ausschließlich für TargetClassPrepareEvents am Dispatcher registriert. Es liegt in der Zuständigkeit des Kontrollvisualisierers, dafür zu sorgen, dass der Controller auch über andere Ereignisse informiert wird. Auch das Hervorheben der aktuell ausgeführten Zeile im Quelltext der Zielklasse (siehe Kap. 5.7) muss vom Controller angestoßen werden.

Visualisierungskomponente

Die Visualisierungskomponente ist für die letztendliche Darstellung des ablaufenden Algorithmus zuständig. Eine Visualisierungskomponente kann für mehrere Algorithmen eingesetzt werden, z.B. für verschiedene Sortierverfahren, es kann jedoch auch ein Algorithmus durch mehrere Visualisierungskomponenten dargestellt werden, z.B. ein Sortierverfahren als Liste von Zahlen oder als Reihe von Balken.

Eingabekomponente

Die Form der Parametereingabe, die innerhalb der Eingabekomponente erfolgt, kann jeder Visualisierer frei gestalten. Es kann sich z.B. um ein einfaches Texteingabefeld handeln, um Buttons, die zufällige oder sortierte Zahlenfolgen generieren oder um ein komplexes Werkzeug zur Erstellung von Graphen. Einzig relevant ist, dass die Eingabekomponente die als Parameter für die Zielmethode zu verwendenden Objekte in einen `ObjectOutputStream` schreiben kann.

Komponenten müssen Standardkonstruktor besitzen

Da die drei Komponenten mittels Reflection instanziiert werden und Kartina bisher keine Parameter für die Konstruktoren vorsieht, müssen die Komponenten (analog zur Zielklasse) über einen Standardkonstruktor verfügen.

Mit der Aufteilung der visualisierungsbezogenen Komponenten gemäß des MVC-Entwurfsmusters in eine View und einen Controller wird u.a. die Nutzung ein- und derselben View für mehrere Algorithmenvisualisierungen ermöglicht, z.B. eine View für mehrere Sortieralgorithmen. Ebenso ist es möglich, einen Controller für unterschiedliche Views zu nutzen, z.B. um den Ablauf eines Sortieralgorithmus als eine Reihe von Zahlen oder Balken darzustellen.

5.6 Konfiguration

Konfiguration im Rahmen von Kartina bezieht sich hauptsächlich auf die Auswahl des zu visualisierenden Algorithmus und die Angabe der dafür benötigten Komponenten.

Die Konfiguration erfolgt über eine XML Datei, die der Benutzer einer DTD[21] (siehe Lst. 11) entsprechend editieren kann und die Kartina als Kommandozeilenparameter übergeben werden kann. Der im folgenden aufgeführte beispielhafte Auszug aus einer solchen Konfigurationsdatei spiegelt dabei die Struktur der Java-Implementierung eines Algorithmus wieder:

```
<kartina>
   <excludes>java.*, javax.*, sun.*, com.sun.*</excludes>
   <files>
      <file>
         <name>Bubble Sort</name>
         <desc>Bubblesort or Selectionsort</desc>
         <source>.\bubbleSort\Algorithm.java</source>
         <class>bubbleSort.Algorithm</class>
         <methods>
            <method>
               <name>sortDescending</name>
               <inputComponent>
                  gui.DefaultArgumentsInputPanel
               </inputComponent>
               <visualisationComponent>
                  bubbleSort.Visualisation
               </visualisationComponent>
               <visualisationController>
                  bubbleSort.Controller
               </visualisationController>
            </method>
         </methods>
      </file>
   </files>
</kartina>
```

Listing 10: Konfigurationsdatei von Kartina

Eine Konfigurationsdatei kann mehrere Klassen (`file`-Knoten) beinhalten, die wiederum mehrere visualisierbare Methoden (`method`-Knoten) umfassen können.

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT kartina (name, desc, classpath?, excludes?, files)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT desc (#PCDATA)>
<!ELEMENT excludes (#PCDATA)>
<!ELEMENT classpath (#PCDATA)>
<!ELEMENT files (file*)>
<!ELEMENT file (name, desc, source, class, classpath?, excludes
    ?, methods)>
```

21 http://de.wikipedia.org/wiki/Dokumenttypdefinition

```
10 <!ELEMENT source (#PCDATA)>
11 <!ELEMENT class (#PCDATA)>
12 <!ELEMENT methods (method*)>
13 <!ELEMENT method (name, desc, classpath?, excludes?,
       inputComponent, visualisationComponent,
       visualisationController)>
14 <!ELEMENT inputComponent (#PCDATA)>
15 <!ELEMENT visualisationComponent (#PCDATA)>
16 <!ELEMENT visualisationController (#PCDATA)>
```

Listing 11: DTD der Konfigurationsdatei

Classpath und Ausschließungen werden vererbt

Die drei Hierarchiestufen `kartina`, `file` und `method` können jeweils mit einem Namen, einer Beschreibung, einem Classpath und einer Liste mit von der Ereignisgenerierung auszuschließenden Klassen versehen werden. Der Classpath und die Liste auszuschließender Klassen werden dabei an die tieferen Hierarchiestufen vererbt. Für das obige Beispiel bedeutet dies, dass die Klassen im Paket `com.sun` immer von der Ereignisgenerierung ausgeschlossen werden, egal welche Methode ausgeführt wird.

Ausschluss-Syntax

Die Syntax, in der die auszuschließenden Klassen angegeben werden, entspricht der vom JDI erwarteten[22] Syntax. Anfängliche Überlegungen, die Syntax auf die bereits etablierte, von Ant[23] genutzte Syntax zur Angabe von Verzeichnissen[24] umzustellen, wurden wieder verworfen, da die Ant-Syntax eine weitaus feinere Granularität ermöglicht und eine Konvertierung von der Ant- in die JDI-Syntax viel Aufwand bei wenig Nutzen bedeuten würde. Zudem ist die simple, da ausdrucksschwache JDI-Syntax einfach zu erlernen.

Classpath-Syntax

Die Syntax des Classpaths entspricht der Syntax, in der der Classpath auch beim Aufruf des Java-Interpreters (`java`) über die Kommandozeile übergeben wird.

Die in Kap. 5.5 eingeführten Klassen Visualisierungscontroller, Visualisierungskomponente und Eingabekomponente, die eine Visualisierung umfasst, werden dem Methodenknoten mitgegeben.

Die Umsetzung der XML-Struktur in Java-Objekte erfolgt händisch unter Benutzung des `javax.xml.parsers.DocumentBuilders`, der das Arbeiten mit dem DOM[25]-Baum einer XML-Datei ermöglicht. Händisch bedeutet hierbei, dass Werkzeuge zur automatischen Generierung von Java-Objekten aus XML-Strukturen, z.B. JAXB[26], genutzt werden. Dies hat folgende Gründe:

- Die Struktur der Konfigurationsdatei ist einfach und eine händische Umsetzung in Objekte daher mit wenig Aufwand verbunden.
- JAXB ist eine sehr mächtige Schnittstelle, die jedoch eine steile Lernkurve aufweist, die nicht im Verhältnis zum erzielten Nutzen steht.
- Von einer Nutzung von Werkzeugen Dritter, wie JiBX[27] oder JMX[28], ist abzuraten, da ihre Weiterentwicklung durch die Verabschiedung von JAXB in Frage gestellt wurde.

22 http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/com/sun/jdi/request/StepRequest.html, Methode addClassExclusionFilter
23 http://ant.apache.org/
24 http://ant.apache.org/manual/dirtasks.html#patterns
25 http://www.w3.org/DOM/
26 http://java.sun.com/developer/technicalArticles/WebServices/jaxb/
27 http://jibx.sourceforge.net/
28 http://jxm.sourceforge.net/

Der Vorführer kann die über die Konfigurationsdatei zur Verfügung gestellten Visualisierungen anschließend über eine zweistufige Liste auswählen (siehe Abb. 13), deren Icons ebenfalls der Netbeans-IDE entnommen wurden.

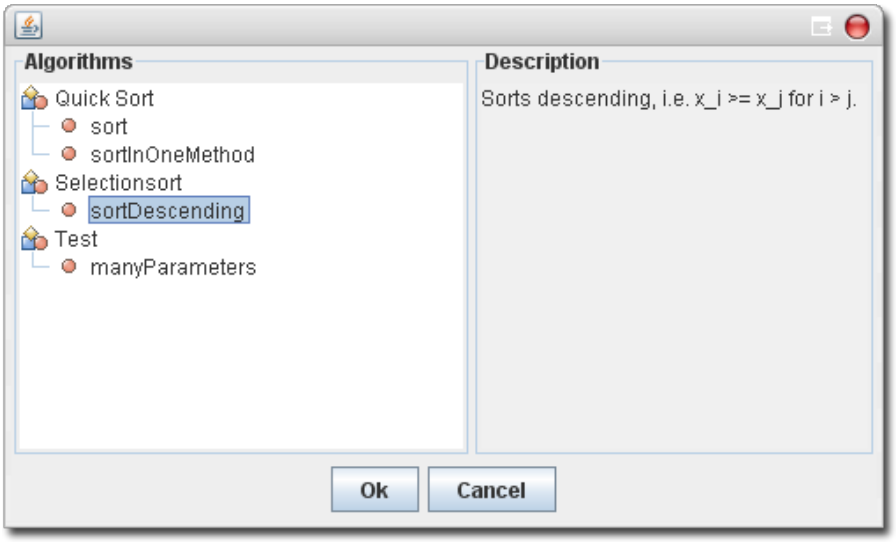

Abbildung 13: Visualisierungsauswahl

Ausblick: Eingabekomponente parametrierbar machen

Eine sinnvolle Erweiterung der zur Zeit vorhandenen Struktur der Konfigurationsdatei wäre die Möglichkeit, der Eingabekomponente beliebige Parameter, evtl. als eigener XML-Block, mitzugeben (siehe auch Kap. 6.1.1). Bei der Übergabe der zusätzlichen Parameter als XML-Struktur kann es unter Umständen zu Problemen beim Parsen der Konfigurationsdatei kommen, da deren DTD eindeutig festlegt, welche Elemente in der Konfigurationsdatei vorkommen dürfen. Da die Struktur der Parameterdaten vom Benutzer frei wählbar sein soll, kann die DTD nicht einfach um zusätzliche Elementdefinitionen erweitert werden.

5.7 Netbeans IDE Codeeditor

Zum Anzeigen des Quelltextes der visualisierten Klasse wird der Editor der Netbeans IDE 5.5 verwendet (siehe Abb. 14). Dies hat den Vorteil, dass sämtliche Funktionen dieses sehr mächtigen Editors genutzt werden können, auch wenn im Rahmen von Kartina zur Zeit nur das Syntaxhighlighting und das Hervorheben der aktuell ausgeführten Zeile genutzt werden. Denkbar wäre jedoch auch die Nutzung der linken Randspalte des Editors, die in Netbeans z.B. für das Setzen von Breakpoints und die Anzeige von Hinweisen bzw. Fehlern genutzt wird.

Der Quelltext der Netbeans IDE und somit auch der des Editors sind frei[29] verfügbar und können über die Webseite des Netbeans-Projektes bezogen

29 http://www.netbeans.org/cddl-gplv2.html

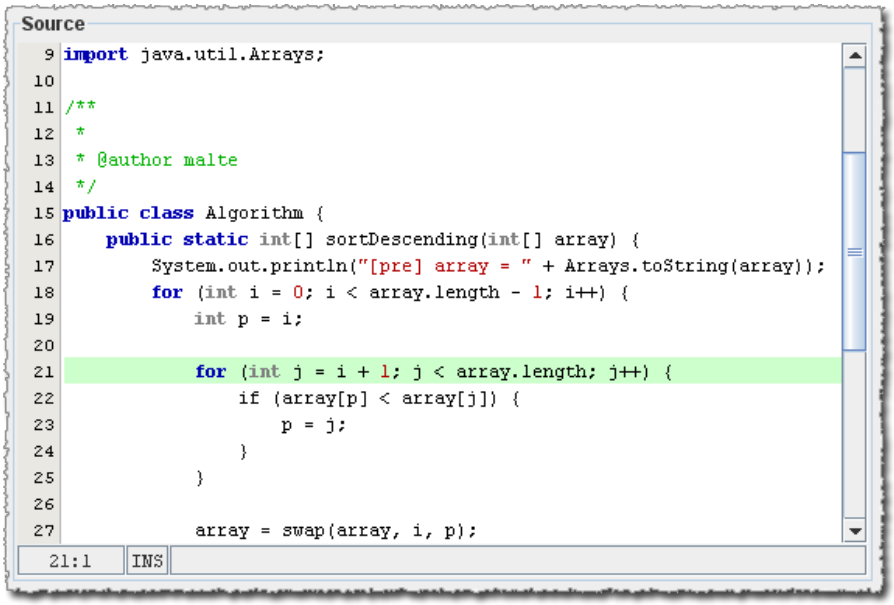

Abbildung 14: Integrierter Netbeans IDE Quelltexteditor

werden. Trotz einer ausführlichen JavaDoc-Dokumentation[30], eines Wikis[31] sowie zahlreicher Mailinglisten[32], gestaltete sich das Integrieren der Editorkomponente nicht einfach. Dies lag jedoch nicht an der API des Editors, sondern daran, dass der Einstieg in den Quelltext einer so umfangreichen Software wie der Netbeans IDE eine große Hürde darstellt.

Eine deutliche Vereinfachung der Integration des Editors in eigene Java-Programme stellt der Artikel „Building a standalone NetBeans editor“dar [24]. Dieser erläutert nicht nur das Vorgehen bei der Integration des Editors, sondern stellt auch den Quelltext einer einfachen Integration des Editors mit Syntaxhighlighting für Java und Scheme[33] zur Verfügung. Der von Vieiro erstellte Quelltext wird daher auch in nur minimal abgeänderter Form von Kartina genutzt.

Entwicklern mit Interesse am Quelltext der Netbeans IDE sei das Maßstäbe setzende Weblog von Geertjan[34] empfohlen, in dem es neben vielen weiteren Artikeln über die Erweiterung der Netbeans IDE einen Artikel [14] über das Erstellen eigener Fehlermarkierungen (im Netbeans-Jargon *Annotations* genannt) für den Netbeans Editor gibt.

Ein weiteres interessantes Projekt in diesem Zusammenhang ist das Schliemann-Projekt[35], dessen Ziel die Unterstützung beliebiger Sprachen durch die Netbeans IDE ist.

30 http://bits.netbeans.org/dev/javadoc/
31 http://wiki.netbeans.org/wiki/
32 http://www.netbeans.org/community/lists/
33 http://de.wikipedia.org/wiki/Scheme
34 http://blogs.sun.com/geertjan/
35 http://wiki.netbeans.org/wiki/view/Schliemann

6 BEISPIELVISUALISIERUNGEN

die Beispielvisualisierungen zeigen Möglichkeiten auf

Die im Rahmen dieser Thesis erstellten Visualisierungen sind eher als Proof of Concept denn als einsatzfertige Visualisierungen zu sehen. Es ist anzuraten, sie im Sinne der Softwareentwicklung als Prototypen zu betrachten und sie im Falle des Anlegens einer eigenen, umfangreichen Algorithmenbibliothek zu refaktorisieren.

Nichtsdestoweniger zeigen die beiden Beispiele auf, mit welchem Aufwand für die Visualisierung von Algorithmen zu rechnen ist, welche Möglichkeiten bereits der Einsatz normaler Swingkomponenten (hier JLabel) bietet und wie durch geschicktes Einsetzen der Requests die entstehende Visualisierung schrittweise verbessert werden kann.

Sie zeigen aber auch, dass die eingangs geforderte strikte Trennung zwischen Algorithmenimplementierer und -visualisierer nicht immer sinnvoll ist bzw. einen Mehraufwand für den Visualisierer nach sich zieht (siehe Kap. 6.3).

Beispielhaft umgesetzt wurden das iterative Sortierverfahren Selectionsort (Sortieren durch Auswählen) und das rekursive Sortierverfahren Quicksort, wobei letzterer in zwei Versionen vorliegt.

6.1 Allgemeine Komponenten

Die hier aufgeführten Komponenten wurden zwar für die Visualisierung der genannten Algorithmen entwickelt, sind aber mindestens auch für die Umsetzung weiterer Sortierverfahren einsetzbar.

6.1.1 *DefaultArgumentsInputPanel*

Das `kartina.gui.DefaultArgumentsInputPanel` ist eine universell einsetzbare Eingabekomponente, die für jeden Parameter der Zielmethode ein Texteingabefeld vorsieht (siehe Abb. 15).

Den Textfeldern kann dabei zur Zeit lediglich die Typenbezeichnung desjenigen Parameters vorangestellt werden, der über das entsprechende Texteingabefeld gesetzt wird, da sich die Namen der Parameter einer Methode in der zur Zeit aktuellen Version 6 des JDK nicht via Reflection ermitteln lassen. Das DefaultArgumentsInputPanel würde daher massiv von der in Kap. 5.6 genannten Möglichkeit, den Eingabekomponenten über die Konfigurationsdatei zusätzliche Informationen zu übergeben, profitieren, da sich so die erwarteten Parameter näher beschreiben ließen.

BeanShell ermöglicht das Erstellen von Objekten über Java-Ausdrücke

Die Ausdrücke, die in die Textfelder eingegeben werden können, werden anschließend vom BeanShell[1] Interpreter ausgewertet. Bei BeanShell handelt es sich um einen schlanken, einbettbaren und frei erhältlichen Java-Interpreter, der die Standardsyntax von Java interpretieren kann und mit weiteren, von dynamischen Sprachen bekannten Konzepten anreichert.

Die projekteigene Webseite beschreibt Beanshell als leichtgewichtigen, freien und einbettbaren Java-Quelltextinterpreter, der Java-Anweisungen dyna-

1 http://www.beanshell.org/

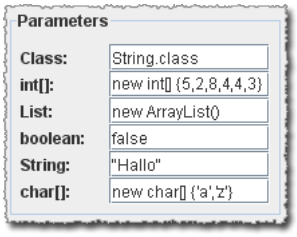

Abbildung 15: DefaultArgumentsInputPanel mit Eingabefeldern für unterschiedliche Typen

misch ausführt und sie um aus Skriptsprachen bekannte Konzepte wie dynamische Typisierung und Closures erweitert (siehe Craig [5]).

Damit wird dem Vorführer eine ebenso einfache wie flexible Komponente zur Parametereingabe an die Hand gegeben, die das Erstellen beliebiger Objekte über die bekannte Java-Syntax ermöglicht.

die Zukunft von BeanShell sieht gut aus

Da Java im Zuge der JSR 223 mit einer Unterstützung für Skriptsprachen ausgestattet wird und BeanShell als eine der zu unterstützenden Sprachen vorgesehen ist, erscheint die Weiterentwicklung von BeanShell als wahrscheinlich, so dass auch zukünftige Erweiterungen der Java-Syntax vom DefaultArgumentsInputPanel unterstützt werden können [1, 20, 2].

6.1.2 *ListInARow, JPointer*

visualisiert beliebige Listenstrukturen

Die Klasse `kartina.visualisations.ListInARow` stellt eine beliebige Liste ganz oder auszugsweise als Reihe von Kästchen dar, die jeweils den Wert als Zahl enthalten, den die Liste an der entsprechenden Position hat. Als Liste kann jede Struktur hinterlegt werden, die einen indexbasierten Zugriff auf die einzelnen Elemente ermöglicht (siehe Schnittstelle `kartina.debugger.expr.ValueList`). In den hier vorgestellten Visualisierungen wird jeweils auf ein Array zugegriffen (siehen `kartina.debugger.expr.ArrayBackedValueList`).

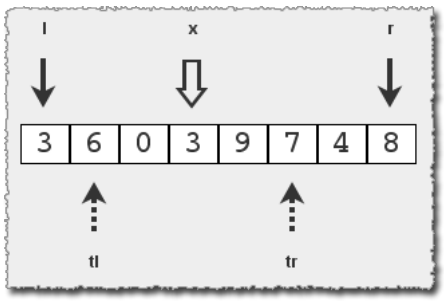

Abbildung 16: ListInARow mit verschiedenen JPointern

Die ListInARow kann mit Zeigern versehen werden, die i.d.R. einen im Algorithmus verwendeten Zeiger, also eine ganzzahlige Variable, auf eine bestimmte Position der Liste repräsentieren. Als grafischer Zeiger kommt die Klasse `kartina.visualisations.JPointer` zum Einsatz, der aus einem Unicode-

Zeichen und einer darüber oder darunter angeordneten Beschriftung, z.B. dem Namen der zugrunde liegenden Variable, besteht.

Die ListInARow muss vom Kontrollvisualisierer händisch aktualisiert werden, wofür i.d.R. ein Aufruf der Methoden `ListInARow.updateList()` und `ListInARow.updatePointers()` genügt. Die ListInARow verhält sich dabei nachsichtig gegenüber ihren Zeigern, d.h. sie blendet Zeiger, deren Wert zum Zeitpunkt der Aktualisierung nicht ermittelt werden kann, aus und blendet sie wieder ein, falls die Variable wieder gültig ist.

Wie bereits erwähnt, bietet auch die ListInARow noch Spielraum für Verbesserungen. Als anzugehende Punkte bei einer Weiterentwicklung sein an dieser Stelle das Problem sich überlagernder Zeiger genannt, sowie die Möglichkeit der zugrunde liegenden Liste, ihre Größe zu verändern.

6.1.3 *SimpleSwapper, BonnySwapper*

Die Klassen `SimpleSwapper` und `BonnySwapper` im Paket `kartina.visualisations` visualisieren das Vertauschen zweier Elemente einer Liste, indem sie die entsprechenden Kästchen einer ListInARow farblich hervorheben und vertauschen.

In der derzeitigen Implementierung findet jedoch noch keine fließende Animation des Tauschvorgangs statt. Interessierten sei das Blog von Kirill Grouchnikov empfohlen, das neben vielen Artikeln über Swing einen Artikel [16] über das Animieren von Swing-Layouts mittels des laf-widget Projekts[2] bietet.

Swapper visualisieren eine existierende Tauschmethode

Beide Vertauscher sind darauf angewiesen, dass der Tauschvorgang innerhalb einer eigenen Methode stattfindet, die die Indizes der zu tauschenden Elemente als Parameter entgegennimmt. Die im Rahmen der Beispielvisualisierungen verwendete Funktion lautet daher

```
private static int[] swap(int[] array, int a, int b).
```

Das Übergeben der eigentlichen Werte (hier `array`), ist nicht nötig, es kann z.B. auch auf Instanzvariablen gearbeitet werden, da die Vertauscher direkt auf einer ListInARow arbeiten.

SimpleSwapper

Der SimpleSwapper arbeitet dabei wie folgt:

1. Er erstellt einen MethodEntry- und einen MethodExitRequest und registriert sich für daraus resultierende Ereignisse.

2. Beim Betreten der angegebenen Tauschmethode hebt der Vertauscher die betroffenen Kästchen durch Einfärben ihrer Hintergründe hervor (auf den Screenshots in Abb. 17 und Abb. 18 wurden die farblichen Hintergründe durch horizontale und vertikale Balken ersetzt, damit die Unterschiede zwischen Simple- und BonnySwapper bezüglich der Darstellung des Tauschvorgangs auch in einer s/w-Darstellung deutlich werden).

3. Im Zuge der normalen Aktualisierung der ListInARow werden die Werte in den betroffenen Kästchen vertauscht, da der Tausch auch vom Algorithmus vorgenommen wird.

2 https://laf-widget.dev.java.net/

4. Beim Verlassen der Tauschmethode stellt der Vertauscher die ursprünglichen Hintergrundfarben wieder her.

Da das Vertauschen der Kästchen auf der Aktualisierung der ListInARow beruht und diese die Kästchen nicht wirklich vertauscht, sondern nur deren Werte ändert, verbleiben die eingefärbten Kästchen den ganzen Tauschprozess über an ihren Plätzen. Dieses unstimmige Verhalten ist nicht auf den ersten Blick zu durchschauen und wirkt sich so negativ auf das eigentliche Ziel, den Algorithmus zu erklären, aus.

SimpleSwapper visualisiert mit hoher Granularität

Außerdem besteht der Tauschvorgang i.d.R. aus den bereits in Kap. 1.4 beschrieben drei Anweisungen, so dass bei einer zeilenweisen Aktualisierung der ListInARow kurzzeitig der Zustand dargestellt wird, indem sich an beiden betroffenen Positionen der Liste derselbe Wert befindet (Abb. 17, Zeitpunkt t_1). Dies spiegelt natürlich den wirklichen Programmzustand und somit den quelltextbezogenen Tauschvorgang wider, eine Darstellung des Tauschvorgangs als eben solcher würde jedoch stärker zum Verständnis des Algorithmus beitragen.

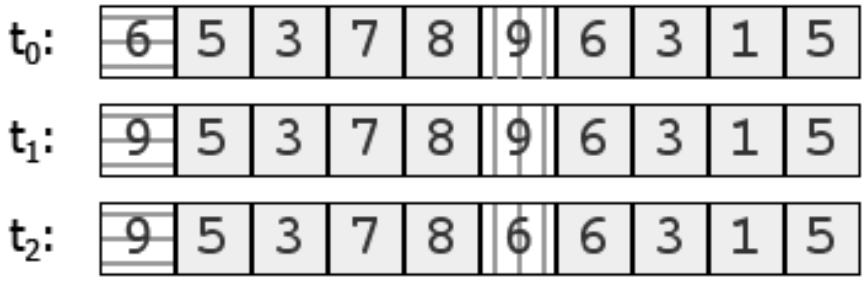

Abbildung 17: Darstellung eines Tauschvorgangs durch den SimpleSwapper zu 3 aufeinanderfolgenden Zeitpunkten

BonnySwapper

Der BonnySwapper ist eine Modifizierung des SimpleSwappers, der die angesprochenen Defizite behebt. Er geht dabei wie folgt vor:

1. Erstellen eines MethodEntryRequests und Registrieren für entsprechende Ereignisse
2. Beim Betreten der angegebenen Tauschmethode
   a) hebt der Vertauscher die betroffenen Kästchen durch Einfärben ihrer Hintergründe hervor.
   b) vertauscht er sowohl den Wert der Kästchen, als auch ihre Hintergrundfarbe in einem (optischen) Schritt.
   c) stellt der Vertauscher die ursprünglichen Hintergrundfarben der Kästchen wieder her.

Zwischen den einzelnen Schritten unter Punkt 2 werden dabei jeweils kleine Pausen eingelegt, so dass der Zuschauer den Tauschprozess verfolgen kann.

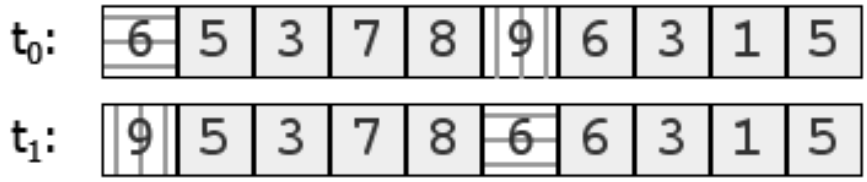

Abbildung 18: Darstellung eines Tauschvorgangs durch den BonnySwapper zu 2 aufeinanderfolgenden Zeitpunkten

6.2 SELECTIONSORT

einfache Visualisierung, nutzt den SimpleSwapper

Die Visualisierung von Selectionsort ist mit relativ wenig Aufwand betrieben worden, um zu zeigen, dass ein (einfacher) Algorithmus bereits mit geringem Aufwand visualisiert werden kann und um in der Gegenüberstellung mit der Visualisierung von Quicksort aufzuzeigen, welche Vorteile eine ausgearbeitete und detaillierte Visualisierung mit sich bringt und welche Möglichkeiten Kartina bietet.

Die Implementierung von Selectionsort findet sich auf der CD, die dieser Thesis beiliegt. Sie besteht aus der eigentlichen Sortiermethode namens `sortDescending()` und der bereits beschriebenen Tauschmethode `swap()`.

Als Eingabekomponente kommt das DefaultArgumentsInputPanel zum Einsatz, die Visualisierung übernimmt eine ListInARow und den Tauschprozess hebt der SimpleSwapper hervor. Der Visualisierungscontroller sorgt neben der anfänglichen Initialisierung nur für die Hervorhebung der aktuell ausgeführten Quelltextzeile sowie für die Aktualisierung der ListInARow.

Dieses einfache Vorgehen ermöglicht bereits eine ansprechende Visualisierung des Ablaufs, hat jedoch folgende Defizite:

- Die Nachteile des SimpleSwappers kommen zum Tragen.
- Die Tauschmethode (`swap()`) wird in der Quelltextansicht betreten (Abb. 19). Für das Verständnis des Algorithmus ist der Ablauf des Tauschvorgangs jedoch irrelevant und es ist daher vorzuziehen, dass der Aufruf der Tauschmethode im Quelltext markiert bleibt, während der Tauschvorgang durchgeführt wird.
- Die Zeiger, die auf das erste unsortierte Element sowie auf das zur Zeit größte gefundene Element (d.h. auf die beiden zu tauschenden Elemente) zeigen, werden während des Tauschvorgangs ausgeblendet, da die zugrunde liegenden Variablen innerhalb der Tauschmethode nicht gültig sind.

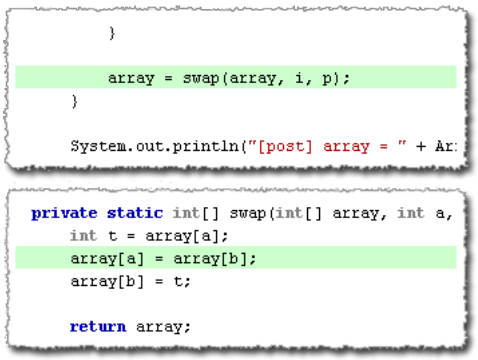

Abbildung 19: Verbleiben auf dem Methodenaufruf (oben) und Betreten der Tauschmethode (oben) in der Quelltextansicht

6.3 Quicksort

aufwändigere Visualisierung, nutzt den BonnySwapper

Die Visualisierung von Quicksort nutzt ebenfalls das DefaultArguments-InputPanel zur Eingabe der zu sortierenden Liste. Anstelle des SimpleSwappers wird jedoch der BonnySwapper genutzt, außerdem wurde der verwendete Controller optimiert, um zu verhindern, dass die Tauschmethode in der Quelltextansicht betreten wird.

Die genannten Defizite der Visualisierung von Selectionsort konnten somit behoben werden, ohne das ein nennenswerter Mehraufwand geleistet werden musste.

Das Hinabsteigen des Algorithmus im Rekursionsbaum wird in dieser Visualisierung durch dunkler werdende Hintergründe der in einem Rekursionsschritt betrachteten Kästchen dargestellt. Eine alternative und besser zu erkennende Repräsentation der Rekursionstiefe stellt das Strecken der betroffenen Kästchen dar, so dass deren Höhe proportional zur Rekursionstiefe ist. Dieses wurde jedoch im Rahmen dieser Thesis nicht mehr umgesetzt.

komplexere Sortiermethode bietet flüssigere Visualisierung...

Der auf der dieser Thesis beiliegenden CD zu findende Quelltext der Quicksortimplementierung weist zwei Methoden auf, die die eigentliche Sortierung durchführen. Die Methode `quickSort()` stützt sich dabei auf die Methode `partition()` ab, um die zu sortierende Teilliste wiederum in eine linke und eine rechte Teilliste zu unterteilen, während `quickSortInOneMethod()` die Funktionalität von `partition()` bereits selber implementiert.

Dies hat bei der hier vorliegenden Visualisierung zur Folge, dass während der Ausführung von `quickSort()` abwechselnd die innerhalb von `partition()` bzw. `quickSort()` verwendeten Zeiger sichtbar sind. Bei der Ausführung von `quickSortInOneMethod()` sind hingegen immer alle Zeiger sichtbar, was als förderlich für das Verständnis des Algorithmus erachtet wird.

...die aber auch anders zu bekommen ist

Es sei an dieser Stelle darauf hingewiesen, dass es durchaus möglich ist, auch während der Visualisierung von `quickSort()` immer alle relevanten Zeiger anzuzeigen. Dies erfordert jedoch Änderungen an der verwendeten ListInARow, da diese, wie bereits erwähnt, Zeiger außerhalb ihres Gültigkeitsbereiches versteckt.

7 ABSCHLUSS

7.1 Zusammenfassung

Aufbauend auf den Java Debugger Interfaces ist mit Kartina ein leicht erweiterbares Framework zur Algorithmenvisualisierung entwickelt worden. Durch den Einsatz einer Trägerklasse werden Beschränkungen des JDI umgangen, außerdem wird die Ausführung der Algorithmen auf beliebigen Daten ermöglicht.

Die lose Kopplung aufgabenspezifischer Klassen mittels des Beobachter-Entwurfsmusters und der weitgehenden Programmierung gegen Schnittstellen gewährleisten eine flexible, gleichzeitig anwenderfreundliche Architektur, die eine breite Palette an Visualisierungen ermöglicht.

Die eingangs beschriebene Rollenaufteilung wird dabei weitestgehend unterstützt, so dass auch unterschiedlich erfahrene bzw. spezialisierte Entwickler als Gruppe an der Visualisierung eines Algorithmus arbeiten können.

Der Einsatz bereits etablierter Werkzeuge wie des Editors der Netbeans IDE und des Java-Interpreters BeanShell unterstützen Entwickler und Vorführer und ermöglichen die Unterstützung kommender Java-Versionen.

Die mit Swing umgesetzten Beispiele zeigen die Möglichkeiten auf, die Kartina bereits bei der Verwendung von Standardkomponenten zur Visualisierung bietet, bieten allerdings auch noch viel Raum für Verbesserungen.

7.2 Offene Punkte

- Vernachlässigt wurde bisher der Umgang mit Texten, insbesondere im Hinblick auf Lokalisierung. Hiervon betroffen sind in erster Linie die grafische Oberfläche von Kartina sowie Fehlermeldungen. Angehen lässt sich dieser Punkt z.B. über sogenannte Ressourcenbündel[1].

- Um den Umgang mit größeren Visualisierungsbibliotheken zu vereinfachen, bietet sich die Möglichkeit an, ein Basisverzeichnis in der Konfigurationsdatei anzugeben. Die Pfade zu den einzelnen Visualisierungskomponenten könnten dann relativ zum Basisverzeichnis angegeben werden. Dies würde die Verwendung einer Konfigurationsdatei auf anderen Rechnern bzw. in einer anderen Umgebung erleichtern.

- Die in Kap. 5.4.10 aufgeführten Einschränkungen des derzeitigen RemoteSupporters.

- Der JDI-StepRequest unterscheidet sich von allen anderen Requests dadurch, dass von ihm nur eine einzige aktivierte Instanz existieren darf. Dies erzwingt nicht nur eine Sonderbehandlung des StepRequests im Vergleich zu den anderen Requests, sondern verhindert in der bisherigen Implementierung von Kartina auch den gleichzeitigen Einsatz von Step- und LineExecutedRequest.

1 http://java.sun.com/docs/books/tutorial/i18n/resbundle/concept.html

Abhilfe schaffen würde ein Kartina-eigener StepRequest (zum Beispiel `KartinaStepRequest`), der als Fassade[2] für den eigentlichen StepRequest dient und diesen vor dem Anwender versteckt.

Dabei kann wie in Abb. 20, Abb. 21 und Abb. 22 dargestellt vorgegangen werden. Hierbei bietet es sich an, dass interessierte Listener mittels eines requestbasierten Filters dafür sorgen, dass sie ausschließlich über die KartinaStepEvents informiert werden, die aufgrund des KartinaStepRequests erzeugt werden, den der Listener erzeugt hat.

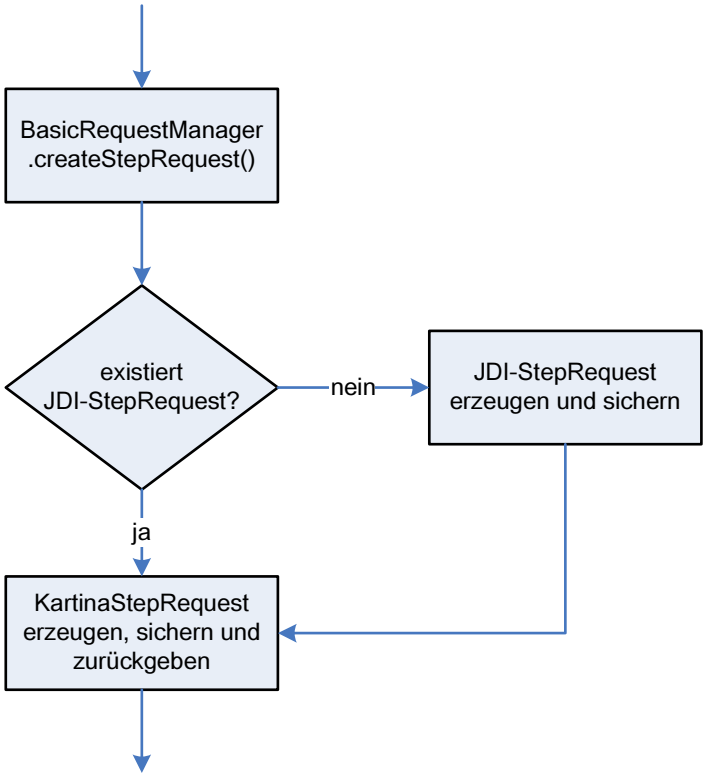

Abbildung 20: Rückgabe der Fassade an den Benutzer, wenn dieser einen StepRequest erzeugen möchte

- Zur Zeit werden nur Logausgaben des RemoteSupporter nur dann ausgegeben, wenn sie mit der Priorität `INFO` oder höher abgesetzt werden. Außerdem werden sie erst angezeigt, wenn der Debuggingprocess abgeschlossen ist. Vermutlich liegt dies an der Anbindung an des Debuggees über das JDI, dies Annahme ist jedoch noch zu überprüfen.

2 In der Literatur wird das Fassaden-Entwurfsmuster [12] zumeist so dargestellt, dass mehrere Objekte durch eine Fassade verborgen werden. Da im Falle der KartinaStepRequests viele Fassaden ein Objekt verbergen, trifft eher das allgemeinere *Handle Body Pattern* (http://c2.com/cgi-bin/wiki?HandleBodyPattern) zu.

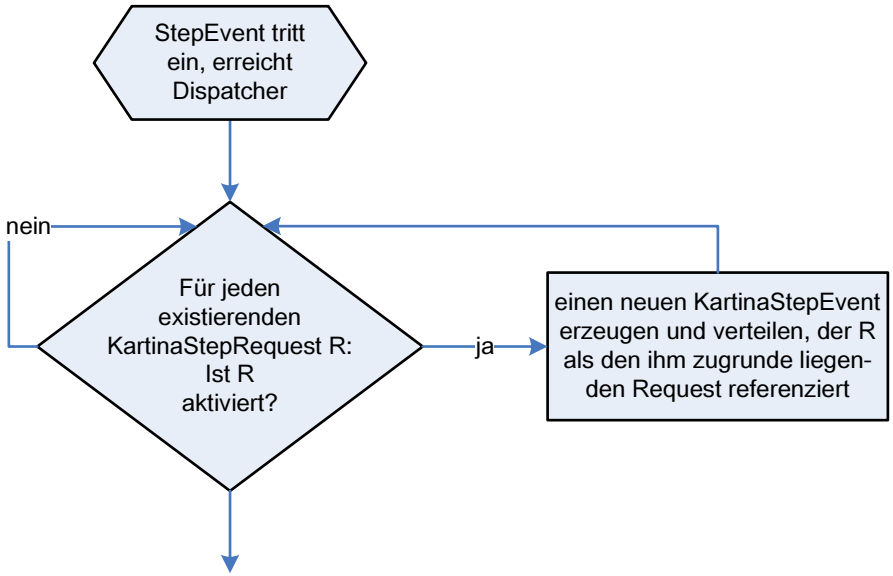

Abbildung 21: Umwandlung eines auftretenden StepRequests in viele KartinaStepEvents

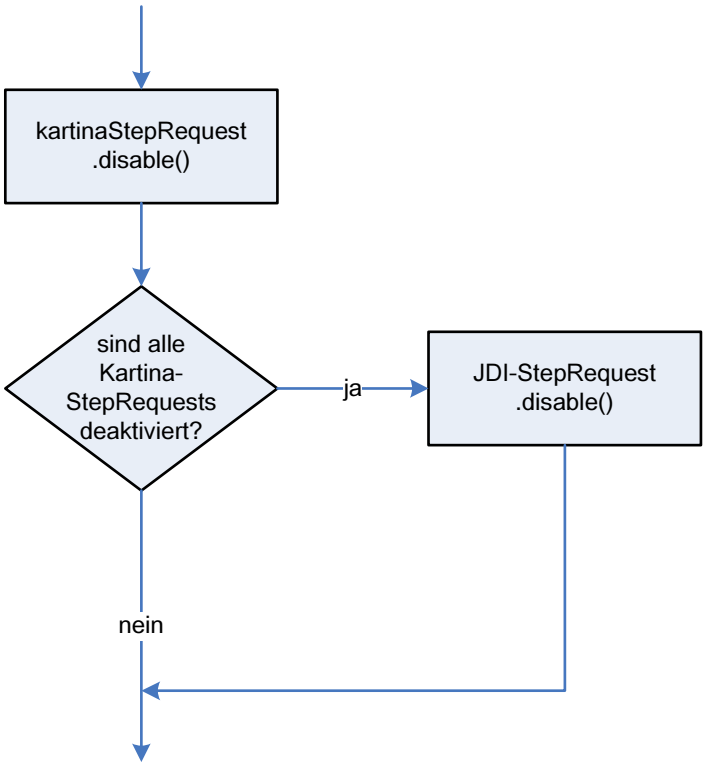

Abbildung 22: Deaktivieren von KartinaStepRequests

7.3 Ausblick

Neben der Erweiterung von Kartina um zusätzliche Funktionalitäten ist auch der Einsatz in anderen Gebieten als der im Rahmen dieser Thesis beschriebenen Algorithmenvisualisierung möglich. Die folgenden Punkte sollen daher Anregungen für die Weiterentwicklung von Kartina geben:

- Eingabekomponente parametrisieren (siehe Kap. 5.6 und Kap. 6.1.1)
- Schrittweises Ausführen

  Der „nächster Schritt"-Steuerknopf führt nicht nur die nächste Quelltextzeile aus, sondern er führt den Debuggee solange aus, bis das nächste Ereignis die Ausführung unterbricht. Zwar lässt sich durch den Einsatz eines StepRequests die zeilenweise Ausführung erreichen, aber ggf. wäre es von Vorteil, den Debuggee schrittweise auszuführen, unabhängig davon, ob der Visualisierer einen aktivierten StepRequest vorsieht oder nicht.
- Nebenläufigkeit, Threadsicherheit

  Kartina ist bisher nicht auf den Umgang mit Algorithmen ausgelegt, die in mehreren Threads ablaufen. Da parallele Algorithmen (siehe [25]) jedoch immer bedeutender werden, sollten sie auch mit Kartina visualisierbar sein.

  Hierbei ist zuerst die Frage zu klären, ob und inwieweit ein Debugger – also auch Kartina – threadsicher sein muss, wenn der Debuggee in mehreren Threads ausgeführt wird.

  Daraus resultieren unterschiedliche Änderungen an Kartina, denn bisher

  1. ist nur der DefaultDelayedVMResumer threadsicher.
  2. wird an verschiedenen Stellen innerhalb Kartinas unter der Annahme operiert, dass der Debuggee nur über einen Thread verfügt. Dies zeigt sich z.B. darin, dass bei Eintritt des ersten Ereignisses eine Referenz auf den am Ereignis beteiligten Thread des Debuggees gesichert wird, der dann auch bei der Behandlung weiterer Ereignisse verwendet wird. Bei einem nebenläufigen Algorithmus ist jedoch nicht gewährleistet, dass alle Ereignisse innerhalb eines Threads auftreten.
- Anmerkungen, Erklärungen

  Der Ablauf eines Algorithmus kann durch (textuelle) Anmerkungen, die zu bestimmten Zeitpunkten eingeblendet werden, näher erklärt werden. j-Algo nutzt dies z.B., um das Verhalten eines Algorithmus bei Grenzfällen näher zu erläutern.
- Fragestellungen, Quiz

  Analog zum vorherigen Punkt kann der Ablauf des Algorithmus auch an bestimmten Stellen unterbrochen werden, um den Zuschauer Fragen beantworten zu lassen. Insbesondere an kritischen Stellen im Ablauf bzw. beim Umgang mit Grenzdaten können geeignete Fragen den Zuschauer zur intensiven Auseinandersetzung mit dem Algorithmus anregen und so das Verständnis fördern.

- Graphenvisualisierung

  Für das Erstellen und Visualisieren von Graphen bietet sich der Einsatz bereits bestehender Komponenten an, bespielsweise das das freie JUNG-Framework[3] oder das ebenfalls freie Graphviz[4]. Evtl. könnte hierzu auch die Netbeans Visual Library[5] heran gezogen werden.

- Unterstützung weiterer Sprachen

  Vermutlich lassen sich auch Algorithmen, die in anderen Sprachen[6] implementiert wurden, mit Kartina visualisieren, so lange sich deren Kompilat auf der Java-VM ausführen lässt. Dies würde u.a. den Einsatz von Sprachen ermöglichen, denen nicht das objektorientierte Programmierparadigma zugrunde liegt.

  Unter Umständen ließe sich Kartina auch zur Visualisierung von nicht auf der Java-VM laufenden Algorithmen einsetzen. Hierzu müsste eine zusätzliche Abstraktionsschicht vor dem JDI eingezogen werden, so dass auch andere ereignisbasierte Debuggerschnittstellen als das JDI angesprochen werden können.

- Es wäre praktisch, über den Zugriff auf lokale Variablen analog zum ModificationWatchpointEvent bzw. AccessWatchpointEvent informiert zu werden. Ob ein entsprechendes Verhalten ohne Unterstützung der virtuellen Maschine realisierbar ist, ist jedoch fraglich.

- Programme rückwärts ausführen

  Durch schrittweises Ausführen des Debuggees und Nachhalten der Programmzustände kann ein Programm rückwärts abgespielt werden. Inwieweit der Debuggee mit durch den Debugger geänderten Programmzustand ab einer bestimmten, bereits passierten Stelle weitergeführt werden kann, ist noch zu untersuchen.

- Laufzeitanalysen

  Durch entsprechende Ereignisse kann der Aufruf von Methoden und der Zugriff auf Instanzvariablen überwacht und gezählt werden. Im Rahmen einer Einführung in die Laufzeitanalyse von Algorithmen kann dies dazu genutzt werden, dem Auditorium die Auswirkung unterschiedlicher Datensätze auf das Laufzeitverhalten eines Algorithmus an praktischen Beispielen zu zeigen.

- Dreidimensionale Visualisierung

  Durch den Einsatz von JOGL[7], jMonkeyEngine[8] oder einer anderen 3D-Bibliothek zur Visualisierung lassen sich alternative Darstellungen ablaufender Algorithmen umsetzen, die ggf. einen einfacheren Zugang zu selbigen ermöglichen.

3 http://jung.sourceforge.net/
4 http://www.graphviz.org/
5 http://graph.netbeans.org/
6 http://www.robert-tolksdorf.de/vmlanguages.html
7 https://jogl.dev.java.net/
8 http://www.jmonkeyengine.com/

- Visualisierung im Web

  Durch das Entwickeln geeigneter EventDispatcher lässt sich Kartina auch als Client-Server System umsetzen, bei dem der Server den Debuggee ausführt und der Client die Visualisierung vornimmt. Zum Einsatz können dabei Java-Applets kommen, offene Technologien wie SVG (Scalable Vector Graphics) oder proprietäre Programme, z.B. Adobe Flash oder Microsoft Silverlight[9].

  Hierbei kommen einmal mehr die Vorteile des Beobachter-Entwurfsmusters und das konsequente Entwickeln gegen Schnittstellen zum Tragen, das den Wechsel zu einem Client-Server System ohne tiefgreifende architektonische Änderungen ermöglicht.

- Profiling

  Auch die Analyse des Laufzeitverhaltens komplexer Softwaresysteme, so genanntes Profiling, kann ereignisbasiert durchgeführt werden. Da allerdings u.a. bereits die Netbeans IDE und die Eclipse IDE über eigene Profiler verfügen, dürften Investitionen in diese Richtung nicht sinnvoll sein.

- Zusatzinformationen aus einem abstrakten Syntaxbaum

  Die *Java Compiler Tree API*[10] ermöglicht die Darstellung von Quellcode als abstraktem Syntaxbaum. Im Zusammenhang mit den Informationen, die den generierten Ereignissen entnommen werden können, ließe sich unter Umständen die Quelltextansicht weiter verbessern, in dem Schleifen oder andere Blöcke gezielt hervorgehoben und annotiert werden.

7.4 Qualitätssicherung

Um eine hohe Qualität des Quelltextes von Kartina zu gewährleisten, wurden die im folgenden beschriebenen Techniken bzw. Werkzeuge im Entwicklungsprozess verwendet:

- Die Durchführung von automatischen Modultests mit JUnit[11] (siehe Bericht auf der beiliegenden CD).

- Das automatische Mocken von Diensten im Rahmen der Modultests mittels EasyMock[12], das eine gut lesbare *einbettbare domänenspezifische Sprache* (*Embedded Domain-Specific Lanuage*, *EDSL*[13]) zur Definition des erwarteten Verhaltens der zu testenden Klasse einsetzt.

- Das Ermitteln des Grades der Testabdeckung mittels Cobertura[14], was das Schreiben effektiver und effizienter Testfälle unterstützt. Cobertura stellt dabei die Anzahl der Zeilen und Verzweigungen einer Klasse,

9 http://silverlight.net/
10 http://java.sun.com/javase/6/docs/jdk/api/javac/tree/index.html
11 http://www.junit.org/
12 http://www.easymock.org/
13 Siehe dazu Freeman und Pryce [13] die zwar von der Entwicklung einer EDSL im Rahmen von jMock berichten, jedoch auch einen allgemeinen Einstieg in das Thema EDSL bieten. Empfehlenswert ist auch Cuadrado und Molina [6], in dem die Entwicklung einer EDSL mit Ruby beschreiben wird.
14 http://cobertura.sourceforge.net/

die innerhalb des Testlaufs ausgeführt werden, der Gesamtanzahl selbiger gegenüber (siehe Bericht auf der beiliegenden CD).

- Das Aufzeigen der Abweichungen von bestimmten Konventionen, die im Hinblick auf qualitativ hochwertigen Code aufgestellt wurden, mittels Checkstyle[15] (siehe Bericht auf der beiliegenden CD).

- Der konsequente Einsatz eines Loggingmoduls unterstützt die Entwicklung neuer Visualisierungen sowie die Weiterentwicklung von Kartina, da es den Entwicklern ermöglicht, den Ablauf anhand der Logausgaben zu verfolgen.

  Die eingesetzte Java Logging API[16] ist über eine externe Eigenschaftendatei konfigurierbar, der Umgang mit ihr wird durch den Einsatz einer Hilfsklasse (`de.oakgrove.kartina.util.KartinaLogger`) vereinfacht.

15 http://checkstyle.sourceforge.net/
16 http://java.sun.com/j2se/1.5.0/docs/guide/logging/overview.html

[1] : *JSR 223: Scripting for the Java Platform*. Januar 2008. – URL http://jcp.org/en/jsr/detail?id=223 (referenziert auf Seite 56.)

[2] : *JSR 274: The BeanShell Scripting Language*. Januar 2008. – URL http://jcp.org/en/jsr/detail?id=274 (referenziert auf Seite 56.)

[3] Ben-Ari, Mordechai ; Myller, Niko ; Sutinen, Erkki ; Tarhio, Jorma: Perspectives on Program Animation with Jeliot. In: Diehl, Stephan (Hrsg.): *Software Visualization* Bd. Volume 2528/2002. Heidelberg, Deutschland, 2002, S. 31–44 (referenziert auf Seite 15.)

[4] Budd, Timothy: *An Introduction to Object-Oriented Programming*. 2. Auflage. Addison-Wesley, 1997 (referenziert auf Seite 19.)

[5] Craig, Ian: *The interpretation of object-oriented programming languages*. 2. Auflage. London, Great Britain : Springer-Verlag, 2002 (referenziert auf Seite 56.)

[6] Cuadrado, Jesús S. ; Molina, Jesús G.: Building Domain-Specific Lanuages for Model-Driven Development. In: *IEEE Software* (2007), September (referenziert auf Seite 66.)

[7] Demetrescu, Camil ; Finocchi, Irene ; Stasko, John T.: Specifying Algorithm Visualizations. In: Diehl, Stephan (Hrsg.): *Software Visualization* Bd. Volume 2528/2002. Heidelberg, Deutschland, 2002, S. 17–30 (referenziert auf den Seiten 5, 6.)

[8] Diehl, Stephan (Hrsg.) ; Springer-Verlag (Veranst.): *Software Visualization*. Bd. *Volume 2528/2002*. Heidelberg, Deutschland, 2002. (Lecture Notes in Computer Science) (referenziert auf Seite 13.)

[9] Eastern Finland Virtual University (Veranst.): *Jeliot 3 – User Guide*. Januar 2008. – URL http://cs.joensuu.fi/jeliot/files/userguide.pdf (referenziert auf Seite 13.)

[10] Fleischer, Rudolf ; Kučera, Luděk: Algorithm Animation for Teaching. In: Diehl, Stephan (Hrsg.): *Software Visualization* Bd. Volume 2528/2002. Heidelberg, Deutschland, 2002, S. 113–128 (referenziert auf den Seiten 3, 4 und 15.)

[11] Fowler, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. Januar 2008. – URL http://www.martinfowler.com/articles/injection.html (referenziert auf Seite 19.)

[12] Freeman, Elisabeth ; Freeman, Eric: *Head First Design Patterns*. Sebastopol, CA, USA : O'Reilly, 2004 (referenziert auf den Seiten 7, 19, 29, 31 und 62.)

[13] Freeman, Steven ; Pryce, Nat: *Evolving an Embedded Domain-Specific Language in Java*. Oktober 2006. – URL http://www.mockobjects.com/files/evolving_an_edsl.ooplsa2006.pdf (referenziert auf Seite 66.)

[14] GEERTJAN: *Creating Error Annotations in NetBeans*. Januar 2008. – URL http://blogs.sun.com/geertjan/entry/creating_error_annotations_in_netbeans (referenziert auf Seite 53.)

[15] GEIGER, Leif: *Design Level Debugging mit Fujaba*. Oktober 2002. – URL http://www.coobra.cs.uni-kassel.de/fileadmin/se/publications/DLD.pdf (referenziert auf den Seiten 45, 46.)

[16] GROUCHNIKOV, Kirill: *Animating layouts*. Januar 2008. – URL http://weblogs.java.net/blog/kirillcool/archive/2006/09/animating_layou.html (referenziert auf Seite 57.)

[17] KREFT, Klaus ; LANGER, Angelika: Performance: Profiling. In: *JavaSPEKTRUM* (2005), November. – URL http://www.angelikalanger.com/Articles/JavaSpektrum/23.ProfilingTools/23.ProfilingTools.html (referenziert auf Seite 5.)

[18] LADDAD, Ramnivas: *AspectJ in Action*. Greenwich, CT, USA : Manning Publications Co., 2003 (referenziert auf Seite 6.)

[19] LARMAN, Craig: *UML 2 und Patterns angewendet*. Bonn, Deutschland : mitp-Verlang, 2005 (referenziert auf Seite 19.)

[20] O'CONNER, John: *Scripting for the Java Platform*. Januar 2008. – URL http://java.sun.com/developer/technicalArticles/J2SE/Desktop/scripting/ (referenziert auf Seite 56.)

[21] SAM-BODDEN, Brian: *Beginning POJOs*. Berkeley, CA, USA : Apress, 2006 (referenziert auf Seite 19.)

[22] SCHÖNING, Uwe: *Algorithmik*. Heidelberg, Deutschland : Spektrum Akad. Verlag, 2001 (referenziert auf den Seiten 14, 15.)

[23] ULLENBOOM, Christian: *Java ist auch eine Insel*. 6. Auflage. Bonn, Deutschland : Galileo Press, 2007 (referenziert auf den Seiten 43, 44 und 48.)

[24] VIEIRO, Antonio: *Building a standalone NetBeans editor*. Januar 2008. – URL http://www.antonioshome.net/kitchen/netbeans/nbms-standalone.php (referenziert auf Seite 53.)

[25] XAVIER, C. ; IYENGAR, S. S.: *Introduction to parallel algorithms*. New York, NY, USA : John Wiley & Sons, Inc., 1998 (referenziert auf Seite 64.)

Kolophon

Diese Thesis wurde in LaTeX 2_ε gesetzt, unter der Verwendung von Hermann Zapfs *Palatino* und *Euler* Fonts. Die Listings sind in dem Font *Bera Mono* gesetzt.

Das Layout basiert auf dem LaTeX 2_ε-Paket „`classicthesis`“ (erhätlich über `CTAN`) von André Miede und wurde nur unwesentlich verändert.

Kartina – mit kyrillischen Buchstaben Картина geschrieben – bedeutet im Russischen soviel wie Bild oder Gemälde. Da ich die Gelegenheit hatte, ein wunderschönes Auslandssemester in Tomsk, Westsibirien zu verbringen und ich gerne daran zurück denke, habe ich mich dazu entschieden, mir mit der Wahl des Namens eine kleine Freude zu bereiten.

Version vom 5. Februar 2008 um 21:16 Uhr.