Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Software-Engineering

Masterarbeit

Visualisierung von Synchronisationspunkten in Kombination mit der Statik und Dynamik eines Softwaresystems

Philipp Döhring (pdo@informatik.uni-kiel.de)

7. Oktober 2012

Betreut von: Prof. Dr. Wilhelm Hasselbring
Dipl.-Inf. Jan Waller

Eidesstattliche Erklärung

Hiermit versichere ich, Philipp Döhring, dass ich diese Arbeit selbständig verfasst, keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel verwendet habe und diese Arbeit in keinem anderen Prüfungsverfahren eingereicht habe.

Ort, Datum, Unterschrift

Zusammenfassung

Softwarevisualisierungen leisten häufig einen entscheidenden Betrag zum Verständnis komplexer Anwendungen. Zum Zeitpunkt dieser Arbeit existieren zwar bereits eine Vielzahl unterschiedlicher Ansätze zur Darstellung der Applikationsstatik und -dynamik, jedoch mangelt es diesen bislang oftmals an der Visualisierbarkeit von Synchronisationsvorgängen. Diese Arbeit hat sich diesem Problem angenommen. Wir stellen dazu zunächst verschiedene bestehende Ansätze vor, evaluieren diese und präsentieren auf deren Grundlage einen eigenen, ganzheitlichen Darstellungsansatz auf Basis einer städtischen Metapher. Unser Softwarewerkzeug *Synchrovis* implementiert unsere Visualisierung prototypisch. Wir werden seine Architektur und Funktionalität grundlegend vorstellen sowie abschließend seine Qualitäten anhand einer eigenen Evaluation sowie einer Expertenbefragung aufzeigen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einführung

Mit dem Trend der vergangenen Jahre zu immer ausgeprägterer Parallelität bei Applikationen, auch ausgelöst durch ausbleibenden Fortschritte in der Entwicklung schnellerer Prozessoren, erhielten Konzepte der nebenläufigen Programmierung zunehmend Einzug in die alltäglichen Arbeitstechniken heutiger Softwareentwickler. Nebenläufige Anwendungen bieten dabei einige Charakteristika, welche rein sequenzielle Anwendungen nicht besitzen. So erlauben sie eine hohe Skalierbarkeit durch geschickte Lastverteilung auf mehrere Prozessoren oder ermöglichen die Erstellung fehlertoleranter Systeme, in welchen beim Auftreten von Ausnahmefällen oder Problemen lediglich einzelne Threads ausfallen.
Synchronisationsmechanismen wie beispielsweise Semaphore oder Monitore sollen ein reibungsloses Nebeneinander der einzelnen Threads ermöglichen. Doch Fehler in nebenläufigen Systemen sind aufgrund der Vielzahl gleichzeitig existierender Kontrollflüsse und des vorherrschenden Nichtdeterminismus des Schedulers deutlich schwieriger zu identifizieren als in sequenziellen Applikationen [52, 53]. So kommt es nicht selten vor, dass erst nach Monaten der Softwarebenutzung Deadlocks auftreten oder Race-Conditions zu Fehlern führen.
Wohingegen Programmiersprachen immer feinere Konstrukte zur nebenläufigen Synchronisation bereitstellen, fallen Debugging-Werkzeuge hinter diesen immer weiter zurück [53]. So existieren bis heute kaum effiziente Werkzeuge, welche den Anwendungsentwickler bei der Fehlersuche geeignet unterstützen. Das alleinige Betrachten des Quellcodes oder der aufgezeichneten Programmtraces hilft kaum beim Verständnis dieser Fehler, da es sich im ersteren Fall häufig als ausgesprochen schwierig erweist, sämtliche potentielle Schedulings durchspielen. Traces sind hingegen oftmals so umfangreich und detailliert in ihren Informationen, dass selbst das bloße Nachvollziehen der Applikationsdynamik eine Herausforderung darstellt. Stattdessen ist eine abstraktere Darstellung des Programmverlaufs erforderlich, welche dem Programmierer lediglich die relevanten Informationen darbietet und ihm die Ursachen von Programmierfehlern leicht identifizieren lässt. Visualisierungen vermögen dies zu leisten, da sie aufgrund ihrer Charakteristik auf eine sehr viel abstraktere Weise als durch eine textuelle Beschreibung auch eine Auswahl von Informationen übersichtlich darstellen können und so den Fokus des Betrachters auf das Wesentliche lenken.
Doch nicht nur zum Debuggen existierender Softwareanwendungen können Visualisierungen verwendet werden. Bereits heute sind Techniken der nebenläufigen Programmierung ein fundamentaler Bestandteil jedes Informatik-Studiums. Das Lehren dieser Konzepte gestaltet sich jedoch häufig schwierig. So ist es oftmals nicht leicht Studenten beizubringen, dass ihre nebenläufige Implementierung fehlerhaft ist, obwohl sie bislang stets das korrekte Ergebnis lieferte [28]. Eine geeignete Darstellung dieser Konstrukte könnte Studenten hingegen dabei unterstützen, ein tieferes Verständnis für nebenläufige Konzepte sowie das hierbei nötige Problembewusstsein zu entwickeln.

Zwar existieren bereits eine Reihe von Ansätzen zur Darstellung von Softwaresystemen, jedoch zeigen diese oftmals lediglich einzelne Facetten einer betrachteten Anwendung. So ist uns bislang kein Ansatz bekannt, welcher sowohl die Statik und Dynamik als auch ggf. auftretende Synchronisationsvorgänge in einer einzigen Visualisierung vereint.
Diese Arbeit hat sich diesem Problem angenommen. Wir präsentieren und evaluieren dazu zunächst bereits bekannte Visualisierungsansätze und entwickeln auf deren Grundlage unsere eigene Darstellung der Statik und Dynamik in Kombination mit den Synchronisationspunkten. Im Rahmen dieser Arbeit haben wir dabei den Softwareprototypen *Synchrovis* entwickelt, welcher unseren Darstellungsansatz aus einem *KDM-Modell* [1] und einem mit dem *KIEKER Monitoring & Analysis Framework* [34, 37] aufgezeichneten Programmtrace zu erzeugen vermag.

1.1. Struktur dieses Dokuments

Nach einer kurzen Einführung in die Welt der Synchronisationsmechanismen in Kapitel 2 präsentieren wir in Kapitel 3 verschiedene, bereits existierende Ansätze zur Darstellung von Softwaresystemen. Viele von diesen besitzen dabei qualitativ große Unterschiede, welche wir in Kapitel 4 anhand einer Evaluation mittels eigens entwickelter Bewertungskriterien aufzeigen. Unser Ziel ist es dabei, denjenigen Ansatz zu identifizieren, welcher sich am besten als Grundlage für die Entwicklung unserer ganzheitlichen Visualisierung eignet. In Kapitel 5 stellen wir anschließend diesen Ansatz detailliert vor, welchen unser Softwarewerkzeug *Synchrovis* prototypisch implementiert. Wir werden seine Architektur und Funktionalität umfassend in Kapitel 6 präsentieren sowie die Qualität unserer Darstellung anhand einer Evaluation mittels Bewertungskriterien sowie einer Expertenbefragung in Kapitel 7 aufzeigen. Eine Vorstellung themenverwandter Arbeiten in Kapitel 8 sowie ein abschließendes Fazit in Kapitel 9 runden diese Arbeit ab.

1.2. Eine Anmerkung zur Sprache

Die Informatik ist eine Wissenschaft, dessen Terminologie in besonderem Maße durch die englische Sprache geprägt ist. Auch obwohl diese Arbeit auf Deutsch verfasst wurde und wir daher bestrebt waren, auch für englischsprachige Fachausdrücke deutsche Begrifflichkeiten zu verwenden, ließ sich der Gebrauch von Anglizismen nicht gänzlich vermeiden. Für viele englische Bezeichnungen wie beispielsweise *Thread* oder *Context-Switch* existieren keine allgemein gebräuchlichen deutschen Begriffe. So würde eine wortwörtliche Übersetzung aufgrund der ungewöhnlichen Wortwahl oder der nicht erkannten Terminologie häufig den Lesefluss stören. Wir haben uns daher dazu entschlossen, die beiden oben genannten Bezeichnungen sowie insbesondere die Begriffe *Trace*, *Thread-Join*, *Deadlock*, *Framework* sowie *Hotspot* anstelle sinngemäßer deutscher Wörter zu verwenden.

2. Grundlagen

In diesem Kapitel möchten wir dem Leser einen Überblick über verschiedene Synchronisationsmechanismen geben sowie dasjenige Wissen vermitteln, welches für das Verständnis dieser Arbeit relevant ist. Von vielen Wissenschaftlern werden dabei die Begriffe der *Statik*, *Dynamik* und *Synchronisationspunkte* leicht unterschiedlich verstanden. Wir beginnen daher zunächst mit der Definition dieser Bezeichnungen gemäß unserer eigenen Vorstellungen, auf welche wir uns in dieser Arbeit einheitlich beziehen. Abschließend werden wir diejenigen Synchronisationsmechanismen erläutern, welche zur Integration in unsere eigene Visualisierung in Frage kommen.

2.1. Begriffsdefinitionen

Die *Statik* einer Applikation bezeichnet die Gesamtheit aller in der Anwendung definierten und durch die Programmiersprache vorgegebenen Entitäten inklusive deren Komposition und gegenseitige Abhängigkeitsbeziehungen. Am Beispiel einer Java-Anwendung sind dies unter Anderem Methoden und Attribute, die Paket- und Klassenstruktur sowie Assoziations, Ableitungs- und Methodenaufrufbeziehungen.
Die *Dynamik* einer Applikation bezeichnet hingegen die Menge sämtlicher zur Laufzeit beobachtbarer Vorgänge und Zustände bezüglich der betrachteten Anwendung. Zu diesen gehören neben Context-Switches und Threadzustände auch konkrete Ausführungspfade, Objektinstantiierungen und -eliminierungen sowie gemessene Ausführungszeiten von Methoden.
Ein *Synchronisationspunkt* bezeichnet einen Inter-Prozess-, bzw. einen Inter-Thread-Kommunikationsvorgang, dessen Ziel die Vermeidung von unerwünschten Systemzuständen (v.A. Deadlocks und Race-Conditions), bzw. die Harmonisierung des Programmablaufs (siehe Thread-Joins, Abschnitt 2.2.4) ist. Diese Interaktion ist dabei nicht nur auf einen direkten Nachrichtenaustausch beschränkt. So bezeichnen wir beispielsweise auch eine gegenseitige Beeinflussung des Schedulings (d. h. insbesondere das Suspendieren von Threads) als einen Synchronisationspunkt.

2.2. Synchronisationsmechanismen

Synchronisationsmechanismen dienen zur Vermeidung von unerwünschten Systemzuständen. Sie schützen verteilt genutzte Applikationselemente (v. A. Variablen und Methoden) vor dem gleichzeitigen Zugriff mehrerer Threads oder Prozesse und verhindern auf diese Weise das Entstehen von Race-Conditions, Deadlocks und sonstiger durch das Scheduling verursachten Problemen. Wir gehen im Folgenden auf diese Mechanismen näher ein.

2.2.1. Semaphore

Eine *Semaphore* [25] ist eine von E. W. Dijkstra entwickelte Datenstruktur, welche mittels ihrer beiden atomaren Operationen p (kurz für *prolaag*, dt. *vermindern*) und v (kurz für *verhoog*, dt. *erhöhen*)[1] [2] eine relativ simple Möglichkeit zur Wahrung des gegenseitigen Ausschlusses von kritischen Sektionen darstellt. Ein Semaphor besteht dabei aus einer internen Zählvariablen s sowie einer Warteschlange w. Der Wert von s gibt initial die Anzahl derjenigen Threads an, welche gleichzeitig die kritische Sektion betreten dürfen. Im Zuge dieser Arbeit betrachten wir dabei ausschließlich *binäre* Semaphoren, d. h. es gilt initial $s = 1$. Solche, welche hingegen mehreren Threads zugleich Zugriff auf die kritische Sektion gewähren werden demgegenüber als *zählende* Semaphoren bezeichnet. Abbildung 1 zeigt einen Ausschnitt aus einer beispielhaften Semaphor-Implementierung in Pseudocode.

```
1  class Semaphore {
2
3      int s = 1;
4      Queue w = new Queue();
5
6      atomic p() {
7          s = s-1;
8          if (s < 0) { suspendiereSelbst(w) }
9      }
10
11     atomic v() {
12         s = s+1;
13         if (s <= 0) { reaktiviereErstenThread(w) }
14     }
15 }
```

Abbildung 1: Ein Ausschnitt aus einer beispielhaften binären Semaphor-Implementierung

Wird die p-Operation einer binären Semaphore vor und die v-Operation unmittelbar nach dem kritischen Abschnitt aufgerufen, ist die kritische Sektion vor gleichzeitigem Zugriff geschützt. Führt ein Thread die p-Methode aus und besitzt s den Wert 1, so kann mit der seiner weiteren Instruktionsausführung fortfahren. Andernfalls wird er suspendiert und in w eingereiht. Verlässt hingegen ein Thread die kritische Sektion, so ruft er die v-Methode auf. Die Zählvariable wird inkrementiert und ein in w wartender Thread (falls existent) wird reaktiviert, welcher seinerseits die Ausführung von Anweisungen im kritischen Abschnitt fortführen kann.

[1] Weitere, ebenso verbreitete Erklärungen sind *passeer*, *probeeren* (dt. *überprüfen*), bzw. *vrijgeven*, *verhogen* (dt. *erhöhen*) [59]

Unsere Beispielimplementierung garantiert dabei eine „echte“ Warteschlange gemäß des so genannten *Windhundprinzips* (engl. „*first come, first served*“) und wird daher auch als *starke* Semaphore bezeichnet. Eine chronologisch korrekte Reaktivierung der suspendierten Threads wird jedoch im Allgemeinen nicht garantiert (*schwache* Semaphore). Die Threads, welche „passieren“ und „verlassen“ müssen zudem nicht identisch sein. Eine feste Bindung an die Aufrufsequenz *p()* → kritische Sektion → *v()* findet daher nicht statt. Dieser Umstand wird beispielsweise bei der Lösung des bekannten Produzenten-Konsumenten-Problems [20] ausgenutzt.

2.2.2. Monitore

Ein *Monitor* [20] bezeichnet eine Datenstruktur, welche Objekte und Operationen beinhaltet, die nur unter gegenseitigem Ausschluss, d. h. von nur einem Thread gleichzeitig, aufgerufen werden dürfen. Der exklusive Zugriff wird dabei mittels einer *Sperrvariablen* (engl. *lock*) geregelt. Möchte ein Thread auf eine oder mehrere Methoden oder Objekte innerhalb eines Monitors zugreifen, so muss er sich zunächst um diese Variable bewerben. Befindet sich kein Thread im Fokus des Monitors, so kann er die Sperrvariable belegen und darauffolgend die Inhalte des Monitors aufrufen. Andernfalls wird der Thread temporär suspendiert und wartet, bis ihm der Lock gegeben wurde. Auf die Methoden und Objekte im Monitor besitzt ein Thread dabei exklusiven Zugriff, d.h. er kann nach Belieben unter Anderem Funkionen Aufrufen oder Attributwerte überschreiben. Verlässt der Thread jedoch den Fokus des Monitors, verliert er auch seinen aktuellen Lock, welcher nun an einen beliebigen anderen auf diesen wartenden Thread vergeben werden kann.
In Java ist ein Monitor ähnliches Konstrukt in Form der *synchronized*-Methoden implementiert [5, 20]. Sämtliche Anweisungssequenzen, welche nur unter gegenseitigem Ausschluss ausgeführt werden dürfen, werden hierbei durch einen *synchronized(o)*-Block umschlossen. Das Objekt *o* gibt hierbei als Parameter diejenige Sperrvariable an, mittels welcher der exklusive Zugriff gesteuert wird und auf welches synchronisiert wird. Werden demnach mehrere Blöcke mit dem selben Parameter gebildet, werden die darin umschlossenen Instruktionen konzeptionell zu gleichen Monitor-Einheit zusammengefasst.
Zur Vereinfachung der Syntax lässt sich das Schlüsselwort *synchronized* auch direkt in einer Methodensignatur verwenden. Dies entspricht der Umrahmung des gesamten Funktionsrumpfes mit einem Synchronized-Block, wobei das aktuelle Klassenobjekt als Sperrvariable verwendet wird.
Im Gegensatz zu den binären Semaphoren, bei welchen die *p*- und *v*-Operationen auch in beliebiger Reihenfolge von unterschiedlichen Threads aufgerufen können, sind die Sperrvariablen der Monitore fest an einen Thread gebunden. Das Synchronized-Konzept in Java stellt daher eine Spezialisierung der (binären) Semaphorenstruktur dar.

2.2.3. Wait und Notify

Die Operationen *wait*, *notify* und *notifyAll* ermöglichen in Java die manuelle Suspendierung bzw. Reaktivierung von Threads und werden von sämtlichen Objekten zur Verfügung gestellt. Ruft ein Thread die *wait*-Methode eines Objekts o auf, so wird dieser suspendiert sowie sämtliche von ihm ggf. belegte Monitor-Locks freigegeben. Damit er mit seinen Berechnungsschritten fortfahren kann, muss ein anderer Thread die *notify*-, bzw. *notifyAll*-Methode auf o aufrufen. Erstere erweckt auf nichtdeterministische Weise einen blockierten Thread, wohingegen Letztere zur Reaktivierung sämtlicher auf o suspendierter Threads führt [5].
Wait- und Notify-Operationen werden oftmals zur Vermeidung von *Busy-Waiting* eingesetzt. Dieser Begriff bezeichnet dabei die (häufig sinnlose) Ausführung von Instruktionen beim Warten auf die Erfüllung einer zur Fortführung der Berechnung notwendigen Bedingung.

2.2.4. Thread-Joins

Der Aufruf der *join*-Operation auf einem Threadobjekt t führt zur Suspendierung der eigenen Ausführung und automatischen Reaktivierung nach der Terminierung des von t repräsentierten Threads. Ist dieser bereits terminiert, hat der Methodenaufruf keinen Effekt [5]. Häufig lässt sich über einen optionalen Parameter eine maximale Zeit angeben, welche ein Thread auf die Beendigung eines anderen wartet.
Die *join*-Methode wird vor Allem dann verwendet, wenn Berechnungsschritte eines Algorithmus auf mehrere Threads ausgelagert werden und nun auf deren Fertigstellung gewartet wird.

2.2.5. Linda-Modell

Das Linda-Modell [21] ist ein sprachunabhängiges Synchronisationsmodell, welches in vielen Programmiersprachen (u. A. C, Fortran und Scheme) verfügbar ist. Gängige Java-Implementierungen dieses Konzepts stellen dabei *JavaSpaces* [9] und *LighTS* [17] dar. Im Gegensatz zu den bisher vorgestellten Synchronisationsmechanismen ermöglicht das Linda-Modell eine Kommunikation zwischen verschiedenen Threads oder Prozessen in Form eines gegenseitigen Nachrichtenaustausches. Es findet jedoch keine unmittelbare Interaktion zwischen den Teilnehmern statt. Stattdessen werden die Nachrichten (*Tupel* genannt) zunächst in einem sogenannten *Tupelraum* als gemeinschaftlich zugängliche Speichereinheit zwischengelagert.
Tupel bestehen dabei aus einem Typ t und einer beliebig langen Sequenz von Werten $(w_1, \ldots, w_n)$. Textuell werden sie häufig in der Form $(`t`, w_1, \ldots, w_n)$ notiert. Die Funktion *out(v)* fügt einen beliebige Tupel v in den Tupelraum ein. Die Methode *in(v)* entfernt hingegen das Tupel v und gibt dieses als Rückgabewert zurück, falls v im Tupelraum vorhanden ist. Andernfalls suspendiert diese Prozedur den aufrufenden Thread solange

bis v verfügbar ist und entfernt diesen Tupel anschließend. *in(v)* kann dabei auch lediglich eine Art Schablone übergeben werden. So entfernt die Funktion beispielsweise bei dem Argument $(`H`, ?i, ?j)$ einen Tupel vom Typ H mit einer Wertesequenz der Länge zwei (*Pattern Matching*). Der erste Wert wird dabei an die Variable i und der zweite an j gebunden. Stimmen mehrere Tupel mit dieser Schablone überein, wird nichtdeterministisch ein passendes Tupel aus dem Tupelraum entfernt. Durch die Variablenbindung stehen die Werte des eliminierten Objekts dem aufrufenden Thread stets zur Verfügung. Dies ermöglicht den indirekten Nachrichtenaustausch.
Neben der *in()*- und *out()*-Operation bietet das Linda-Modell noch weitere, von diesen abgeleitete Prozeduren. So verhält sich *rd(v)* wie *in(v)*, jedoch entfernt diese Methode das Tupel v nicht aus dem Tupelraum. Analog verhalten sich *inp(v)* und *rdp(v)* wie *in(v)*, respektive *rd(v)*, jedoch suspendieren diese den ausführenden Thread nicht bei dem Nichtvorhandensein von v. Als Rückgabewert dieser beiden Methoden erhält man dabei einen booleschen Wert. Dieser ist *false*, falls sich v nicht im Tupelraum befindet. *eval(t)* erzeugt hingegen einen neuen Prozess, welcher den Term t auswertet. Der Rückgabewert von t ist dabei ein Tupel, welcher im Anschluss in den Tupelraum eingefügt wird.

3. Existierende Visualisierungsansätze

Zum Zeitpunkt dieser Arbeit existieren bereits ein Vielzahl unterschiedlicher Visualisierungsansätze zur Darstellung der Statik, der Dynamik und / oder der Synchronisationsvorgänge eines Softwaresystems. Viele dieser Ansätze besitzen dabei gänzlich unterschiedliche charakteristische Eigenschaften. So unterscheiden sie sich häufig in der Sicht, welche sie auf eine Anwendung bieten, ihrem Abstraktionsgrad oder auch in der Menge der Informationen, welche sie zugleich visualisieren (können). Im Sinne dieser Arbeit beschränken wir uns dabei auf die Betrachtung solcher Ansätze, welche entweder eine kombinierte Darstellung der Statik und Dynamik liefern oder Synchronisationspunkte gemeinsam mit der Dynamik visualisieren, da unser Ziel die Entwicklung einer ganzheitlichen Darstellung ist.
Im Folgenden möchten wir eine Auswahl verschiedener existierender Visualisierungsansätze näher vorstellen, welche wir im nachfolgenden Kapitel bewerten werden. Sie dienen uns dabei als Ausgangsbasis für die Entwicklung unseres eigenen Visualisierungsansatzes.

3.1. UML und auf UML basierende Ansätze

Vergleichsweise viele Ansätze zur Darstellung von Threadsynchronisationen basieren auf den Diagrammen der *Unified Modeling Language* (*UML*). Diese unterstützt per se jedoch kaum Synchronisationsmechanismen und ist daher in ihrer ursprünglichen Form für unsere Visualisierung ungeeignet. Aus diesem Grund liegt ein häufiger Ansatz zur Integration von Synchronisationsvorgängen in der Erweiterung der UML-Diagrammtypen. Nachfolgend möchten wir einen Überblick über die UML geben sowie eine Auswahl existierender Erweiterungen näher vorstellen.

3.1.1. UML

Die *Unified Modeling Language* (kurz: *UML*) [54] ist eine von der *Object Management Group* (*OMG*)[2] entwickelte und verwaltete, standardisierte grafische Modellierungssprache zur Spezifikation, Konstruktion und Dokumentation von Softwaresystemen. Heutzutage stellt diese den De-facto-Standard im Bereich der Modellierung in der Softwareentwicklung dar [53] und beinhaltet in ihrer aktuellen Version 2.4.1 insgesamt vierzehn verschiedene Diagrammtypen zur Darstellung der unterschiedlichen Facetten eines objektorientierten Softwaresystems auf unterschiedlichem Abstraktionsniveau.

[2] `http://www.omg.org`

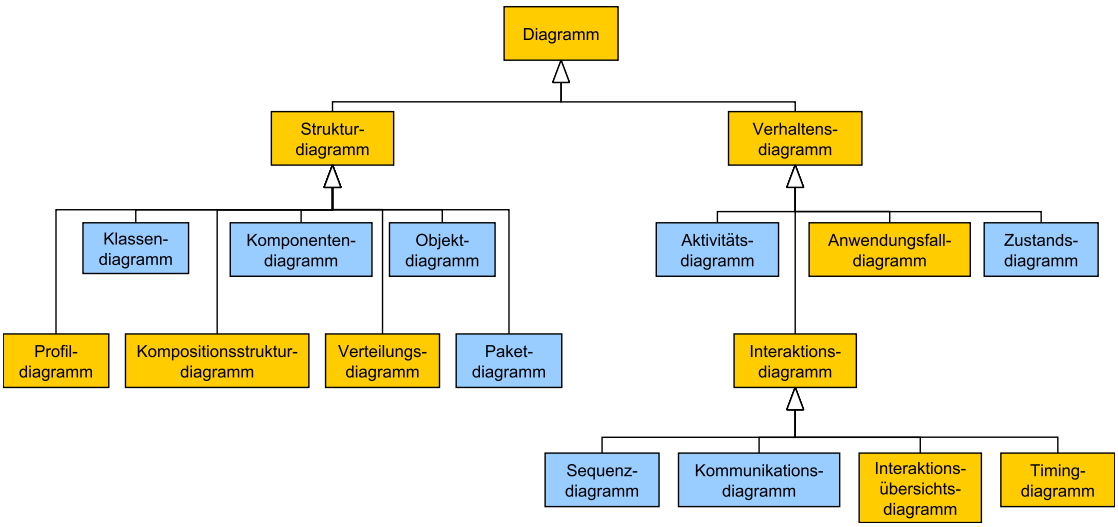


Abbildung 2: Hierarchie der Diagramme in UML 2.4 [54]

Abbildung 2 zeigt eine hierarchische Übersicht über sämtliche, momentan in der UML vorhandene Diagrammtypen. Man unterscheidet in diesen typischerweise die Struktur- von den Verhaltensdiagrammen, wobei Erstere die Statik eines Softwaresystems (u. A. Klassen inklusive deren Methoden und Attribute) modellieren und Letztere die Dynamik (d. h. konkrete Programmausführungen sowie Zustände) visualisieren. Innerhalb der Verhaltensdiagramme unterscheidet man zusätzlich die Gruppe der Interaktionsdiagramme, welche das Verhalten einer Anwendung auf Basis der Interaktionsvorgänge der zur Laufzeit existierenden Objekte darstellen.
Das Einsatzgebiet der UML beschränkt sich jedoch nicht nur auf Softwaresysteme allein. Einige Diagrammtypen wie beispielsweise Aktivitätsdiagramme oder Zustandsdiagramme sind so universell definiert, dass man mit diesen auch komplexe alltägliche Tätigkeiten der realen Welt wie z. B. das Befolgen einer Bedienungsanleitung beschreiben kann.
Im Folgenden möchten wir einige gängige UML-Diagrammtypen näher betrachten [54]. Wir beschränken uns dabei auf die Präsentation derjenigen Darstellungen mit niedrigem Abstraktionsniveau, da unsere eigene spätere Visualisierung ebenfalls eine solche implementierungsnahe Sicht auf die Statik und Dynamik einer Anwendung bieten soll. Die von uns vorgestellten Diagramme sind in Abbildung 2 blau eingefärbt.

Klassendiagramme *Klassendiagramme* stellen einen sehr quellcodenahen Diagrammtyp dar und visualisieren die Statik eines betrachteten Softwaresystems basierend auf den Grundstrukturen einer jeden objektorientierten Anwendung: Klassen und deren gegenseitige Beziehungen. Abbildung 3 zeigt ein Beispiel eines Klassendiagramms, bei welchem wir das bekannte Dinierende-Philosophen-Problem [26] beispielhaft modelliert haben.

Klassen sind als rechteckige Knoten visualisiert und ihr Bezeichnung sowie ihre Attribute und Methoden in dieser Reihenfolge von oben nach unten in diesen Knoten notiert. Attribute sind mit ihrer Bezeichnung und Sichtbarkeit sowie Typ angegeben. Methoden sind hingegen mit ihrer Signatur notiert. Klassen können dabei durch Kanten verbunden sein. Unausgefüllte dreieckige Pfeilspitzen symbolisieren eine Generalisierungsbeziehung, wohingegen viereckige Pfeilspitzen eine Kompositions-, bzw. Aggregationsbeziehung codieren. Optionale *Multiplizitäten* ergänzen diesen Diagrammtyp und geben das quantitative Verhältnis der Klasseninstanzen zur Laufzeit an. Sie werden jeweils an den Enden einer Kante notiert.
Klassendiagramme unterstützen den Entwickler typischerweise bei einer Vielzahl von Aufgaben. Sie eignen sich beispielsweise für Modellierung der Statik eines geplanten Softwareprojekts, aber auch der Dokumentation bereits bestehender Anwendungen.

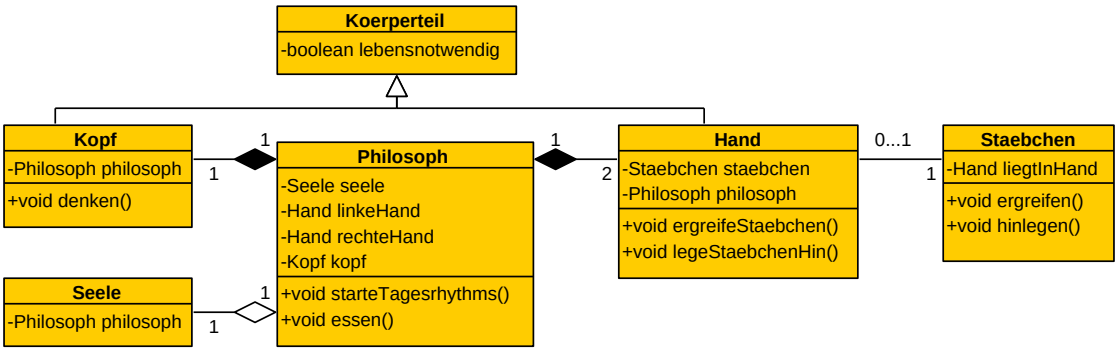


Abbildung 3: Ein beispielhaftes Klassendiagramm der dinierenden Philosophen

Objektdiagramme *Objektdiagramme* visualisieren den Zustand einer Auswahl von Objekten des laufenden Softwaresystems zu einem beliebigen Zeitpunkt der Applikationsausführung. Sie zeigen dabei optisch eine große Ähnlichkeit zu den Klassendiagrammen. Rechtecke repräsentieren einzelne Objekte mit ihrer Bezeichnung und Klasse. Attribute sind jeweils mit ihrem aktuellen Wert angegeben. Da die Anzahl der Attribute bei einigen Klassen jedoch besonders groß werden kann, erlaubt die UML-Spezifikation [54] auch die Darstellung einer Teilmenge dieser Attribute. Dies ermöglicht die Fokussierung auf die relevanten Informationen in einem betrachteten Szenario. Assoziation-, Aggregations- und Kompositionsbeziehungen werden bei diesem Diagrammtyp ohne Multiplizitäten aus dem Klassendiagramm übernommen und repräsentieren die Zusammengehörigkeit der so verbundenen Objekte. Häufig verwendet man Objektdiagramme, um bestimmte Testfälle zu notieren oder Beispiele für Attributbelegungen zu liefern. Abbildung 4 zeigt ein Beispiel, bei welchem wir unser Modell der dinierenden Philosophen wieder aufgegriffen haben.

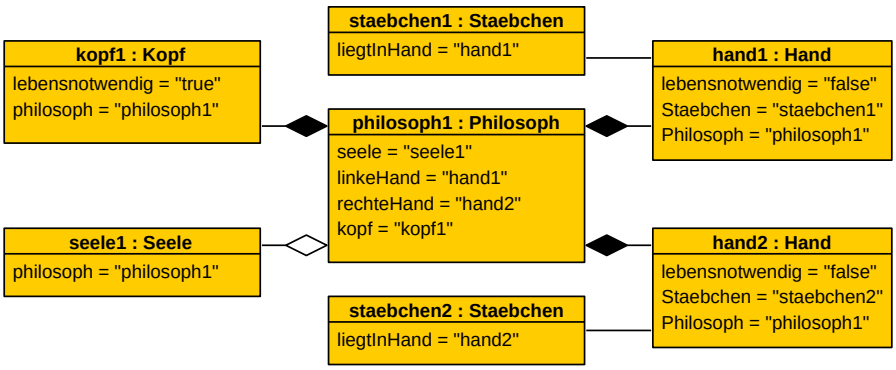


Abbildung 4: Ein beispielhaftes Objektdiagramm passend zu Abbildung 3

Sequenzdiagramme *Sequenzdiagramme* zeigen die Art und Abfolge derjenigen Nachrichten, welche zwischen Objektinstanzen und Akteuren zur Laufzeit bei einer Sequenz von Methodenaufrufen ausgetauscht werden. Ein Sequenzdiagramm realisiert dabei ein zweidimensionales Koordinatensystem. In Richtung der y-Achse sind die am Nachrichtenprozess beteiligten Teilnehmer angeordnet, wohingegen die invertierte x-Achse den logischen Zeitpunkt des Nachrichtenaustausches ordnet. Ein rechteckiger Knoten mit Angabe der Klasse und Objektbezeichnung (Letzteres optional) repräsentiert einen Kommunikationsteilnehmer, dessen Existenz durch eine adjazente, nach unten gerichtete gestrichelte Linie (der sogenannten *Lebenslinie*) symbolisiert wird. Ein Kreuz am Ende einer dieser Linien zeigt die Eliminierung des zugehörigen Objekts aus dem Speicher.
Nachrichten werden in asynchrone und synchrone Methodenaufrufe (Letztere mit und ohne Rücknachrichten) unterschieden und durch gerichtete Kanten zwischen den Lebenslinien zweier Objekte dargestellt. Jeder derartigen Verbindung muss dabei einer Assoziationsbeziehung im Klassendiagramm gegenüberstehen. Ausgefüllte Pfeilspitzen visualisieren synchrone Funktionsaufrufe, wohingegen unausgefüllte Spitzen asynchrone Aufrufe repräsentieren. Gestrichelte Pfeile symbolisieren demgegenüber eine Rückantwort. Nachrichten sind dabei durch die Signatur ihres Methodenaufrufes gekennzeichnet und auch auf diese Weise in der Darstellung angegeben. Zusammenhängende Methodenaufrufe können dabei zu *Fragmenten* unterschiedlichen Typs zusammengefasst werden, welche durch einen umschließenden Kasten symbolisiert werden, und beschreiben Eigenschaften, welche für alle enthaltenen Nachrichten gelten. Auf diese Weise lassen sich beispielsweise wiederkehrende Methodenaufrufe in Form von Schleifenausführungen, nebenläufige Aufrufe oder kritische Sektionen darstellen. Die Visualisierung von Synchronisationsmechanismen ist jedoch mit Fragmenten aktuell nicht möglich. Abbildung 6 zeigt ein Beispiel eines Sequenzdiagramms, welches einen dinierenden Philosophen bis zum Aufheben des ersten Stäbchens zeigt.

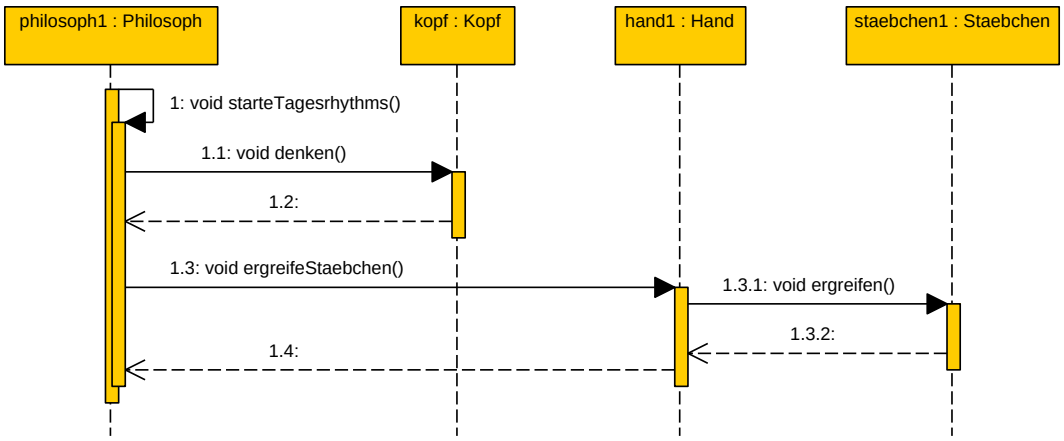


Abbildung 5: Ein beispielhaftes Sequenzdiagramms passend zu Abbildung 3

Kommunikationsdiagramme *Kommunikationsdiagramme* (in früheren UML-Versionen *Kollaborationsdiagramme* genannt) zeigen eine große Ähnlichkeit zu den Sequenzdiagrammen und visualisieren konzeptionell die gleichen Informationen auf eine unterschiedliche Weise. Kommunikationsdiagramme verzichten auf die im Sequenzdiagramm dargestellten gestrichelten Lebenslinien. Stattdessen findet eine Abstraktion der logischen Zeit hin zu der Reihenfolge der Methodenaufrufe statt, welche durch eine vorangestellte fortlaufende Nummer, der sogenannten *Sequenznummer*, vor einer Methodensignatur codiert wird. Kleine schwarze Pfeile symbolisieren Methodenaufrufe und verlaufen entlang Kanten, welche die Knoten verbinden. Die Form der Pfeilspitzen wird dabei aus den Sequenzdiagrammen übernommen. Ein zu Abbildung 5 äquivalentes Kommunikationsdiagramm zeigt Abbildung 6.
Kommunikationsdiagramme bieten gegenüber Sequenzdiagrammen mehrere Vorteile. Auf der einen Seite sind sie in der Regel platzsparender, da in diesen auf eine Dimension der Darstellung verzichtet wird und die Elemente daher frei im zweidimensionalen Raum angeordnet werden können. Auf der anderen Seite liefern diese auch eine leichter erkennbare Integration der Statik, da jeder Kante im Kommunikationsdiagramm eine Assoziationsbeziehung im Klassendiagramm gegenüberstehen muss.

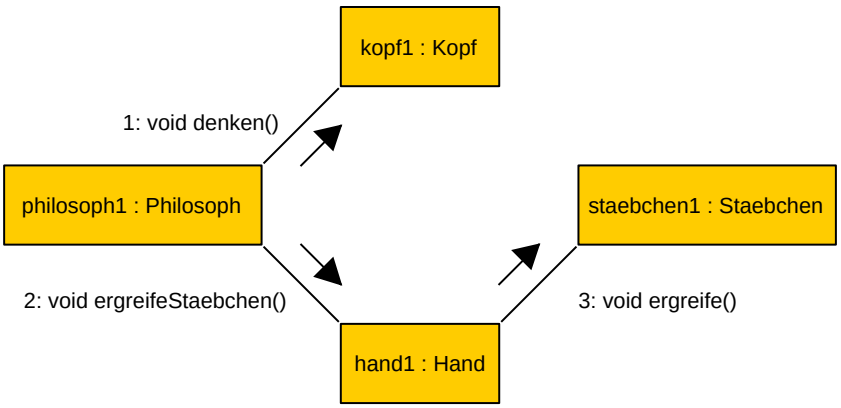


Abbildung 6: Ein zu Abbildung 5 äquivalentes Kommunikationsdiagramm

Weitere Diagrammtypen Neben den oben vorgestellten Diagrammtypen existieren in der UML noch weitere Sichten, welche bei der Entwicklung unsere eigenen Visualisierung in Frage kommen. Wir möchten uns jedoch auf die Beschreibung der wesentlichen Diagrammtypen beschränken. Daher erläutern wir diese Diagramme nachfolgend nur kurz.

- *Paketdiagramme* stellen die Paketstruktur einer Anwendung inklusive deren gegenseitige Abhängigkeitsbeziehungen (z. B. Import oder Verwendung) dar. Wie für die UML typisch, findet eine graphenbasierte Visualisierung statt. Pakete werden dabei als Knoten dargestellt und Abhängigkeiten durch gerichtete Kanten repräsentiert.

- *Zustandsdiagramme* zeigen die internen Zustände, welche ein Objekt, Komponente oder (Teil-)System zur Laufzeit der Anwendung annehmen kann. Zu jedem Zeitpunkt darf dabei nur ein einziger Zustand aktiv sein. Zustandsübergänge finden bei Erfüllung der im Diagramm vermerkten Bedingungen statt, in deren Zuge optional definierte Aktionen ausgelöst werden. Hierbei repräsentieren Knoten Zustände. Mögliche Transitionen werden hingegen durch adjazente, gerichteten Kanten dargestellt.

- *Komponentendiagramme* zeigen die makroskopische Untergliederung eines Softwaresystems in einzelne Komponenten inklusive deren gegenseitige Abhängigkeitsbeziehungen und Schnittstellen. Knoten zeigen Komponenten. Eine simple Symbolik der Kanten visualisiert Schnittstellen und deren Verwendung.

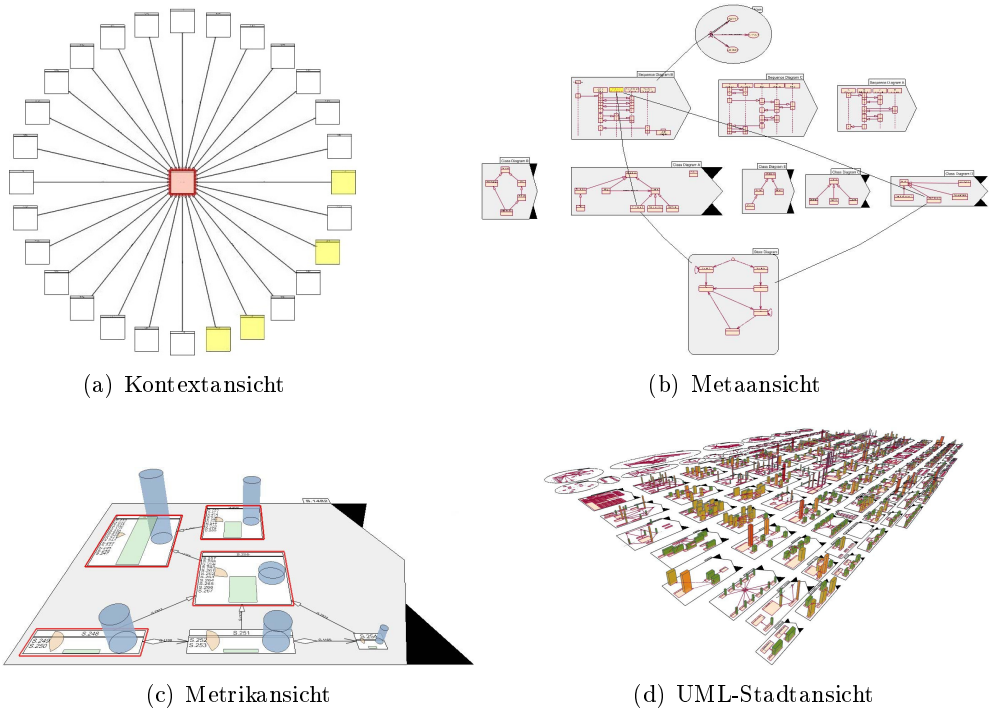
- *Aktivitätsdiagramme* visualisieren den Ablauf eines Vorgangs oder Algorithmus durch Untergliederung in Teilschritte, welche mittels Knoten repräsentiert werden. Kanten zwischen diesen zeigen jeweils die Folgeaktion an. Auch alternative oder parallele Arbeitsschritte können mittels einer entsprechenden Symbolik dargestellt werden.

3.1.2. Erweiterungen nach Lange und Chaudron

Ein großer Nachteil der UML liegt in der fehlenden Kombinierbarkeit der einzelnen Diagrammtypen. Zwar zeigt die UML ein System aus verschiedenen Perspektiven, eine ganzheitliche und detaillierte Visualisierung des Gesamtsystems bietet sie jedoch nicht. So existiert beispielsweise kein Diagrammtyp, welcher zu einer Klasse sämtliche assoziierte und abgeleitete Klassen inklusive der sie umschließenden Pakete, Komponenten und Subsysteme gemeinsam darstellt. Lange und Chaudron [39, 40, 41] greifen diesen Makel auf und präsentieren eine Reihe von Visualisierungen, welche verschiedene UML-Diagrammtypen zu einem einzigen Gesamtschaubild kombinieren, bzw. diese durch Hinzufügen von dreidimensionalen Objekten in der Menge darstellbarer Informationen erweitern. Die von ihnen entwickelte Anwendung *MetricView Evolution* [11] implementiert ein Großteil dieser Sichten, von denen wir die vier Kerndiagramme im Folgenden näher vorstellen werden.

Kontextansicht Der *Kontext* eines Elements bezeichnet sämtliche Elemente, welche mit diesem aus statischer Sicht in Beziehung stehen. Bei einem Paket sind dies beispielsweise sämtliche enthaltenen Klassen, Sub- und Oberpakete sowie die Komponente und das (Sub-)System, welches dieses Paket beinhaltet. In der UML sind diese Daten jedoch über mehrere Diagramme verteilt. Ersteres zeigt sich z. B. in Klassendiagrammen, Letzteres in Komponentendiagrammen. Die *Kontextansicht* vereint all diese zu einem Objekt in Beziehung stehende Elemente in einer einzelnen Visualisierung. Sie zeigt dabei das Bezugsobjekt im Zentrum des Diagramms und platziert sämtliche mit diesem assoziierte oder von ihm abgeleitete Elemente kreisförmig um dieses herum. Abbildung 7(a) zeigt ein Beispiel dieser Ansicht, bei welchem die Abhängigkeiten einer beispielhaften Klasse visualisiert wurden. Der Ansatz beinhaltet darüber hinaus auch eine farbliche Markierungsfunktion. Alle abgeleiteten Klassen des Bezugsobjekts sind hier gelb eingefärbt.

Metaansicht UML-Diagramme zeigen Sichten auf eine Anwendung mit unterschiedlichem Abstraktionsniveau. Auf höchstem Niveau lassen sich beispielsweise Anwendungsfalldiagramme anordnen, welche Benutzerinteraktionen mit der Software visualisieren. Dieser Vorgang lässt sich jedoch weiter mittels Sequenzdiagrammen konkretisieren, welche die Softwarebenutzung in Methodenaufrufe aufschlüsselt. Sequenzdiagramme hingegen beinhalten Objekte, welche auf tiefster Abstraktionsschicht durch Klassendiagramme näher spezifiziert werden können. Deren internes Verhalten lässt sich wiederum mittels Zustandsdiagramme darstellen. Die *Metaansicht* vereint all diese Diagramme in einem einzigen Schaubild, indem sie gegenseitige Abhängigkeiten zwischen den Elementen einzelner Diagrammtypen mittels adjazente Kanten darstellt. Abbildung 7(b) zeigt ein Beispiel einer solchen Darstellung, welches die oben genannten UML-Diagramme beinhaltet.



(a) Kontextansicht (b) Metaansicht

(c) Metrikansicht (d) UML-Stadtansicht

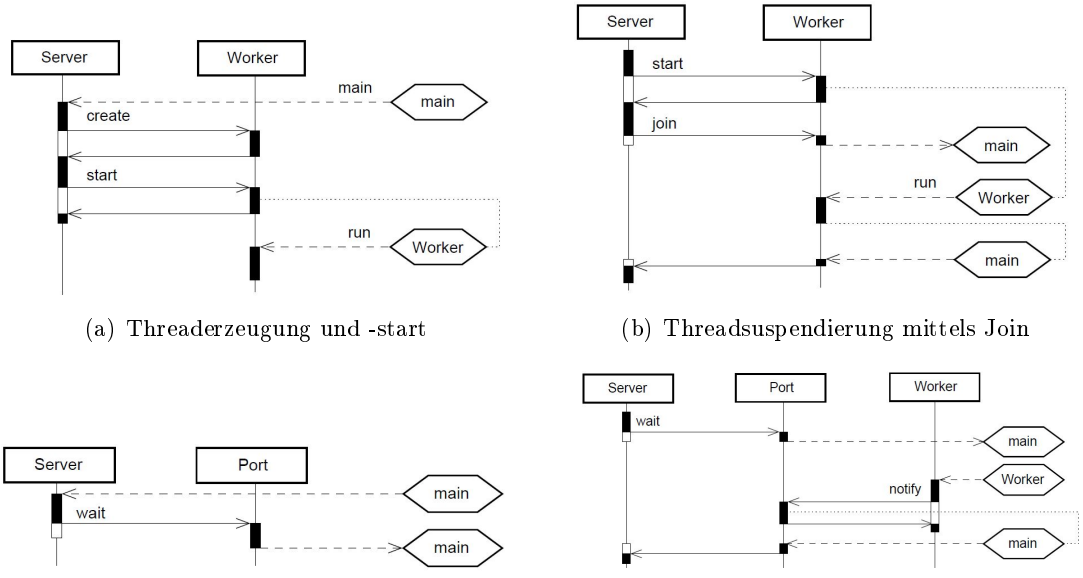
Abbildung 7: Vier Ansichten der UML-Erweiterung von Lange und Chaudron [41]

Metrikansicht Die *Metrikansicht* erweitert UML-Diagramme um beliebige zwei- und dreidimensionale Elemente. Geometrische Strukturen wie z. B. Quader, Säulen oder Rechtecke werden dabei direkt auf der zweidimensionalen Grundfläche platziert und erlauben die Darstellung beliebiger benutzerdefinierter, zusätzlicher Informationen und Metriken. Im Fall von Klassendiagrammen könnten auf diese Weise beispielsweise die Anzahl der Zeilen im Code oder auch die Menge der Methoden innerhalb der betrachteten Klasse zusätzlich in das Schaubild integriert werden. Die Form, Farbe und Gestalt der Objekte gibt dabei die Merkmalsausprägung bezüglich der betrachteten Metrik an. Ein Beispiel ist in Abbildung 7(c) gegeben, in welcher drei verschiedene Eigenschaften einer Klasse mittels geometrischer Objekte dargestellt wurden.

UML-Stadtansicht Abbildung 7(d) zeigt ein Beispiel der UML-Stadtansicht. Diese Form der Darstellung kombiniert die Meta- und Metrikansicht zu einer einzigen Visualisierung. Eine dreidimensionale Darstellung ist entstanden, welche zwar stark der städtischen Metapher ähnelt, jedoch diese bei genauerer Betrachtung nicht wirklich implementiert. Die städtische Metapher werden wir in Abschnitt 3.2 jedoch näher besprechen, sodass wir hier auf dessen Beschreibung verzichten möchten.

3.1.3. Erweiterungen nach Artho, Havelund und Honiden

Auch trotz der umfangreichen Visualisierungsmöglichkeiten der UML unterstützen dessen Diagramme bislang kaum die Darstellung der Dynamik nebenläufiger Applikationen. Sequenzdiagramme zeigen beispielsweise die zeitlichen Abläufe von Methodenaufrufen, jedoch sind diese nicht im Stande, nebenläufige Vorgänge wie Context-Switches oder Synchronisationsabläufe geeignet zu visualisieren. Artho et al. [16] präsentieren eine Erweiterung der UML, mittels welcher nebenläufige Phänomene wie Context-Switches, Thread-Joins sowie der Aufruf von Wait- und Notify-Operationen in Sequenzdiagrammen dargestellt werden können. Abbildung 8 zeigt eine Übersicht über diese Erweiterungen. Ihre Ideen basieren dabei auf den Arbeiten von Li et al. [47], bzw. erweitern diese, welche wir aus diesem Grund nicht weiter vorstellen möchten.
Abbildung 8(a) zeigt die Grundelemente ihrer Visualisierung am Beispiel der Erzeugung eines neuen Threads und dessen Aktivierung durch den Scheduler. Artho et al. unterscheiden dabei zwischen der Datenstruktur eines Threads, d. h. dessen Objekt im Speicher, und dem ausführbaren Programm, d. h. dessen Abfolge von Methoden- und Instruktionsausführungen. Ersteres wird wie jedes andere Objekt in der horizontalen Dimension eines Sequenzdiagramms dargestellt, wohingegen Letzteres mittels eines Hexagons an der rechten Seite des Schaubilds visualisiert wird. Ein gestrichelter Pfeil von diesem zu seiner zugehörigen Lebenslinie symbolisiert einen Context-Switch und daher letztendlich die



(a) Threaderzeugung und -start

(b) Threadsuspendierung mittels Join

(c) Threadsuspendierung mittels Wait

(d) Threadreaktivierung mittels Notify

Abbildung 8: Erweiterungen der UML-Sequenzdiagramme nach Artho et al. [16]

(weitere) Ausführung des Threads. Dies reicht jedoch nicht aus, um die nebenläufigen Abhängigkeiten zwischen einer Menge von Threads zu verstehen, da beispielsweise im Falle der Visualisierung eines Programmtraces unklar bleibt, ob ein Context-Switch an einer Stelle stets erforderlich ist oder lediglich zufällig durch den Scheduler verursacht wurde. Artho et al. führen aus diesem Grunde zusätzliche gepunktete Linien zwischen einer Lebenslinie und der hexagonen Repräsentation eines Thread in die Visualisierung ein, welche Ereignisabhängigkeiten gemäß einer *Passiert-zuvor*-Relation repräsentieren. Deren Bedeutung ist dabei relativ simpel: Die adjazente Methode muss zunächst ausgeführt werden, bevor der referenzierte Thread nach einem Context-Switch fortfahren kann.
Im angegebenen Beispiel verbleibt der Kontrollfluss des Hauptprogramms (*main*) zunächst beim Server, welcher einen Arbeiterthread instantiiert. Erst nach dessen Erzeugung und Start kann der Scheduler einen Context-Switch durchführen und den neuen Thread ausführen.
Sämtliche übrigen Elemente aus den Sequenzdiagrammen sind in diesem Visualisierungsansatz erhalten geblieben. So zeigen wie gewöhnlich zwischen den Lebenslinien verlaufende Pfeile Methodenaufrufe und ausgefüllte Balken deuten die Berechnungsschritte innerhalb einer Methode an. Mit Hilfe dieser Erweiterungen lassen sich nun auch Synchronisationsmechanismen in Sequenzdiagrammen realisieren. Abbildung 8(b) zeigt die Suspendierung eines Threads durch Aufruf der Join-Methode. Der Hauptthread wird suspendiert und wartet auf die Terminierung des Arbeiterthreads. Da ein Thread-Join mittels Methodenaufrufs auf dem Speicherobjekt des Threads in der Java-Programmierung realisiert wird, ist auch hier der Join zunächst als reguläre Funktion, d. h. als Pfeil zwischen zwei Lebenslinien dargestellt. Die Methode kann jedoch zunächst nicht terminieren und gibt den Kontrollfluss vorübergehend ab. Eine Suspendierung des Threads findet statt, welche in der Visualisierung durch einen auf das Hexagon gerichteten gestrichelten Pfeil symbolisiert wird. Nach Terminierung des Arbeiterthreads gelangt der Kontrollfluss wieder zurück zum Hauptthread, welcher mit seiner eigenen Berechnung fortfahren kann.
Abbildung 8(c) und 8(d) zeigen demgegenüber die Realisation der Wait-/Notify-Operationen. Der Hauptthread ruft die Wait-Methode des *Port*-Objekts auf, wird daraufhin suspendiert und gibt den Kontrollfluss ab. Der Scheduler (re-)aktiviert nachfolgend den Arbeiterthread, welcher die Notify-Methode des *Port*-Objekts startet. Ein Context-Switch findet statt, wodurch der Hauptthread reaktiviert wird und mit seinen Ausführungen fortführt.

3.1.4. Erweiterungen nach Mehner und Wagner

Die Erweiterungen von Artho et al. ermöglichen zwar mit der Darstellung von Context-Switches und Wait-/Notify- sowie Thread-Join-Operationen die Visualisierung von vergleichsweise speziellen und z.T. Java-spezifischen Synchronisationsmechanismen, jedoch existiert mit damit immer noch kein Konzept zum Umgang mit Semaphoren und Monitoren. Mehner und Wagner [52] haben sich diesem Problem angenommen und präsentieren eine eigene Erweiterung der UML zur Visualisierung von Synchronized-Blöcken in Kommunikationsdiagrammen. Da diese Forscher das Betreten und Verlassen von synchronisierten Methoden als Belegen und Freigeben von Semaphoren oder Monitoren verstehen, lassen sich deren Konzepte auch ohne große Anpassungen direkt auf das allgemeine Semaphor- und Monitor-Konzept übertragen.

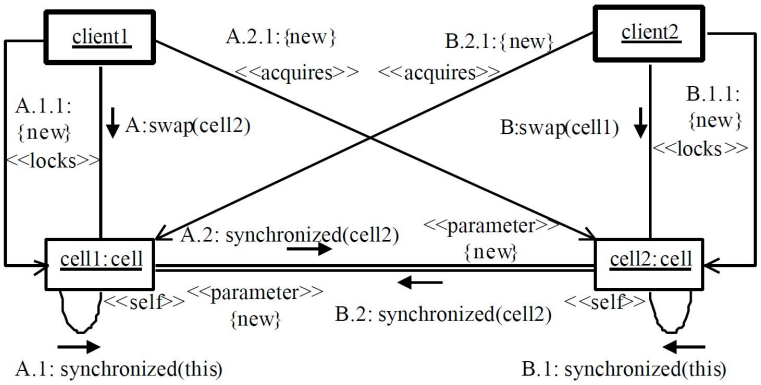


Abbildung 9: Ein Kommunikationsdiagramm mit den Erweiterungen von Mehner und Wagner [52]

Abbildung 9 zeigt ein Beispiel eines auf ihre Weise erweiterten Kommunikationsdiagramms, welches den gleichzeitigen nebenläufigen Aufruf der *swap*-Methode in zwei Threads dargestellt und letztendlich einen Deadlock zeigt. Diese Methode synchronisiert sich dabei zunächst auf die eigene, anschließend auf die per Parameter übergebene *Cell*-Instanz. Der Synchronisierungsvorgang wird dabei von den Autoren als eigenständiger Methodenaufruf verstanden und ist auch derartig in dem Kommunikationsdiagramm vermerkt. So findet zuerst ein Prozeduraufruf auf die eigene *Cell*-Instanz, angedeutet durch den Stereotyp *«self»* inklusive dem Aufruf *synchronized(this)*, und im Anschluss der Aufruf auf die Fremdinstanz statt. Da einer Verbindung im Kommunikationsdiagramm einer Assoziationsbeziehung im Klassendiagramm gegenüberstehen muss, die Objektreferenz im letzteren Fall jedoch in Form eines Parameters übergeben wird, ergänzen die Autoren den Aufruf im Diagramm um den Stereotyp *«parameter»* mit Einschränkung *{new}*. Dieser besagt, dass die Verbindungen zwischen den beiden *Cell*-Instanzen lediglich aufgrund

von Parameterübergaben im Zuge der Ausführung der *swap*-Methode entstanden sind. Da das Betreten von synchronisierten Methoden als Belegen von Semaphoren, d. h. als Relation zwischen belegendem und belegtem Objekt verstanden wird, fügen die Autoren zudem die selbst definierten Stereotypen *«locks»* und *«aquires»* jeweils mit Sequenznummern ein. Die Sequenznummern von nebenläufigen Threads werden hierbei um vorangestellten Großbuchstaben (hier A und B) ergänzt. So werden sämtliche Aufrufe, welche mit dem selben Buchstabenpräfix notiert sind, von ein und dem selben Thread ausgeführt. Dies erlaubt dabei keine globale sondern lediglich eine Thread lokale Ordnung der Prozeduraufrufe. Der Stereotyp *«locks»* besagt, dass ein Thread eine Semaphore erfolgreich belegt hat, wohingegen *«aquires»* das Warten auf die Belegung einer Semaphore symbolisiert.
Mit Hilfe dieser Erweiterungen lassen sich nun Deadlocks in einem Sequenzdiagramm identifizieren und analysieren. In unserem Beispiel passieren zwar beide Threads die Semaphore auf dasjenige Objekt, auf welches die *swap*-Methode aufgerufen wurde (*cell1*, bzw. *cell2*), jedoch blockieren beide anschließend auf den «aquires»-Verbindungen und damit auf den nicht terminierenden *synchronized(cell1)*- und *synchronized(cell2)*-Aufrufen.

3.1.5. Erweiterungen nach Malnati et al.

Malnati et al. [48, 49] stellen einen alternativen Ansatz zur Darstellung von Threadsynchronisationen auf Basis von Monitor-, bzw. Semaphor-Operationen vor. Ihre Softwareanwendung *JThreadSpy* verwendet eine Erweiterung der UML-Sequenzdiagramme zur Visualisierung, von welchen Abbildung 10 ein Beispiel zeigt. Dieses zeigt dabei den nebenläufigen Aufruf mehrerer Methoden durch zwei Threads, welche letztendlich in einen Deadlock laufen.
Jedem Thread wird im Vorfeld der Visualisierung eine eindeutige Farbe zugeordnet, in welcher seine Methodenausführungen gefärbt werden. Aufrufe, welche direkt von einem Threadobjekt (d. h. innerhalb der *run*-Methode), bzw. von der Hauptprogramminstanz (d. h. durch die *main*-Methode) ausgelöst werden, werden dabei in Form eines von links kommenden Pfeils mit doppelten Wellensymbol symbolisiert. Geht mit einem Funktionsaufruf das Belegen einer Sperrvariable einher, so wird der Aufrufpfeil durch das Symbol eines Vorhängeschlosses ergänzt. Dieser bedeutet, dass der Thread sich auf das aufgerufene Objekt im Sinne der *synchronized*-Operation synchronisiert.
Die Visualisierung unterscheidet nun zwei Fälle. Hat der Thread die Sperrvariable erfolgreich genommen, wird dies durch einen breiteten Streifen im Methodenausführung für die Zeit der Monitorbelegung symbolisiert. Bewirbt sich der Thread jedoch erst um die entsprechende Sperrvariable, so codiert dies eine gestrichelt/gepunktete Linie am linken Rand der Funktionsausführung.

Optional erlaubt JThreadSpy ebenso die Einblendung der jeweils aktuellen Systemzeit bei jedem Methodenein-, bzw. austritt. Am linken Rand des Schaubildes wird diese auf Höhe jeder Nachricht zwischen den beiden jeweils beteiligten Laufzeitobjekten abgetragen. Auf diese Weise lässt sich leicht die vergangene Zeit zwischen zwei beliebigen Methodenaufrufen berechnen und daher deren Ausführungszeit ermitteln.

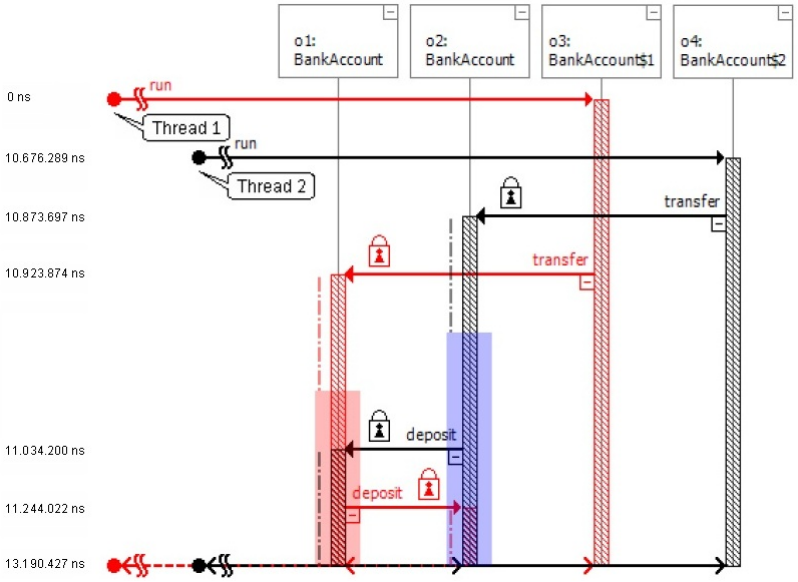


Abbildung 10: Ein Sequenzdiagramm mit den Erweiterungen von Malnati et al. [49]

3.1.6. Jacot

Zum Zeitpunkt dieser Arbeit existieren relativ viele Softwarewerkzeuge, welche Programmausführungen auf Basis der nicht erweiterten UML (v.A. mittels Sequenzdiagrammen) visualisieren [42, 43, 51]. Das *Java Concurrent Object Tool* (Jacot) [44, 45, 46] unterscheidet sich jedoch von den übrigen Applikationen dahingehend, dass in dieser auch eine Visualisierung der Threadzustände stattfindet, welche die Applikation in Form von Zustandsdiagrammen darbietet. Jedem Thread wird hierbei eine eindeutige Farbe sowie zur leichteren Identifizierung eine eindeutige Nummer zugewiesen. Jacot visualisiert die Dynamik mittels Sequenzdiagrammen, in welchen jeder Pfeil, der einen Methodenaufruf symbolisiert, in der Farbe seines ausführenden Threads gefärbt wird. Auf diese Weise lässt sich relativ leicht der gesamte Aufrufgraph eines Threads unmittelbar identifizieren. Zur Visualisierung von Threadzuständen wird zusätzlich ein Zustandsdiagramm verwendet, welches jedoch nur geringe Ähnlichkeiten mit dessen UML-Pendant aufweist. Ein Beispiel ist in Abbildung 11 dargestellt. Dieses Diagramm unterstützt dabei insgesamt sechs verschiedene Zustände, welche im Schaubild angegeben sind.

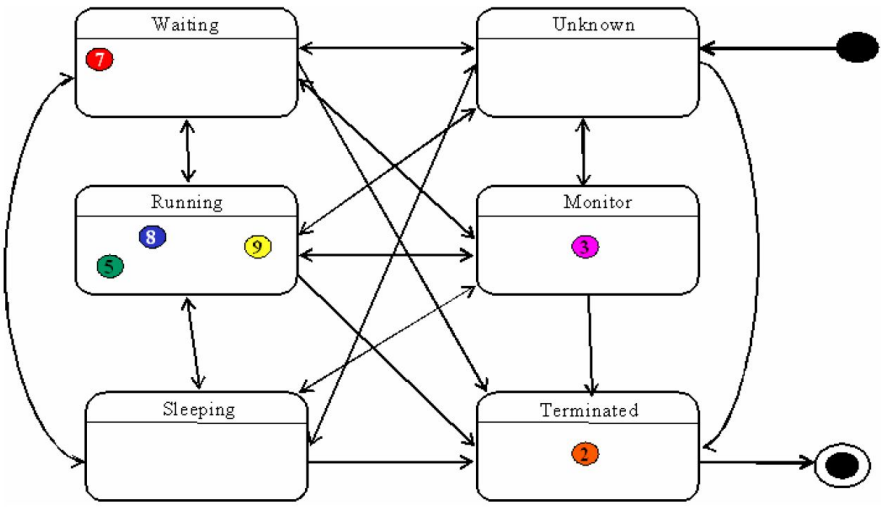


Abbildung 11: Ein Thread-Zustandsdiagramm aus Jacot [44]

Im Zuge der Programmausführung kann ein Thread in Java insgesamt sieben Zustände annehmen: *created, runnable, running, waiting, waiting on a monitor, sleeping* sowie *terminated* [44]. Die Entwickler haben sich jedoch gegen die Abbildung der beiden erstgenannten Zustände entschieden, da diese lediglich konzeptioneller Natur sind und Threads diese gewöhnlich nur kurzzeitig annehmen [46]. Gleichzeitig wurde mit dem *unknown*-Zustand ein weiterer Status hinzugefügt, mittels welcher auch die Abbildung einer Programmausführung ermöglicht wird, welche unter Anwendung des Java Virtual Machine Debug Interface (JVMDI) [7] stattfand. Diese Technologie versetzt einen Thread in den *unknown*-Zustand, falls dieser auf manuelle Weise vom Benutzer suspendiert wurde.
Die Visualisierung der Dynamik findet auf Basis eines hybriden Ansatzes mittels Sequenzdiagrammen statt. Jacot erlaubt zum Einen die Darstellung des terminierten Systems, d. h. die gleichzeitige Abbildung sämtlicher im Trace verfügbarer Methoden, zum Anderen ist auch eine iterative Visualisierung der zeitlichen Abfolge der Methodenaufrufen möglich. Das Sequenzdiagramm wird dabei inkrementell in Richtung der Zeitachse erweitert sowie Änderungen an den Threadzuständen im Zustandsdiagramm entsprechend übernommen. Eine automatisiere Identifikation von Deadlocks und Starvation rundet diese Applikation ab.

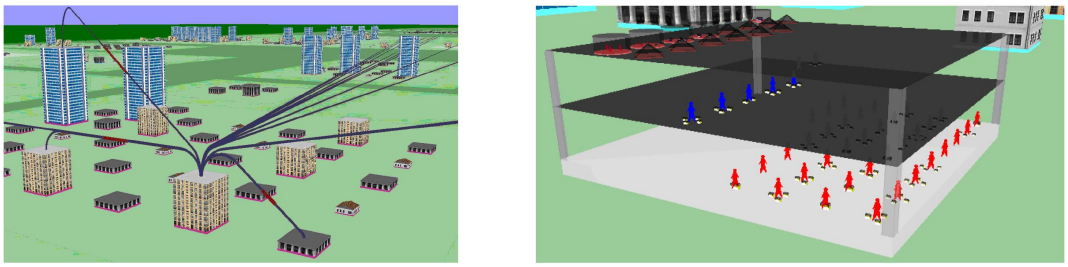
3.2. Städtische Metapher

Die *städtische Metapher* [24] (auch englisch *City-Metaphor* genannt) stellt einen speziellen und vergleichsweise markanten Ansatz zur Visualisierung komplexer Softwaresysteme dar. Sie basiert auf der dreidimensionalen Darstellung der Applikationsstatik und -dynamik in Form einer Großstadt und profitiert auf diese Weise von der Vertrautheit ihrer Betrachter mit dieser Struktur. Der tägliche Umgang mit den Gegebenheiten einer Stadt (z. B. Lesen von Straßenkarten, Orientierung an markanten Gebäuden und Wegpunkten) unterstützt den Betrachter dabei, sowohl die Statik als auch die Dynamik einer auf dieser Weise visualisierten Softwareanwendung zu verstehen [24, 63]. Die Komplexität einer Stadt bringt dabei Vorteile mit sich, die sie besonders für die Verwendung als Visualisierungsform für Softwaresysteme qualifiziert und die städtische Metapher von vielen anderen Darstellungsansätzen unterscheidet.
Eine Stadt lässt sich typischerweise als eine Menge zusammenstehender Gebäude beschreiben. In vielen Anwendungen der City-Metapher werden diese dazu verwendet, um die Klassen einer objektorientierten Anwendung zur repräsentieren [14, 38, 55, 62, 65]. Gebäude unterscheiden sich jedoch in einer Vielzahl charakteristischer Eigenschaften wie beispielsweise Form, Höhe, Farbgebung, Textur sowie Platzierung innerhalb der Stadt. Mittels dieser Dimensionen lassen sich eine Großzahl unterschiedlicher Eigenschaften einer Klasse in einer einzigen Darstellung integrieren [63]. Gleichzeitig zeigen sich auch Parallelen zwischen der Erkundungsstrategie einer Großstadt und dem Verstehensprozess von großen Softwaresystemen. So können Menschen Städte aufgrund ihrer Komplexität und all ihren Facetten (u. A. Sehenswürdigkeiten, Straßen und Gebäuden) oftmals lediglich inkrementell erkunden. Dies entspricht auch häufig dem Verhalten bei großen Applikationen. So beschränken sich Menschen in diesen meistens zunächst auf die Betrachtung einiger weniger Klassen und erweitern ihren Fokus nach und nach auch auf umliegende Strukturen [62].
Nachfolgend möchten wir zwei verschiedene bestehende Softwarewerkzeuge vorstellen, welche die städtische Metapher als Visualisierungsgrundlage verwenden.

3.2.1. EvoSpaces

S. Alam und P. Dugerdil [13, 14, 15, 27] kombinieren die Statik einer objektorientierten Anwendung und einen gegebenen Programmtrace in einer einzigen, auf der städtischen Metapher basierenden Visualisierung. Ihr Softwarewerkzeug *EvoSpaces* repräsentiert objektorientierte Klassen in Form von Gebäuden unterschiedlicher Höhe und Form sowie Pakete mittels (z.T. ineinander verschachtelter) Distrikte. Gebäude sind dabei in drei verschiedenen Typen (Häuser, Apartmentblöcke und Bürogebäude) mit jeweils unterschiedlicher Textur vorhanden. Mittels dieser Diversität lassen sich unterschiedliche Metriken bezüglich der Anwendungsstatik visualisieren. Am Beispiel der Zeilen an Code, welche das Programm standardmäßig verwendet, werden Häuser dazu verwendet, um besonders

kleine Klassen zu symbolisieren, Apartmentblocks hingegen, um solche mittlerer Größe zu repräsentieren sowie Bürogebäude, um sehr umfangreiche Klassen zu symbolisieren. Jeder Gebäudetyp liegt darüber hinaus in drei verschiedenen Höhen vor. Auf diese Weise lässt sich auch eine zusätzliche zweite Metrik (z. B. die Anzahl der Attribute einer Klasse) unmittelbar integrieren. Die Entwickler haben sich dabei gegen eine bijektive Abbildung der Metrik auf die Höhe eines Gebäudes entschieden, da es vielen Menschen im dreidimensionalen Raum oftmals sehr schwer fällt, leichte Größenunterschiede zu erkennen [27]. Der Typ und die Höhe eines Gebäudes sind dabei nicht fest an vordefinierte Metriken gebunden, sondern lassen sich durch den Anwender individuell festlegen. Abbildung 12(a) zeigt ein Ausschnitt dieser Visualisierung, in welcher unter Anderem auch die Form- und Größenunterschiede bei den Gebäuden gut erkennbar sind.



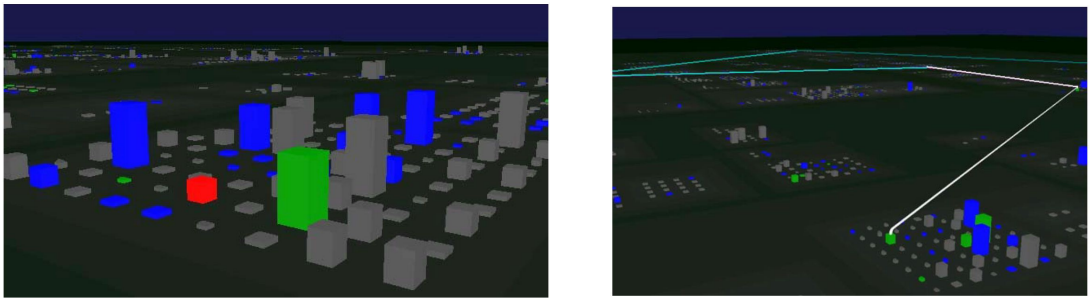
(a) Aussenansicht inklusive Relationen

(b) Methodendarstellung im Gebäudeinneren

Abbildung 12: Tagansicht in EvoSpaces [15]

Im Inneren von Gebäuden werden Methoden in Form von Arbeitern repräsentiert, welche auf mehrere Ebenen abhängig von ihrem Typ (z. B. Klassen- oder Objektmethode) angeordnet sind. Jeder Arbeiter ist dabei von gelben Kästchen umgeben, welche die lokalen Variablen der Methode symbolisieren. Die innere Modellierung eines Gebäudes ist in Abbildung 12(b)) dargestellt.
Als Grundfläche der Stadt dienen eine Reihe von ineinander verschachtelten, rechteckigen Flächen, welche die Pakethierarchie der Applikation repräsentieren. Da die Rechtecke dabei keine Höhe besitzen, ist eine insgesamt planare Grundfläche entstanden, wessen unterschiedliche Helligkeitsgebung die Verschachtelungstiefe der symbolisierten Pakete widerspiegelt. Jedes Gebäude wird dabei auf dasjenige Rechteck platziert, welches sein Paket im Quellcode symbolisiert.
Die Entwickler unterscheiden nun zwischen der *Tag-* und *Nachtansicht*. Erstere visualisiert die statischen Abhängigkeiten zwischen Klassen, wohingegen Letztere die Dynamik durch Wiedergabe des Programmtraces zeigen.

In der Tagansicht werden statische Relationen zwischen Objekten in Form von schwarzen geschwungenen Linien dargestellt, welche die Dächer zweier Häuser miteinander verbinden. Die Richtung des Informationsflusses wird durch ein rotes Segment symbolisiert, welches sich auf jeder Linie bewegt. Auf diese Weise lassen sich beispielsweise Ableitungs- oder Assoziationsbeziehungen visualisieren. Derartige Relationen sind auch in Abbildung 12(a) zu erkennen.
Die Nachtansicht hingegen dient der Darstellung des Programmtraces in Form von Methodenaufrufen und unterscheidet zwei alternative Visualisierungsmodi: Die *makroskopische* und *mikroskopische Ansicht*. Im Zuge der Visualisierung wird der gesamte Trace dabei zunächst in zusammenhängende Segmente von fester Länge untergliedert. In der makroskopischen Ansicht wird für jedes Segment eine Statistik über die involvierten Klassen und dessen Anzahl an Methodenausführungen und -Aufrufen erstellt. Auf Basis dieser Daten wird anschließend jede Klasse einer von drei Farben zugeordnet. Solche, welche sehr viele Prozeduren aufrufen erhalten beispielsweise die Farbe Rot, solche, die nur wenige Aufrufe tätigen hingegen Blau, und wiederum andere die Farbe Grün. In ihrer zugewiesenen Farbe wird nun auch ihr repräsentierendes Gebäude gefärbt. Einzelne Methodenaufrufe werden dabei nicht angezeigt. Abbildung 13(a) zeigt ein Beispiel dieser Visualisierung.



(a) Makroskopische Ansicht (b) Mikroskopische Ansicht

Abbildung 13: Nachtansicht in EvoSpaces [27]

EvoSpaces erlaubt nun unter Anderem die Darstellung sämtlicher Segmente des Programmtraces in Form einer interaktiven Animation, in welchem jedes Einzelbild einem Segment entspricht. Dabei wird die Farbgebung aller Gebäude eines Segments für eine feste Zeiteinheit eingeblendet und so fortführend iterativ der gesamte Trace abgearbeitet. Dies ermöglicht die Identifikation von besonders aktiven Klassen zu jedem Zeitpunkt der Ausführung.Die Mikroskopische Ansicht hingegen lenkt den Fokus der Visualisierung auf

nur ein einzelnes Segment, d. h. einen kleinen Ausschnitt des Programmtraces.
Methodenaufrufe werden mittels Verbindungslinien zwischen den beteiligten Klassen in angegebener Orientierung dargestellt, welche ähnlich zu denen sind, welche auch schon bei der Tagansicht zur Symbolisierung von statischen Relationen verwendet werden. Auch bei dieser Visualisierung ist eine Animation möglich, in welcher ein Einzelbild einem einzelnen Methodenaufruf entspricht. Pro Zeiteinheit wird also nur ein Methodenaufruf gleichzeitig dargestellt. Die Färbung der Gebäude wird dabei zusätzlich zur Visualisierung von Performance-Hotspots aus der makroskopischen Ansicht des Segments übernommen. Abbildung 13(b) zeigt einen Beispiel dieser Visualisierung.
Sämtliche Ansichten in EvoSpaces bieten dabei eine Vielzahl an Interaktionsmöglichkeiten. Der Benutzer kann beispielsweise jedes optische Element auswählen und über ein Kontextmenu die Werte einiger Metriken abfragen und sich den repräsentierten Quellcode eines Objekts anzeigen lassen. Mittels einer Zoom- und Rotationsfunktion kann sich der Anwender zudem frei im dreidimensionalen Raum bewegen und so den Blickwinkel auf die Darstellung nach eigenem Ermessen verändern.

3.2.2. DyVis

C. Wulf [65] entwickelte im Zuge seiner Bachelorarbeit das Visualisierungswerkzeug *Dynamic Visualizer* (kurz *DyVis*). Diese Anwendung stellt die Statik und Dynamik einer Softwareapplikation mit Fokus auf Letzterem dar und verwendet die städtischen Metapher. DyVis visualisiert objektorientierte Klassen in Form von Gebäuden, deren Etagen die Instanzen einer jeweiligen Klasse repräsentieren. Das Erdgeschoss symbolisiert dabei das Klassenobjekt selbst, die darüber liegenden Stockwerke hingegen visualisieren die dynamischen Laufzeitobjekte. Die Gebäude besitzen eine variabel große, quadratische Grundfläche, welche auf Basis der Anzahl der Attribute einer Klasse berechnet wird. Interfaces, abstrakte Klassen sowie leere Pakete bleiben dabei unvisualisiert, da diese für die Darstellung der Dynamik nicht relevant sind.
Für die Grundfläche verwendet Wulf ein ähnliches Konzept, welches auch schon EvoSpaces anwendet. Pakete werden in Form von übereinanderliegenden Rechtecken symbolisiert, welche im Gegensatz zu EvoSpaces eine feste Höhe besitzen. Die Grundfläche ist somit nicht vollständig eben. Stattdessen ist eine dreidimensionale Treppenstruktur entstanden, auf welcher die einzelnen Gebäude basierend auf ihrer Paketzugehörigkeit platziert werden. Auch hier lässt sich an der Helligkeit einer Ebene die Verschachtelungstiefe eines Pakets innerhalb des Programms erkennen.
Die Dynamik einer Applikation visualisiert DyVis in Form Kanten zwischen Etagen nicht notwendigerweise verschiedener Gebäude und hebt diese Stockwerke zur leichteren Identifikation farblich hervor. Jede Kante repräsentiert dabei einen Methodenaufruf, dessen Färbung auf Basis der Dauer der Methodenausführung bestimmt wird. Abbildung 14 zeigt ein Beispiel dieser Visualisierung.

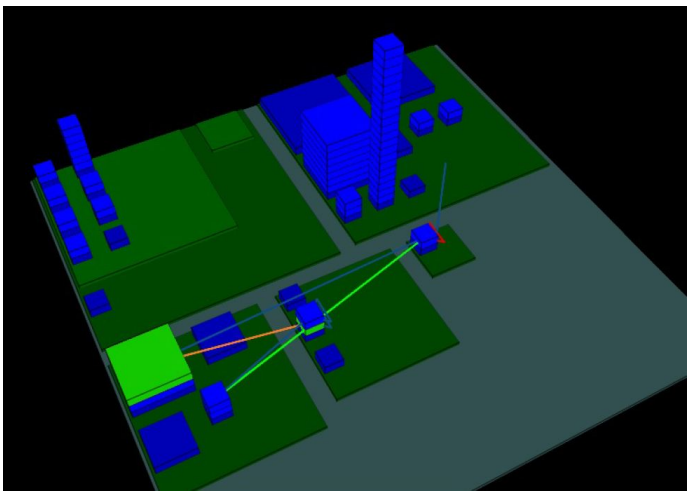


Abbildung 14: Ein Ausschnitt aus DyVis [65]

Bei der Wiedergabe eines Programmtraces bietet DyVis eine Vielzahl von Optionen. So lässt sich auf der einen Seite der gesamte Trace in Form einer Animation abspielen, auf der anderen Seite ist aber auch eine manuelle, schrittweise Wiedergabe möglich. Das freie Navigieren innerhalb des Programmtraces wird ebenfalls unterstützt. So erlaubt die Anwendung auch das Springen zu einzelnen Zeitstempeln im Trace. DyVis bietet dabei ähnliche Interaktionsmöglichkeiten wie EvoSpaces. So lässt sich beispielsweise jedes visuelle Element auswählen, um Hintergrundinformationen (z. B. den Quellcode des repräsentierten Objekts) anzeigen zu lassen, oder die Kameraperspektive durch Rotieren, Schwänken oder Zoomen verändern.

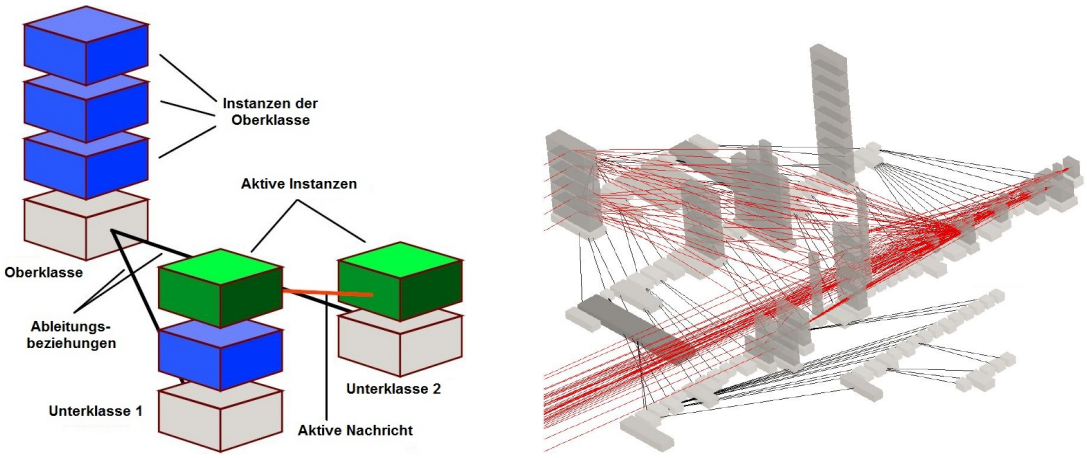
3.3. Sonstige Ansätze

Neben den zuvor vorgestellten, auf der UML oder der städtischen Metapher basierenden Ansätzen existieren noch weitere Visualisierungsformen, welche sich aufgrund ihrer Diversität nicht ohne Weiteres in gemeinsame Kategorien fassen lassen. Drei dieser Ansätze werden wir im Folgenden näher vorstellen.

3.3.1. TraceCrawler

Greevy et al. [32, 33] präsentieren einen mit der Visualisierung von DyVis vergleichbaren Ansatz zur Kombinierten Darstellung der Statik und Dynamik einer Anwendung. Ihr Softwarewerkzeug *TraceCrawler* erzeugt eine auf dreidimensionale Graphen basierende Visualisierung. Knoten symbolisieren Klassen und deren Instanzen. Kanten repräsentieren Ableitungsbeziehungen oder den Austausch von Nachrichten, d. h. Methodenaufrufe. Abbildung 15(a) zeigt eine Übersicht über die sichtbaren Elemente in TraceCrawler.

Knoten, welche sich auf unterster Ebene befinden (*Basisknoten* genannt), symbolisieren die Klassenobjekte der Anwendung, wohingegen die darüber liegenden Knoten die Instanzen zur Laufzeit (d. h. die dynamischen Objekte) repräsentieren und als Turm auf die Basisknoten gestapelt sind. Der Anwender kann in der Applikation dabei nach Belieben drei verschiedene Metriken aus einer Menge unterstützter Metriken wählen, welche die Höhe, Breite und Farbe der Basisknoten definieren. Adjazente schwarze Linien zwischen zwei Objekten stellen dabei stets eine Ableitungsbeziehung dar, wohingegen rote Kanten zwischen zwei dynamischen Laufzeitobjekten einen Methodenaufruf visualisieren. Bei Letzterem werden die an einem Aufruf beteiligten aktiven Objekte zudem zur leichteren Identifizierung grün gefärbt. Eine Kantenorientierung wird jedoch in beiden Fällen nicht dargestellt.



(a) Schematische Sicht auf die Visualisierung

(b) Instanzkollaborationsansicht

Abbildung 15: Ein Ausschnitt aus TraceCrawler [32]

Für die Visualisierung der Dynamik besitzt TraceCrawler zwei verschiedene Ansichten: Die *dynamische Feature-Trace-Ansicht* und die *Instanzkollaborationsansicht*. Die Erstgenannte stellt jeden Methodenaufruf des aufgezeichneten Programmablaufs isoliert betrachtet dar, d. h. zeigt in dem gesamten Schaubild gleichzeitig lediglich eine rote Kante zwischen zwei dynamischen Objekten. Dem Anwender stehen nun eine ganze Reihe von Funktionen zu Verfügung. So kann er sich vor oder zurück durch den gesamten Trace bewegen, diesen als Animation als Ganzes betrachten oder Informationen zum aktuellen Status der Programmausführung abrufen. In einem separaten Fenster wird zudem der gesamte visualisierte Trace in textueller Form angezeigt, wodurch auch ein Springen zu spezifischen Stellen möglich ist.

Die Instanzkollaborationsansicht hingegen zeigt die betrachtete Applikation nach dessen Terminierung. Sämtliche Methodenaufrufe im Programmtrace werden gemeinsam dargestellt, wobei die grünliche Einfärbung der beteiligten Objekte ausgelassen wird. Ein Beispiel dieser Ansicht ist in Abbildung 15(b) gegeben. Im Vergleich zur dynamischen Feature-Trace-Ansicht findet hier eine Abstraktion von der Reihenfolge der Methodenaufrufe hin zu einer Gesamtansicht statt. Diese Darstellung ist vor Allem dann nützlich, wenn man besonders frequentiert aufgerufenen Klassen und Objekte, d. h. die Applikations-Hotspots, identifizieren möchte.

3.3.2. Kausalitätsgraphen

Kausalitätsgraphen [66] zeigen den Kontrollfluss einer Anwendung in Form einer graphenbasierten Visualisierung, welche mit einem Aktivitätsdiagramm oder Kontrollflussgraph vergleichbar ist. In ihrer ursprünglichen Version repräsentieren Knoten Synchronisationsvorgänge sowie Interprozess-, bzw. Interthreadkommunikation. Kanten zeigen jeweils auf nachfolgende Synchronisationspunkte, d. h. symbolisieren den eigentlichen Kontrollfluss. Diese Darstellung lässt sich leicht auch auf allgemeine Methodenaufrufe erweitern, indem man diese ebenfalls durch Knoten visualisiert. Start- und Endknoten sind dabei entsprechend in der Visualisierung notiert. Die Erzeugung neuer Threads wird mittels eins *Fork*-Knotens symbolisiert. Abbildung 16 zeigt ein Beispiel dieser Darstellung, in welchem eine Anwendung zunächst zwei weitere Threads erzeugt, jede von diesen nach vorheriger Synchronisierung ein Methode ausführt und die Gesamtapplikation nach einem abschließenden Join terminiert.
Da die Menge an Knoten dabei besonders groß werden kann, erlauben Kausalitätsgraphen den Einsatz von so genannten *Superknoten*. Diese unterscheiden sich von regulären dahingehend, dass diese weitere zusammenhängende Subgraphen des Kausalitätsgraphen beinhalten dürfen und diese daher in einem einzigen Knoten zusammenfassen. In einer interaktiven Visualisierung können diese so zur besseren Übersichtlichkeit und Platzersparnis ein- und ausgeklappt werden. Die dynamische Visualisierung eines Programmtraces lässt sich dabei zusätzlich durch eine entsprechende Einfärbung der aktuell ausgeführten Methode eines Threads unterstützen.

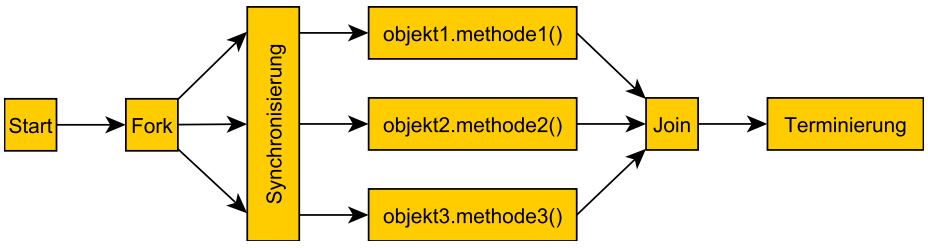


Abbildung 16: Beispiel eines (erweiterten) Kausalitätsgraphen

3.3.3. Tupelraumdarstellung

Auch der Tupelraum des Linda-Modells und die Interaktion der Threads und Prozesse mit diesem werden typischerweise auf Basis eines graphenbasierten Ansatzes visualisiert. Häufig wird der Tupelraum dabei in Form einer Wolke oder eines Ovals repräsentiert, in welcher die im Tupelraum enthaltenen Elemente eingeschlossen sind. Interaktionen werden dabei durch eingehende (für die *out*-Methode), bzw. ausgehende Pfeile (für die *in*-Methode) symbolisiert. Der Parametertupel dieser Operationen sowie z.T. auch der aufrufende Thread oder Prozess wird dabei jeweils im Diagramm notiert.

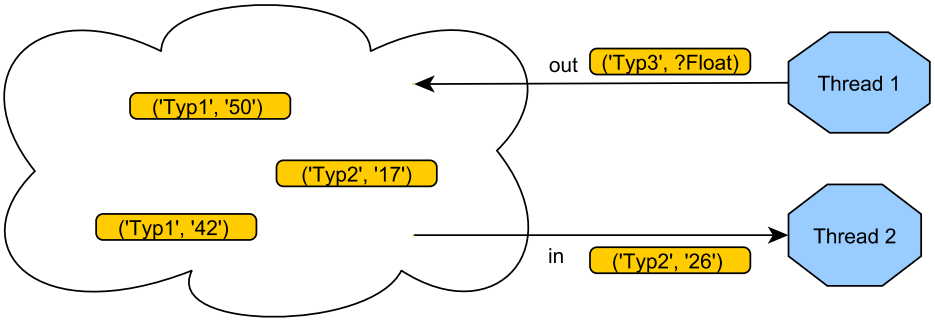


Abbildung 17: Klassische Tupelraumdarstellung im Linda-Modell

Eine vereinheitlichte Symbolik, bzw. genereller Konsens bei der Visualisierung des Linda-Modells existiert jedoch nicht, sodass diese Informationen auf eine Vielzahl unterschiedlicher Weisen bezüglich Form- und Farbgebung der Knoten und Kanten dargestellt werden können. Abbildung 17 zeigt ein Beispiel, in welcher wir Threads als blaue Hexagone und den Tupelraum als weiße Wolke symbolisieren. Gelbe Knoten zeigen entsprechend einzelne Tupel mitsamt ihrer Attribute.

4. Bewertung existierender Ansätze

Nachdem wir nun eine Reihe für unsere Arbeit relevanter Visualisierungsansätze vorgestellt haben, gilt es nun, diese nun hinsichtlich ihrer Qualität und ihres Umfangs der Darstellung zu beurteilen. Unser Ziel ist es dabei, diejenigen Ansätze zu identifizieren, welche sich am Besten als Ausgangsbasis für die Entwicklung unser eigenen Visualisierung der Statik und Dynamik in Kombination mit Synchronisationspunkten eignen.
Viele der von uns vorgestellten Darstellungen unterscheiden sich dabei in ihren charakteristischen Eigenschaften. So ermöglicht beispielsweise das Softwarewerkzeug EvoSpaces bereits ein recht umfangreiche Darstellung der Statik und Dynamik, beinhaltet jedoch kaum Ansätze zur Integration von Synchronisationsmechanismen. Eine Tupelraumdarstellung hingegen visualisiert lediglich Letztgenanntes, und dies auch nur im Zusammenhang mit den *in*- und *out*-Operationen des Linda-Modells. Jedoch ließe sich diese Darstellung gegebenenfalls leicht um weitere Aspekte der Dynamik erweitern, was zu prüfen ist. Unser Bestreben ist es daher nicht, einzig den aktuellen Ist-Zustand einer Visualisierung zu beurteilen sondern auch dessen Potential hinsichtlich ihrer Erweiterbarkeit um bislang noch nicht unterstützte Aspekte zu bewerten.

4.1. Evaluationskriterien

Freitas et al. [29] haben bereits einige, relativ allgemein gefasste Evaluationskriterien zur Bewertung von (Software-)Visualisierungen zusammengestellt, aus welchen wir gemeinsam mit den Werken weiterer Autoren unsere eigenen Bewertungsmaßstäbe geformt haben. Diese lassen sich zu den vier Kategorien *Metapher*, *Vollständigkeit*, *Funktionalität* und *Ästhetik* zusammenfassen, welche wir im Folgenden detailliert vorstellen und begründen möchten.

4.1.1. Metapher

Die Kriterien in dieser Kategorie dienen der Bewertung der verwendeten Metapher. Metaphern sollen es dem Benutzer erleichtern, sich in einer Visualisierung zurechtzufinden und sind daher häufig an Gegebenheiten der realen Welt angelehnt. Eine wichtige Eigenschaft für die Verständlichkeit einer Darstellung stellt daher der *Realismus* der verwendeten Metapher dar. Je näher sich eine solche an der Umwelt und den Gewohnheiten des Betrachters orientiert, desto leichter fällt es diesem, sich aufgrund der Vertrautheit mit dieser Struktur in der Darstellung zurechtzufinden [38]. Dieser Effekt wird dabei im Fall einer dreidimensionale Darstellung verstärkt. So profitiert der Betrachter von den räumlichen Wahrnehmungs- und Orientierungsfähigkeiten, welche er sich im Laufe seines Lebens angeeignet hat [50] und deren kognitive Vorgänge oftmals Zeit und Denkprozess sparend ins Unterbewusstsein verschoben werden können [38]. Symptomatischerweise fällt es beispielsweise vielen Menschen leichter, adjazente Knoten in drei- als

in zweidimensionalen Graphen zu identifizieren [50]. Realismus verstehen wir jedoch auch im Sinne der symbolischen Codierung. Repräsentiert eine Visualisierung beispielsweise einen Softwareanwender mittels eines Symbols, ist es aufgrund der Analogie zur Realität wesentlich intuitiver, diesen als Strichmännchen darzustellen als in Form eines Rechtecks. Das Vorhandensein von markanten oder einzigartigen graphischen Elementen sowie eine ausgeprägte Asymmetrie unterstützen zusätzlich die Orientierung und begünstigen das Wiederfinden gesuchter Objekte in einer Darstellung. Dies lässt sich damit begründen, dass man sich wesentlich leichter an außergewöhnliche Visualisierungselemente erinnert als an unauffällige. Diese Eigenschaft einer Visualisierung bezeichnen wir als *Diversität*.
Marcus et al. [50] beschreiben mit der *Ausdrucksstärke* (engl. *expressiveness*) ein weiteres wichtiges Charakteristikum einer Metapher. Ein zentraler Bestandteil dieser Begrifflichkeit sind dabei die *visuellen Parameter* einer Darstellung, welche die potentiellen, aber nicht notwendigerweise implementierten Variationsmöglichkeiten der Eigenschaften ihrer graphischen Elemente beschreiben. Beispiele hierfür sind die Maße, Form, Farbe oder Textur eines sichtbaren Objekts in der Darstellung. Eine Metapher wird als *ausdrucksstark* bezeichnet, wenn sie über ausreichend viele visuelle Parameter verfügt, um sämtliche darzustellende Eigenschaften des Softwaresystems codieren zu können.
Als Beispiel einer ausdrucksstarken Metapher könnte man einen Graphen anführen, dessen Knoten sich in ihrer Höhe und Breite unterscheiden, mittels welchen man die Anzahl der Attribute und Methoden von Klassen visualisieren möchte. Die Dimension der Daten sowie die Menge der visuellen Parameter beträgt hier zwei und sind daher zahlenmäßig kompatibel. Im Rahmen unserer Evaluierung verwenden wir diesen Begriff jedoch nicht als absolute sondern als quantitative Eigenschaft einer Metapher. So bezeichnen wir eine Darstellung, welche über mehr visuelle Parameter als eine andere verfügt, allgemein als *ausdrucksstärker*. Da jedoch deren Anzahl nicht immer exakt quantifizierbar ist, geben wir lediglich eine grobe Beurteilung der Ausdrucksstärke an. Die Menge der jeweils verwendeten virtuellen Parameter können wir jedoch genau benennen.
Aus der Anzahl der nicht verwendeten visuellen Parametern können wir dabei direkt auf die Erweiterbarkeit der zu Grunde liegenden Metapher schließen. Ist deren Anzahl gering, lassen sich ohne die Einführung weiterer optischer Elemente kaum weitere, bislang noch nicht codierte Informationen in dieser integrieren.
Im vorangegangenen Kapitel haben wir mit der städtischen und der graphenbasierten Metapher zwei unterschiedliche Visualisierungsformen kennengelernt. Einige der oben genannten Kriterien bewerten dabei unmittelbar die verwendete Metapher und werden sich daher von Ansatz zu Ansatz nicht besonders unterscheiden. Wir können daher bereits hier die städtische Metapher in puncto Realismus und Dimensionalität als Sieger gegenüber dem graphenbasierten Ansatzes festhalten. Im Bereich der Ausdrucksstärke und Diversität finden sich jedoch innerhalb der Visualisierungsansätze oftmals leichte Unterschiede, welche wir daher individuell bewerten werden. Dennoch impliziert eine höhere Dimensionalität und die Verwendung unterschiedlicher Knoten im Fall der städtischen Metapher (Distrikte und Gebäude) häufig auch eine höhere Ausdrucksstärke.

4.1.2. Vollständigkeit

Die Kriterien in dieser Kategorie dienen der Bewertung der Vollständigkeit und Erweiterbarkeit der Visualisierung. Eine Darstellung muss dabei sowohl Informationen der Statik und Dynamik als auch der Synchronisationspunkte in einer ganzheitlichen Abbildung integrieren können. Auch obwohl diese Arbeit vornehmlich die Programmiersprache Java betrachtet, sollten auch sprachunabhängige Konstrukte wie z. B. Monitore, welche in Java lediglich implizit im Form der synchronisierten (*synchronized*) Methoden vorhanden sind, visualisiert werden können. Dabei ist es nicht erforderlich, bzw. sogar kontraproduktiv, jeden Aspekt des Softwaresystems detailliert in der Visualisierung wiederzugeben, da die Informationsflut andernfalls die Übersichtlichkeit mindern könnte. Stattdessen sollten auf Basis des Abstraktionsprinzips lediglich relevante Inhalte (z. B. im Fall von Funktionsaufrufen nur das aufrufende und aufgerufene Objekt) abgebildet werden, um die Darstellung möglichst leicht verständlich zu halten und den Fokus des Betrachters auf die wesentlichen Informationen zu lenken [28, 42, 55]. Konkret erwarten wir von einer ganzheitlichen Visualisierung die Abbildung der folgenden Programmelemente:

Statik Im Bereich der Statik sollte eine Visualisierung Pakete, Klassen, Assoziations- und Ableitungsbeziehungen sowie Attribute und Methoden repräsentieren können. Eine Darstellung von Interfaces, abstrakten Klassen sowie leeren Paketen ist dabei nicht erforderlich, da diese für die Darstellung der Dynamik nicht relevant sind. Viele Ansätze visualisieren dabei Methoden und Attribute lediglich als Metriken, welche beispielsweise die Größe von graphischen Elementen definieren. Dies ist im Sinne des Abstraktionsprinzips ebenfalls akzeptabel.

Dynamik Die Darstellung sollte die Dynamik der Programmausführung umfassend visualisieren können, d. h. Instanzen von Klassen sowie Methodenaufrufe und deren Ausführungsdauer müssen abgebildet werden können. Im Fall von nebenläufigen Applikationen sollten zudem Threadzustände sowie nach Möglichkeit auch Context-Switches in der Darstellung codiert werden können.

Synchronisationsmechanismen Die Darstellung sollte eine Vielzahl unterschiedlicher Synchronisationsmechanismen visualisieren können. Zu diesen gehören neben Semaphore, Monitore und Thread-Joins auch die Tupelräume des Linda-Modells sowie die Java spezifischen Wait- und Notify-Operationen. Synchronisierte Methoden sind in dieser Auflistung bereits durch Semaphore, bzw. Monitore enthalten. Darüber hinaus wäre zudem die Kenntlichmachung von kritischen Sektionen wünschenswert.

4.1.3. Funktionalität

Eine ganzheitliche Visualisierung kann sehr komplex werden. Da sie dem Betrachter aber dennoch eine übersichtliche Darstellung der wesentlichen Charakteristika der Applikation bieten soll, muss eine sie implementierende Applikation Benutzerinteraktionen erlauben. Diese Fähigkeit ermöglicht insbesondere erst das dynamische Abspielen eines gegeben Programmtraces, welche unserer Meinung nach eine Kernfunktionalität einer Softwarevisualisierung darstellen sollte.
Ben Shneideman [58] stellt in seiner *Taxonomie der Informationsvisualisierung* sieben Funktionalitäten vor, welche eine attraktive und leicht verständliche Darstellung besitzen sollte. Zu diesem Zweck erweitert er sein Mantra der visuellen Informationssuche *Overview first, zoom and filter, then details-on-demand* [58] um die Aspekte *Verknüpfung*, *Historie* und *Extraktion*, welche besonders den Faktor der Benutzerfreundlichkeit einer Anwendung adressieren.
Im Folgenden möchten wir unsere Evaluationskriterien der Darstellungsfunktionalität näher vorstellen, von denen die ersten sieben Eigenschaften der Taxonomie von Shneideman entsprechen.

Übersicht Eine Visualisierung sollte eine Übersicht über den gesamten zu Grunde liegenden Datensatz bieten. Der Benutzer kann sich auf diese Weise einen Überblick über die Darstellung verschaffen und so für ihn relevante Bereiche identifizieren. Jede speziellere Ansicht sollte daher auch über eine Funktion verfügen, mittels welcher er jederzeit zu dieser Übersichtsdarstellung zurückkehren kann. Durch weitere Navigationsmöglichkeiten, z. B. durch das Verschieben des aktuellen Bildausschnitts, sollte es dem Benutzer zudem leichter fallen, sich in der Ansicht zu orientieren.

Zoom Der Benutzer sollte mittels einer Zoom-Funktion jederzeit den für ihn interessanten Bildausschnitt vergrößern können. Dies ermöglicht eine wesentlich fokussiertere Betrachtung des ausgewählten Areals sowie die Darstellung von Details, welche in der Übersichtsansicht gegebenenfalls ausgeblendet werden. Der Zoom-Faktor sollte hierbei frei wählbar sein sowie die Zoom-Geschwindigkeit möglichst gering sein, um den natürlichen Sinn des Betrachters für Position und Umgebung nicht zu stören. Idealerweise wählt der Benutzer mit dem Mauspfeil einen Punkt auf dem Bildschirm aus und zoomt anschließend auf diesen Punkt ein (vgl. z. B. *Google Earth*[3]).

Filter Der Betrachter sollte in der Lage sein, für ihn uninteressante Elemente ausblenden zu können. Gewährt man Benutzern die Kontrolle über die Inhalte der Visualisierung, können diese ihren Fokus durch Eliminierung sämtlicher nicht relevanter Objekte schnell auf die für sie interessanten Elemente lenken.

[3] `http://earth.google.com/`

Details-on-Demand Sämtliche sichtbaren Objekte in der Darstellung sollten mit der Maus anwählbar sein, um bei Bedarf (z. B. über ein Kontextmenü) detaillierter Informationen über diese abfragen zu können. Sobald eine Menge von Objekten vom Benutzer fokussiert wurde, dürfte es ein Leichtes sein, Details über jedes einzelne Objekt abzurufen.

Verknüpfung Abhängigkeiten zwischen sichtbaren Objekten sollten auf Benutzerwunsch angezeigt werden können. Hat der Betrachter ein Objekt ausgewählt, sollten Objekte mit vergleichbaren Eigenschaften hervorgehoben werden. Eine dynamische Wahl dieser Abhängigkeiten über die Benutzeroberfläche unterstützt den Benutzer in besonderem Maße beim Verständnis komplexer Zusammenhänge und sollte in jeder Visualisierung integriert sein.

Historie Die Visualisierung sollte sämtliche Benutzereingaben intern speichern, um Aktionen rückgängig zu machen oder wiederholen zu können. Es ist selten, dass eine Eingabe das vom Benutzer gewünschte Ergebnis liefert. Dies gilt insbesondere dann, wenn die Person mit der Visualisierung und dessen Funktionalität noch nicht vertraut ist. Auf der anderen Seite ist das Erkunden einer umfangreichen Darstellung ein komplexer Vorgang. Durch eine solche Funktion kann der Benutzer jeden Schritt rekonstruieren und seinen Lerneffekt vergrößern.

Extraktion Der Benutzer sollte in der Lage sein, einzelne Ausschnitte der Visualisierung extrahieren zu können. Mit entsprechenden Speicherfunktionen kann eine Darstellung später wiederhergestellt werden oder zum Zweck der Zusammenarbeit mit anderen Personen ausgetauscht werden. Auch das gezielte Auswählen von Arealen zum Ausdrucken wäre eine denkbare Anwendung dieser Funktionalität.

Wiedergabe Die Visualisierung sollte Möglichkeiten zur dynamischen Wiedergabe von zuvor aufgezeichneten Programmtraces bieten. Durch eine Animation ließe sich die gesamte Applikationsausführung bequem rekonstruieren, wobei ebenso eine manuelle, schrittweise Visualisierung möglich sein sollte. Funktionen zur Navigation erlauben es dem Benutzer zudem, für ihn relevante Stellen im Trace gezielt auszuwählen und voneinander isoliert darzustellen.

4.1.4. Ästhetik

Damit eine Visualisierung ansprechend und leicht verständlich ist, sollte diese neben den funktionalen Anforderungen eine Reihe von ästhetischen Kriterien erfüllen. Eine wichtige Eigenschaft ist hierbei die *Kompaktheit* der Darstellung, welche die Eignung einer Visualisierung zur Repräsentation großer Softwaresysteme oder Programmtraces widerspiegelt und der Skalierbarkeit entspricht. Ist die Darstellung zu groß, passt sie unter Umständen nicht mehr auf eine Bildschirmseite. Die Navigation und Orientierung wird dabei

erschwert, da der Benutzer entweder ständig den momentan auf den Monitor sichtbaren Ausschnitt verschieben oder die Visualisierung auf die Bildschirmgröße verkleinern muss. Hierbei geht häufig der Überblick über den Gesamtzusammenhang häufig verloren oder die Darstellung wird so klein, dass Details oder Beschriftungen nicht mehr problemlos erkannt werden können [61]. Auf der anderen Seite erhöht sich mit größer werdenden Darstellungsgröße auch die Länge der Kanten. Die Identifikation adjazenter Knoten fällt vielen Betrachtern jedoch besonders leicht, wenn die Kanten möglichst kurz sind [31].
Eine Visualisierung sollte die Anzahl optischer Anomalien darüber hinaus so gering wie möglich halten. Viele Visualisierungsansätze verwenden Kanten zur Repräsentation von Abhängigkeiten zwischen den Entitäten des Softwaresystems. Steigt die Anzahl der Kanten, erhöht sich jedoch häufig auch die Anzahl der Kantenkreuzungen. Dies kann zur Folge haben, dass adjazente Knoten unter Umständen nicht mehr problemlos identifiziert werden können. Dieser Effekt wird verstärkt, wenn eine Kante zusätzlich starke Biegungen besitzt oder viele Richtungsänderungen vollzieht [31, 35]. Bei ungeeigneter Kantenführung oder überlappender Knotenplatzierung können zudem wichtige Elemente verdeckt oder ihre Details verschleiert werden. Schneidet z. B. eine Kante eine Beschriftung, wird die Lesbarkeit bereits beeinträchtigt oder im Extremfall verhindert.
Diese Anforderungen fassen wir mit den Begriffen *Kompaktheit*, *Kantenkreuzungen* sowie *graphische Überlappungen* zusammen und verwenden diese als Kriterien für unsere Evaluierung.

4.1.5. Übersicht über unsere Bewertungskriterien

Der Übersicht halber sowie als Rekapitulation der letzten Passagen haben wir unsere Bewertungskriterien nochmals einer handlichen Form zusammengefasst. Tabelle 1 zeigt diese Übersicht, in welcher unser Erwartungshorizont an eine ideale Visualisierung angegeben ist. Die letzte Spalte beschreibt dabei unsere Bewertungsstrategie bei Anwendung des jeweiligen Kriteriums. Ist eine Ausprägung mit *qualitativ* angegeben, vergeben wir einen Wert der Skala (++, +, +/−, −, −−), wobei ‚++' andeutet, dass der betrachtete Ansatz das Kriterium bestmöglich erfüllt. Bei der Ausprägung *Repräsentation* hingegen nennen wir lediglich die Darstellungsweise des beurteilten Aspekts, falls dieser von der Visualisierung unterstützt wird.

(Unter-)Kategorie	Kriterium	Ausprägung
Metapher	Bezeichnung	Name
	Dimensionalität	2D/3D
	Realismus	qualitativ
	Diversität	qualitativ
	Ausdrucksstärke	qualitativ
Statik	Pakete	Repräsentation
	Klassen	Repräsentation
	Assoziationen	Repräsentation
	Ableitungen	Repräsentation
	Attribute	Repräsentation
	Methoden	Repräsentation
Dynamik	Klasseninstanzen	Repräsentation
	Methodenaufrufe	Repräsentation
	Ausführungszeit	Repräsentation
	Context-Switches	Repräsentation
	Threadzustände	Repräsentation
Synchronisationspunkte	Semaphore	Repräsentation
	Monitore	Repräsentation
	Thread-Joins	Repräsentation
	Tupelräume	Repräsentation
	Wait/Notify	Repräsentation
	Kritische Sektionen	Repräsentation
Funktionalität	Übersicht	Ja/Nein
	Zoom	Ja/Nein
	Filter	Ja/Nein
	Details-on-Demand	Ja/Nein
	Verknüpfung	Ja/Nein
	Historie	Ja/Nein
	Extraktion	Ja/Nein
	Wiedergabe	Ja/Nein
Ästhetik	Kompaktheit	qualitativ
	Kantenüberschneidungen	qualitativ
	Grafische Überlappungen	qualitativ

Tabelle 1: Übersicht über unsere Bewertungskriterien

4.2. Evaluation der Ansätze

Nachdem wir unsere Bewertungskriterien festgelegt haben, können wir nun mit der Evaluierung der Visualisierungsansätze beginnen. Mit Ausnahme der UML [54] bewerten wir dabei jeden Ansatz isoliert voneinander. Da sich die Evaluation der UML-Erweiterungen kaum in anderen Bereichen als der Vollständigkeit der Synchronisationsdarstellung unterscheiden, fassen wir diese konzeptionell mit der nicht erweiterten UML zusammen und beurteilen sie gemeinsam. Sollten sich bezüglich eines Bewertungskriteriums individuelle Unterschiede zwischen einzelnen Erweiterungen ergeben, weisen wir jedoch entsprechend auf diese hin.

4.2.1. UML inklusive Erweiterungen

Da die UML (siehe Abschnitt 3.1.1) keine fertige Softwarelösung darstellt sondern lediglich eine Spezifikation zur Visualisierung von Anwendungssysteme ist, ist eine Bewertung der Funktionalität wenig sinnvoll. Zum Zeitpunkt dieser Arbeit existieren eine Vielzahl verschiedener Applikationen, welche die UML auf leicht unterschiedlicher Weise implementieren. Daher werden wir im Folgenden keine spezifische Anwendung evaluieren sondern das Potential der UML als Ganzes beurteilen.

Metapher Die Diagramme der UML basieren überwiegend auf der graphenbasierten Metapher. Lediglich Zeitverlaufsdiagramme visualisieren Attributausprägungen in Form von Pegelverläufen über die Zeit in einem zweidimensionalen Koordinatensystem. Auch obwohl dahingegen die Entitäten von Sequenzdiagrammen an einer horizontalen und vertikalen Raumachse ausgerichtet werden, ordnen wir ihre Darstellung dabei aufgrund ihrer Topologie und offensichtlichen Analogie den graphenbasierten Ansätzen zu.
Die UML-Stadtansicht aus den Erweiterungen von Lange und Chaudron [39, 40, 41] (siehe Abschnitt 3.1.2) verwendet zwar dem Namen nach die städtische Metapher, da deren „Gebäude“ jedoch bis auf den Wert einer Metrik keinerlei Repräsentationsfunktion besitzen und auf regulären UML-Diagrammen als Grundfläche platziert werden, kann man jedoch kaum von einer städtebasierten Visualisierung sprechen.
In der UML repräsentieren Knoten typischerweise Entitäten des Softwaresystems, d. h. Klassen, Objektinstanzen und Pakete, aber auch abstraktere Gebilde wie z. B. Zustände. Kanten hingegen symbolisieren Relationen zwischen diesen Objekten, beispielsweise Assoziationen, Ableitungsbeziehungen und Methodenaufrufe. Unterschiedliche Knoten- und Kantenformen geben dabei den Typ der Entität oder Relation an.
Zwar erlaubt die UML-Spezifikation [54] die Verwendung unterschiedlicher Symboliken, dennoch sind realistische Darstellungen oftmals lediglich bei der Darstellung von menschlichen Akteuren sowie von Paketen gegeben. Erstere werden dabei mittels Strichmännchen repräsentiert, wohingegen die Knoten der Letzteren in vielen Implementierungen die Form von Aktenmappen haben. UML-Diagramme sind gemäß der Spezifikation farblos

definiert, jedoch setzten viele Implementierungen sowie Jacot [44, 45, 46] (siehe Abschnitt 3.1.6) und die Erweiterungen von Malnati et al. [48, 49] (siehe Abschnitt 3.1.5) Farben ein, um Entitäten hervorzuheben.
Die Metapher der UML ist nicht besonders ausdrucksstark. So erlauben beispielsweise interne Knotenbeschriftungen oder textuelle Inhalte keine Verwendung der Höhe oder Breite als Alleinstellungsmerkmal. Gleichzeitig würde deren Nutzung, d. h. die Darstellung von Knoten in nicht minimaler Größe, aber auch die benötigte Diagrammfläche erhöhen und daher die Skalierbarkeit vermindern. Lediglich die Textur und die Platzierung der Knoten scheinen im Sinne der Erweiterbarkeit sinnvolle, bisher ungenutzte virtuelle Parameter zu sein. Da zudem die Anzahl der aktuell verwendeten Parameter nicht besonders groß ist, ist die Diversität der sichtbaren Elemente gering. Lediglich die Elemente der Klassendiagramme sowie die 3D-Objekte der Metrik- und der UML-Stadtansicht [39, 40, 41] unterscheiden sich in deren Größe signifikant genug, um als Orientierungsmarken innerhalb der Visualisierung verwendet zu werden.

Statik Die UML bietet auch in ihrer nicht erweiterten Form Möglichkeiten zur Darstellung sämtlicher statischer Elemente. So visualisieren Paketdiagramme die Paketstruktur einer Anwendung in Form von (z.T. ineinander verschachtelter) Knoten. Jeder Knoten repräsentiert dabei ein Paket.
Klassendiagramme enthalten die übrigen gewünschten Statikelemente und können wiederum in Paketdiagrammen enthalten sein. Klassen werden als rechteckige Knoten dargestellt, wohingegen Ableitungs- und Assoziationsbeziehungen mittels unterschiedlicher Kantentypen symbolisiert werden. Methoden sowie Attribute werden dabei auf textueller Basis in den zugehörigen Knoten notiert. Die Komposition dieser beiden Diagramme ermöglicht daher eine sehr detaillierte Sicht auf die Statik eines Softwaresystems und erfüllt sämtliche Kriterien dieser Unterkategorie.

Dynamik Die UML kann viele Facetten der Applikationsdynamik lediglich mit Hilfe ihrer Erweiterungen darstellen. Zwar zeigen Sequenz-, bzw. Kollaborationsdiagramme Klasseninstanzen in Form von Knoten sowie Methodenaufrufe als gerichtete Pfeile, jedoch können diese per se keine Context-Switches visualisieren oder Threadzustände und Laufzeiten angeben. Erstere zeigen jedoch die Erweiterungen von Artho et al. [16] (siehe Abschnitt 3.1.3) in Form von gepunkteten Linien, welche auf das dynamische Threadobjekt gerichtet sind. Threadzustände können hingegen mittels des Zustandsdiagramms von Jacot [44, 45, 46] dargestellt werden. Laufzeiten oder sogar die aktuelle Systemzeit lassen sich hingegen bereits in der unerweiterten UML sehr leicht, z. B. mittels einer Notiz oder zusätzlicher Beschriftung, direkt an einen Methodenaufruf im Sequenzdiagramm anfügen. Dies scheint jedoch zum Einen gemäß der UML-Spezifikation [54] nicht der Sinn der Notizen zu sein. Zum Anderen würde deren Anzahl linear zur Menge der dargestellten Methodenaufrufe wachsen und damit die Übersichtlichkeit der Darstellung

beeinträchtigen. Malnati et al. [48, 49] notieren hingegen von Vorn herein die jeweils aktuelle Systemzeit bei jedem Methodenein- und austritt direkt am linken Rand ihrer Diagramme. Auf diese Weise lässt sich leicht die vergangene Zeit zwischen zwei beliebigen Funktionsaufrufen ablesen. Ihre Erweiterungen stellen daher gerade in diesem Aspekt eine sehr sinnvolle Erweiterung der UML dar.

Synchronisationspunkte Die UML ermöglicht in ihrer ursprünglichen Form im Bereich der Synchronisationspunkte lediglich die Darstellung kritischer Sektoren. In Sequenzdiagrammen werden diese mittels Fragmenten, d. h. Umrahmungen, markiert sowie anhand des Schlüsselworts *critical* identifiziert [54]. Synchronisationsmechanismen zur Wahrung des gegenseitigen Ausschlusses können hingegen lediglich mit Hilfe einiger der beschriebenen Erweiterungen visualisiert werden. So zeigen die Sequenzdiagramme von Artho et al. [16] Context-Switches, Thread-Joins sowie Wait-/Notify-Operationen ähnlich wie Methodenaufrufe in Form von gestrichelten und gepunkteten Linien. Mehner und Wagner [52] (siehe Abschnitt 3.1.4) stellen Semaphor- und Threadoperationen in Kollaborationsdiagrammen mit Hilfe zusätzlicher Verbindungslinien und Stereotypen dar. Malnati et. al. [48, 49] hingegen zeigen Semaphor-Belegungen in Sequenzdiagrammen durch eine Knotensymbolik unter Verwendung unterschiedlicher Farben.
Das Linda-Modell wird von keinem Erweiterungsansatz unterstützt, jedoch ließen sich diese unmittelbar in die Sequenzdiagramme integrieren. Eine Möglichkeit wäre es, den Tupelraum als reguläres Objekt zu betrachten und die *in()*- und *out()*-Funktionen wie reguläre Methodenaufrufe inklusive Angabe der beteiligten Tupel im Diagramm zu notieren. Dieser Ansatz besticht dabei durch seine Einfachheit. Jedoch ist zur Wahrung der Verständlichkeit erforderlich, den aktuellen Zustand des Tupelraums, d. h. die Menge der beinhaltenden Tupel, nach jeder Tupelraumoperation z. B. nach Vorbild von Abbildung 17 auf Seite 40 erneut abzubilden. Dies würde jedoch die Symmetrie und Gleichartigkeit der Methoden- und Objektdarstellungen verletzten.

Funktionalität Da wir hier keine spezifische Implementierung der UML beurteilen, sondern diese lediglich als reinen Darstellungsansatz evaluieren, können wir einzig die Eignung der UML für die Integration dieser Funktionalitäten evaluieren.
Die *Übersichtsansicht* stellt bei der UML das abstrakteste Statik- bzw. Dynamikschaubild, d. h. je nach Schwerpunktlegung ein Anwendungsfall- oder Verteilungsdiagramm dar. Einzelne in diesen enthaltene Objekte lassen sich mittels einer *Zoom*-Funktion durch andere Diagrammtypen tieferer Abstraktionsschichten detaillierter darstellen. So lässt sich beispielsweise ein Anwendungsfalldiagramm durch ein Sequenzdiagramm konkretisieren, dessen Methodenaufrufe sich wiederum mittels Zustandsdiagramme weiter spezifizieren lassen. Darüber hinaus erlaubt die UML auch die *Filterung* einer gegeben Visualisierung. Abhängig von der realisierten Implementieren müssen hierzu jedoch die gewählten optischen Elemente oftmals (abhängig von der gewählten Implementierung)

zunächst markiert und anschließend in ein neues Diagramm kopiert werden [65]. Relationen zwischen Visualisierungselementen sind dabei von Vornherein bereits in vielen Diagrammtypen in Form von Kanten (z. B. Assoziations- und Ableitungsbeziehungen) vorhanden und müssen daher nicht extra ergänzt werden (*Verknüpfung*) Mittels optional darstellbarer Daten (z. B. in Form von Notizen) lassen sich Hintergrundinformationen anzeigen (*Details-on-Demand*). Programmtraces lassen sich mittels (erweiterter) Sequenzdiagramme, welche dynamisch erstellt werden, direkt darstellen (*Wiedergabe*), in welchen der Benutzer nach Belieben navigieren und Szenarien wiederholen kann (*Historie*). Somit lassen sich bezüglich der UML sämtliche Evaluationskriterien in dieser Kategorie erfüllen.

Ästhetik Einige situationsspezifische Skalierbarkeitsprobleme haben wir bereits in den vorangegangenen Bewertungskategorien angesprochen. Jedoch ist die UML auch im Allgemeinen Anwendungsfall aufgrund ihrer fehlenden Kompaktheit kaum skalierbar. So werden beispielsweise Sequenzdiagramme schon bei sehr kleinen Applikationen oder Programmtraces relativ groß sowie Zustandsdiagramme aufgrund bei einer Vielzahl an Aufrufpfeilen pro Verbindung sehr unübersichtlich. Die überwiegend textgestützten Diagramme (v. A. Klassen- und Objektdiagramme) können kaum größere Mengen an Klassen gleichzeitig darstellen ohne zwecks Abbildung auf einer begrenzten Zeichenfläche verkleinert werden zu müssen. Die Lesbarkeit ihrer textuellen Komponenten wird unmittelbar erschwert. Dies macht insbesondere die Metaansicht von Lange und Chaudron [39, 40, 41] als Komposition verschiedener UML-Diagramme sehr schnell unbrauchbar.
In vielen UML-Diagrammen treten Kantenüberschneidungen vergleichsweise selten auf. Dies liegt darin begründet, dass sich Kanten in der Regel vom Benutzer frei zeichnen lassen und nicht an räumliche Beschränkungen gebunden sind. Daher existiert stets eine kreuzungsminimale Darstellung der Kanten. Lediglich Sequenzdiagramme scheinen aufgrund ihrer Topologie besonders häufig von diesen betroffen zu sein. Methodenaufruf darstellende Pfeile werden oftmals von Lebenslinien geschnitten, wodurch Methodensignaturen bei ungeeigneter Platzierung in ihrer Lesbarkeit beeinträchtigt werden.
In der Metrikansicht und der UML-Stadtansicht der Erweiterungen von Lange und Chaudron [39, 40, 41] kommt es bei unsachgemäßer Platzierung der Metrik visualisierenden Objekte zudem häufig zu graphischen Überlappungen. So kann es passieren, dass geometrische Körper auf Diagrammflächen stehen, welche bereits durch andere UML-spezifische Knoten oder Beschriftungen genutzt werden. In Folge dessen ist das Erkennen oder Lesen dieser Zeichen unmöglich. Dies war bereits in Abbildung 7(c) auf Seite 26 zu erkennen.

Fazit Die UML bietet auch in ihrer nicht erweiterten Form umfangreiche Möglichkeiten zur detaillierten Darstellung von statischen und dynamischen Softwareelementen. Dennoch ermöglicht diese per se noch keine Visualisierung von Synchronisationsmechanismen. Diese können erst mittels der in Anschnitt 3.1.1 beschriebenen Erweiterungen sinnvoll veranschaulicht werden. Die größte Schwierigkeit der UML liegt jedoch in der

fehlenden Kombinierbarkeit ihrer einzelnen Diagrammtypen. Zwar versuchen Lange und Chaudron [39, 40, 41] mittels ihrer Metaansicht diesen Makel zu überwinden, dennoch vermindert diese Darstellung auch die ohnehin schon geringe Skalierbarkeit dieses Visualisierungsansatzes noch weiter. In den übrigen ästhetischen Kriterien zeigen sich jedoch die Vorzüge der UML, da ihre Schaubilder aufgrund der häufig überschneidungsfreien Darstellung besonders ansprechend wirken.

4.2.2. EvoSpaces

EvoSpaces [13, 14, 15, 27] (siehe Abschnitt 3.2.1) unterscheidet sich von vielen betrachteten Softwareanwendungen in seiner großen Anzahl integrierter Darstellungen und bietet mit der Tag-, Nacht-, Etagen- sowie Außenansicht mehr Sichten als die vergleichbaren Tools DyVis [65] und TraceCrawler [32, 33]. Insbesondere modellieren die beiden letztgenannten Darstellungen keine Gebäude- oder Knoteninterna.

Metapher EvoSpaces basiert auf der dreidimensionalen städtische Metapher. In der Tagansicht kommen dabei eine Vielzahl unterschiedlicher virtueller Parameter zum Einsatz. So bildet die Anwendung Klassen auf Gebäude unterschiedlicher Textur und Höhe ab. Pakete hingegen werden als verschachtelte quadratische Grundflächen gleicher Färbung in unterschiedlicher Helligkeit symbolisiert. In der Etagenansicht codieren unterschiedlich gefärbte „Arbeiter“ in Form von Strichmännchen auf intuitive Weise ausführbare Methoden und ergänzen damit die ohnehin schon realistische Metapher um eine weitere passende Assoziation. Unterschiedlichen Etagen dienen dabei als Klassifizierung der visualisierten Methodentypen.
In der Nachtansicht sind die Gebäudetexturen und Pakethierarchien unsichtbar. Stattdessen werden Farben verwendet, um die unterschiedliche Aktivität der Klassen im Programmtrace sichtbar zu machen. Schwarze Linien zeigen Relationen zwischen einzelnen Klassen. Durch die unterschiedlichen und klar erkennbaren Gebäudetypen entstehen häufig markante Gebäudekonstellationen. Diese kann man bei der Suche nach bereits georteten Klassen als Orientierungshilfe verwenden und prägen ein individuelles und realistisches Stadtbild.
EvoSpaces deckt bereits einen Großteil der verfügbaren virtuellen Parameter ab und nur wenige weitere Parameter lassen sich unserer Meinung nach sinnvoll zusätzlich integrieren. Beispiele für diese wären unterschiedliche Färbungen der Relationen oder dreidimensionale Grundflächen mit differierender Höhe. Der Einsatz verschiedener Farben bei den Gebäudetexturen wäre auf der anderen Seite beispielsweise unvorteilhaft, da sich auf dieser Weise die unterschiedlichen Gebäudetypen und -Texturen nicht mehr problemlos identifizieren lassen.

Statik EvoSpaces visualisiert Klassen in Form von Gebäuden und Pakete als unterschiedlich saturierte Grundflächen. Mittels unterschiedlicher Ausprägungen der Gebäudehöhe und ihrer quadratischen Grundrisse lassen sich dabei eine Reihe quantitativer Metriken abbilden. Alam und Dugerdil [14, 15] erwähnen zwar nicht, ob EvoSpaces auch die Verwendung der Methoden- und Attributanzahl unterstützt, jedoch ließen sich diese Metriken gegebenenfalls leicht implementieren. Methoden werden zusätzlich in der Etagenansicht durch Strichmännchen repräsentiert. Einzelne Attribute könnten in diese Darstellung bei Bedarf durch weitere Strukturen integriert werden. Relationen zwischen den Klassen werden durch schwarze Linien mit angedeuteter Orientierung visualisiert. Zwar nennen Alan und Dugerdil nicht die Arten von Relationen, welche EvoSpaces unterstützt, jedoch ließe sich durch eine unterschiedliche Färbung der Kanten relativ leicht zwischen Assoziations- und Ableitungsbeziehungen unterscheiden.

Dynamik EvoSpaces stellt die Dynamik einer Anwendung lediglich in der Nachtansicht dar. In dieser werden Methodenaufrufe analog zu den Relationen aus der Tagansicht durch Kanten zwischen unterschiedlichen Gebäuden symbolisiert. Deren Laufzeiten werden dabei nicht angegeben, jedoch ließe sich diese Information leicht durch eine unterschiedliche Farbgebung der Kanten codieren. Da Methodenaufrufe in EvoSpaces abstrahiert von den Klassen und nicht von deren Instanzen ausgehen, bleiben konkrete Laufzeitobjekte unvisualisiert. Hierunter leidet jedoch die Granularität und Nutzbarkeit der Darstellung, da gerade in einigen Szenarien (z. B. bei dem Dinierenden-Philosophen-Problem [26]) die genaue Angabe der beteiligten Laufzeitobjekte entscheidend für das Verständnis des Applikationsverhaltens ist. Würde man Instanzen jedoch wie in DyVis in Form von Etagen visualisieren, könnte man die Höhe nicht mehr zur Codierung einer eigenen Metrik verwenden sowie müsste bessere Struktur unterstützende Texturen entwickeln. Dennoch würde dieser Schritt die Darstellung unserer Ansicht nach qualitativ deutlich aufwerten.
Threadzustände und Context-Switches sind kein Bestandteil der EvoSpaces-Darstellung und lassen sich auch nicht ohne weitere Sichten oder Veränderung der bisherigen Informationscodierung integrieren. Dies liegt vor Allem darin begründet, dass EvoSpaces nicht zwischen unterschiedlichen Threadausführungen unterscheidet, d. h. von einem sequenziellen Programm(-fluss) ausgeht. Somit müsste zunächst eine Unterstützung von nebenläufigen Anwendungen geschaffen werden.

Synchronisationspunkte Mit der eben genannten Begründung wird ebenso die Darstellung von Synchronisationspunkten nicht unterstützt. Daher lassen sich diese ähnlich schwierig wie Context-Switches oder Threadzustände abbilden und demzufolge nicht auf Anhieb in die Darstellung integrieren.

Funktionalität EvoSpaces verfügt über vier der von uns geforderten acht Funktionalitäten. So bietet die Tagansicht eine *Übersicht* über die gesamte (statische) Struktur einer Applikation. Mittels der Etagenansicht lassen sich weitere Details über Gebäude anzeigen *Zoom*. Eine *Wiedergabe* von Programmtraces ermöglicht die mikroskopische Ansicht in Form einer Animation und Relationen zwischen Gebäuden können als schwarze Linien in der Tagansicht eingeblendet werden (*Relation*). Über visuelle Elemente lassen sich dabei keine Hintergrundinformationen abrufen (*Details-on-Demand*). Alam et al. [13, 14, 15, 27] geben leider nicht an, ob ihr Softwarewerkzeug auch eine Filter- oder Extraktionsfunktion beinhaltet oder sogar eine Historie über die Benutzerinteraktion anlegt. Da EvoSpaces zudem zwecks Selbsttests nicht frei verfügbar ist, müssen wir leider davon ausgehen, dass die Anwendung auch die Kriterien *Filter*, *Extraktion* und *Historie* nicht erfüllt.

Ästhetik Da Klassen durch Gebäude mit vergleichsweise kleiner Grundfläche repräsentiert werden, ermöglicht EvoSpaces die Darstellung einer Vielzahl von Klassen auf relativ kleinem Raum. Eine hohe Kompaktheit und damit Skalierbarkeit des Ansatzes ist daher gegeben. Dennoch könnten aufgrund der Paketstruktur und der stets rechteckigen Darstellung der Grundflächen zum Teil große Areale der Grundfläche ungenutzt bleiben, d. h. sind unbebaut, wodurch Kanten zwischen den Gebäuden gegebenenfalls unnötigerweise verlängert werden. Dies stellt aus unserer Sicht jedoch nur einen kleinen Makel dar. Kantenüberschneidungen oder graphische Überlappungen sind uns in EvoSpaces nicht aufgefallen.

Fazit EvoSpaces besticht durch seine realistische Metapher und durch den Einsatz von Texturen lebendig wirkenden Gebäudedarstellungen. Auch im Bereich der Ästhetik zeigt die Darstellung ansprechende Ergebnisse. Die große Schwäche dieses Ansatzes zeigt sich jedoch in seinem Visualisierungsumfang. Zwar ist EvoSpaces beinahe auf Anhieb in der Lage, sämtliche Bewertungskriterien der Statik zu erfüllen. Dennoch zeigen sich große Schwächen in seinem Darstellungspotential in Bezug auf die übrigen Programmelemente. So erfüllt EvoSpaces von den insgesamt elf Kriterien der Vollständigkeit der Dynamik und Synchronisationspunkte lediglich ein einziges. Auch die fehlende Erweiterbarkeit des Darstellungsansatzes ist hierfür verantwortlich, da die Applikation nicht zwischen mehreren Threads oder Prozessen unterscheidet.

4.2.3. DyVis

DyVis [65] (siehe Abschnitt 3.2.2) wurde mit dem Ziel entwickelt, eine möglichst umfassende Darstellung der Statik und Dynamik zu bieten. Im Vorfeld seiner Implementierung fand vergleichbar mit unseren bisherigen Ausführungen eine Recherche nach verwendbaren Basisvisualisierungen sowie deren Evaluation statt. Es wurden jedoch zum Teil andere Bewertungskriterien verwendet.

Metapher DyVis basiert auf der städtischen Metapher und visualisiert Klassen in Form von gleichfarbigen Gebäuden unterschiedlicher Etagenzahl mit quadratischem Querschnitt. Pakete hingegen werden durch rechteckige Grundflächen gleicher Farbe und Höhe in abgestufter Helligkeit repräsentiert. Zwischen den Etagen können dabei unterschiedlich gefärbte Linien verlaufen, welche einen Methodenaufruf von codierter Dauer symbolisieren. Zwar variiert DyVis die Farbe einzelner Etagen, jedoch dient dies lediglich der besseren Identifikation referenzierter Objekte und codiert daher keine zusätzlichen Informationen. Die verwendete Metapher ist relativ ausdrucksstark. Im Sinne der Erweiterbarkeit können wir uns zudem die Verwendung unterschiedlicher Höhen und Färbungen der Grundflächen und Etagen sowie die unterschiedliche Gestaltung der Methodenaufruf repräsentierenden Linien vorstellen. Eine differierende Höhe der Stockwerke und Distrikte würde hingegen den ohnehin schon vergleichsweise geringen Realismus der Visualisierung noch weiter beeinträchtigen und sollte daher nur mit Bedacht gewählt werden. Die Verwendung von Texturen sowie einer realistischeren Gebäudedarstellung könnte jedoch die Realitätsnähe der Visualisierung sehr steigern.
Da eine Etage eine Klasseninstanz repräsentiert und deren Anzahl oftmals stark differiert, besitzt die Darstellung aufgrund der unterschiedlichen Gebäudehöhen eine hohe Diversität. Da die Gebäudehöhe und damit die Gesamtstruktur der Stadt jedoch zur Laufzeit variiert, kann man bei der Wiedergabe von Programmtraces markante Gebäudekonstellationen nur bedingt als Orientierungshilfen verwenden, wodurch das Wiederfinden von gesuchten Objekten stark erschwert wird.

Statik DyVis codiert Pakete in Form von Distrikten und Klassen in Form von Gebäuden, deren Breite und Länge die Anzahl der Attribute widerspiegeln. Methoden werden dabei nicht explizit visualisiert, könnten jedoch in Form einer Erweiterung als zusätzliche Metrik durch die Gebäudebreite oder -länge isoliert codiert werden. Eine Darstellung von Assoziations- und Ableitungsbeziehungen wird von DyVis zudem nicht unterstützt, jedoch ließe sich die Applikation problemlos um diesen Aspekt ergänzen. Hierzu könnte man beispielsweise nach dem Vorbild von EvoSpaces [13, 14, 15, 27] gerichtete Kanten zwischen den Gebäudedächern zur Symbolisierung verwenden.

Dynamik Jede Etage eines Gebäudes mit Ausnahme des Erdgeschosses repräsentiert eine Klasseninstanz. Methodenaufrufe werde dabei als Kanten zwischen diesen visualisiert. Die unterschiedliche Färbung der Kanten codiert dabei die Laufzeit des repräsentierten Methodenaufrufs. DyVis zeigt jedoch keine Context-Switches oder Threadzustände. Da die momentane Visualisierung jedoch keinen Unterschied zwischen den Ausführungen einzelner Threads macht, lassen diese sich diese Informationen auch nur schwer auf Anhieb integrieren. Ein entsprechendes Konzept müsste daher noch entwickelt werden.

Synchronisationspunkte DyVis unterstützt keinerlei Darstellung von Synchronisationsmechanismen und müsste daher um diesen Aspekt noch erweitert werden. Hierzu müsste man jedoch zunächst unterschiedliche Threads visualisieren können. Da jedoch der visuelle Parameter der Kantenfarbe bei der Darstellung von Methodenaufrufen bereits verwendet wird, gestaltet sich dieses Vorhaben als ausgesprochen schwierig. Eine Ergänzung der Kanten um eine Thread-ID als Label oder eine unterschiedliche Kantendarstellung pro Thread wäre beispielsweise nicht sinnvoll, da Erstere bei gleichzeitigem Vorhandensein vieler Kanten vom Betrachter ggf. nicht korrekt zugeordnet werden können. Letztere hingegen werden bei besonders heller Kantenfarbe oftmals nicht problemlos erkannt und eine nicht durchgängige Linie erschwert häufig das Finden von adjazenten Knoten. Das bisherige Konzept der Laufzeitcodierung sollte daher überdacht und gegebenenfalls überarbeitet werden.

Funktionalität DyVis erfüllt fünf unserer acht Funktionalitätskriterien. So verfügt die Applikation über lediglich eine Sicht, welche gleichzeitig der *Übersichtsansicht* entspricht. Mittels einer *Zoom*-Funktion kann sich der Betrachter in dieser frei bewegen (*Navigation*) sowie über ein Kontextmenü nähere Informationen zu jedem optischen Element abrufen (*Details-on-Demand*). Mittels einer Animation lassen sich dabei Programmtraces abspielen (*Wiedergabe*), in welcher Methodenaufrufe, d. h. Abhängigkeiten zwischen Objekten, in Form von Linien dargestellt werden (*Verknüpfung*). Die Kriterien *Filter*, *Historie* und *Extraktion* erfüllt DyVis nicht.

Ästhetik Die von DyVis erzeugte Darstellung wirkt aufgrund der Verwendung der städtischen Metapher sehr kompakt. Dennoch erscheinen einige Paketdarstellungen aufgrund der stets quadratischen Gestaltung deutlich zu groß im Vergleich zur Menge und Größe der enthaltenen Klassen. Große Areale der Grundfläche bleiben daher oftmals ungenutzt, woraus eine unnötige Verlängerung der Methoden darstellenden Kanten resultiert. Dies ist bereits in Abbildung 14 auf Seite 37 zu erkennen.
Da DyVis bei nicht aktivierter *Zeige-Ausführungspfad*-Option stets nur eine begrenzte Anzahl an Methodenaufrufen gleichzeitig visualisiert, bleibt die Zahl der Kantenüberschneidungen gering. Jedoch sind die referenzierten Etagen aufgrund ungeeigneter Kantenführung im Fall von Selbst-Schleifen, d. h. bei Methodenaufrufe innerhalb eines Objekts, oftmals nur schwer zu identifizieren. Dies kann daher das Verständnis des visualisierten Programmablaufs negativ beeinflussen und sollte in einer zukünftigen Version verbessert werden. Im Rahmen unsere Evaluation sind uns zwar keine Darstellungsprobleme aufgrund von graphischen Überlappungen aufgefallen, dennoch ist nach Betrachtung des Quellcodes keine Konzept zur Vermeidung von Gebäude schneidenden Kanten implementiert.

Fazit Die Qualität der von DyVis verwendeten Metapher unterscheidet sich nur gering von derjenigen in EvoSpaces [13, 14, 15, 27]. Zwar ist DyVis vor Allem aufgrund fehlender Texturen nicht derart realistisch, punktet dafür aber durch seine besondere Ausdrucksstärke. Die größte Schwierigkeit dieses Darstellungsansatzes liegt jedoch in dessen mangelndem Visualisierungsumfang. So zeigt DyVis keinerlei Synchronisationspunkte in der Programmausführung und die Darstellung der Dynamik ist nicht vollständig. Auch in der Ästhetik und der Funktionalität müssen Abstriche gemacht werden.

4.2.4. TraceCrawler

Im Gegensatz zu vielen anderen Visualisierungen existiert in TraceCrawler [32, 33] (siehe Abschnitt 3.3.1) keine eindeutig erkennbare Grundfläche. Obwohl eine solche dem Betrachter häufig davor bewahrt, die Orientierung bezüglich der Richtungen Oben und Unten zu verlieren, haben sich Greevy et al. [32, 33] dazu entschieden, die Diagrammelemente quasi schwebend auf einer Ebene im dreidimensionalen Raum anzuordnen.

Metapher TraceCrawler visualisiert Klassen in Form von übereinanderliegenden, dreidimensionalen Knoten, wobei jeder einzelne mit Ausnahme des Basisknotens eine Instanz der vom Stapel repräsentierten Klasse darstellt. Abhängig von der jeweilige Funktion (Klassen-, bzw. Instanzobjekt) wird jeder Knoten unterschiedlich farbig markiert sowie die Länge, Breite und Farbe jedes Basisknotens auf Grundlage von frei wählbaren Metriken über die Eigenschaften der symbolisierten Klasse definiert. Schwarze Kanten zwischen zwei Basisknoten codieren dabei Ableitungsbeziehungen, wohingegen rote Linien zwischen darüberliegenden Instanzobjekten einen Methodenaufruf repräsentieren. Die Kantenfärbung dient dabei lediglich der leichteren Erkennbarkeit des dargestellten Relationstyps und codiert keine eigenständige Information.
Die verwendete Metapher ist vergleichsweise ausdrucksstark und bietet auch im Sinne einer Erweiterbarkeit eine Reihe weiterer sinnvoll verwendbarer visueller Parameter. So könnte man beispielsweise eine unterschiedliche Knotenhöhe oder -Textur sowie eine verschiedenfarbige Gestaltung der Kanten und Nicht-Basisknoten zur Repräsentation weiterer relevanter Informationen verwenden.
Auch obwohl es sich bei der zu Grunde liegenden Metapher aufgrund der fehlenden Grundfläche nicht um eine städtische Darstellung handelt, besitzt diese dennoch eine große Ähnlichkeit mit einer Großstadt und ist insbesondere auch mit der von DyVis erzeugten Visualisierung vergleichbar. Aufgrund der starken Nähe zur City-Metapher erinnert uns die Darstellung dennoch eine urbane Struktur. Wir folgern daher einen entsprechend ausgeprägten Realismus von TraceCrawler.
Ebenso die Diversität der Darstellung ist als hoch einzuschätzen. Aufgrund der Abbildung von Klasseninstanzen als übereinanderliegende Knoten mit unterschiedlicher Grundfläche unterscheiden sich viele Stapel signifikant in ihrer Form und Höhe. Für jede dargestellte Applikation entsteht so häufig ein individuelles, charakteristisches Diagramm.

Statik In TraceCrawler werden Klassen in Form von Knotenstapeln sowie Methodenaufrufe und Ableitungsbeziehungsbeziehungen mittels farbiger Linien repräsentiert. Die Anzahl an Methoden und Attribute einer Klasse lassen sich als Metrik optional auf die Farbe, Länge oder Breite des Basisknotens abbilden. Der Benutzer entscheidet dabei über die exakte Komposition dieser Metriken. Pakete sowie Assoziationsrelationen können dabei nicht visualisiert werden. Greevy et al. [32, 33] haben auf die vollständige Darstellung der Statik verzichtet, da EvoSpaces vorrangig mit dem Ziel entwickelt wurde, die Dynamik umfassend darstellen zu können. Zwar könnten Pakete leicht nach dem Vorbild von DyVis durch eine Reihe ineinander verschachtelte Grundflächen repräsentiert werden, jedoch würde sich das von TraceCrawler erzeugte Schaubild dann kaum mehr von der Darstellung von DyVis unterscheiden. Assoziationsbeziehungen könnte man hingegen ohne eine geschaffene stärkere Ähnlichkeit durch eine unterschiedliche Farbcodierung der bereits verwendeten Kanten symbolisieren.

Dynamik Zwar wurde TraceCrawler für die Darstellung der Dynamik entwickelt, jedoch erfüllt seine Visualisierung in dieser Kategorie lediglich zwei der fünf Evaluationskriterien. So kann die Anwendung Klasseninstanzen sowie Methodenaufrufe gemäß des oben beschriebenen Musters darstellen, jedoch unterscheidet die Anwendung nicht zwischen mehreren Kontrollflüssen, d. h. verschiedenen Threads. Context-Switches und Threadzustände können demnach nicht repräsentiert werden. Ebenso werden gemessene Laufzeiten von Methoden nicht angezeigt. Jedoch könnten Letztere beispielsweise durch eine unterschiedliche farbliche Codierung der Methodenaufruf symbolisierenden Kanten mittels einer Metrik darstellt werden. Threadzustände könnten hingegen durch ein zusätzlichen Knotenstapel repräsentiert werden, in welcher jeder Knoten einen Thread symbolisiert und dessen Farbe seinen Zustand codiert.

Synchronisationspunkte Aufgrund der fehlenden Unterscheidung zwischen einzelnen Threadabläufen sind in TraceCrawler keine Darstellungen von Synchronisationsmechanismen integriert. Jedoch könnte man nach einem ähnlichen Prinzip wie auch schon bei der Visualisierung der Threadzustände viele von diesen als separate Knotenstapel mit entsprechender farblicher Codierung visualisieren. Da wir diese Idee auch in unserer eigenen Visualisierung verwenden, möchten wir hier jedoch auf die exakte Gestaltung nicht weiter eingehen sondern verweisen stattdessen auf unsere weiteren Ausführungen in Kapitel 5.

Funktionalität TraceCrawler erfüllt lediglich vier der insgesamt acht funktionalen Anforderungen. Die Anwendung besitzt dabei eine *Übersichtsansicht* insofern, dass der Betrachter so weit herauszoomen kann, bis das gesamte Diagramm sichtbar ist. Der Benutzer kann zudem direkt den referenzierten Quellcode betrachten (*Details-on-Demand*) sowie Programmtraces wiedergeben (*Wiedergabe*). Durch die Natur der Darstellung werden

ebenso statische wie dynamische Objektabhängigkeiten visualisiert (*Verknüpfung*). Die Bedürfnisse nach *Zoom*, *Filter*, *Historie* und *Extraktion* bleiben hingegen unbefriedigt.

Ästhetik TraceCrawler erfüllt unsere ästhetischen Kriterien nur ungenügend. So produziert die Darstellung in der Instanzkollaborationsansicht aufgrund der gleichzeitigen Einblendung sämtlicher im Programmtrace gespeicherter Methodenaufrufe häufig viele Kantenüberschneidungen. Zwar soll diese Sicht der Erkennung besonders aktiver Klassen dienen, jedoch ist bereits in Abbildung 15(b) auf Seite 38 zu erkennen, dass viele Kanten nicht adjazente Knoten schneiden. Eine Identifikation von verbundenen Knoten und daher von Applikations-Hotspots ist in diesem Fall häufig unmöglich. Demgegenüber lassen sich aufgrund der zu blassen Farbgebung einzelne Knoten pro Stapel besonders bei sehr kleiner Darstellung nur schwer erkennen. Lediglich die hohe Kompaktheit, welche vergleichbar mit derjenigen einer auf der städtischen Metapher basierenden Anwendung ist, kann man dieser Darstellung zu Gute halten.

Fazit Die Visualisierung von TraceCrawler ähnelt sehr stark der städtischen Metapher, auch obwohl die Darstellung diese aufgrund der fehlenden Grundfläche nicht wirklich implementiert. Qualitativ übernimmt sie aber trotzdem viele positive Eigenschaften dieser Struktur. Dennoch ist dessen Ästhetik vor Allem durch häufige Kantenüberschneidungen und graphischen Überlappungen geprägt. Die wahre Stärke von TraceCrawler liegt jedoch in dessen Visualisierungsumfang. Zwar sind einige Erweiterungen notwendig, aber dennoch ist die Darstellung in der Lage, 14 von 17 Kriterien in dieser Kategorie zu erfüllen.

4.2.5. Kausalitätsgraphen

Kausalitätsgraphen [66] (siehe Abschnitt 3.3.2) stellen einen relativ überschaubaren Ansatz zur Darstellung von Programmtraces dar und ähneln in ihrer Struktur stark den Kontrollflussgraphen, welche auf vergleichbare Weise den Programmablauf in Form eines übersichtlichen Diagramms veranschaulichen.

Metapher Kausalitätsgraphen visualisieren einen gegebenen Programmtrace auf Grundlage einer graphenbasierten Darstellung, in welcher Methodenaufrufe mittels Knoten und Übergänge zwischen diesen durch uniforme Pfeile symbolisiert werden. Die Metapher ist in ihrer Ausdrucksstärke mit derjenigen der UML vergleichbar. Da hier jedoch bislang nicht viele virtuelle Parameter Anwendung finden, eröffnen sich eine Vielzahl von Erweiterungsmöglichkeiten. So kämen unserer Ansicht nach die Textur, Form und Farbe der Knoten sowie die Struktur und Farbe der Kanten als sinnvolle Parameter für die Codierung zusätzlicher Informationen in Frage.

In ihrer jetzigen Form entsprechen Kausalitätsgraphen keiner unmittelbar erkennbaren Gegebenheit der realen Welt. Der Realismus der Darstellung ist daher gering. Ebenso ist dessen Diversität lediglich als gering einzustufen, da sich die Knoten lediglich in ihrer Breite unterscheiden. Kanten hingegen werden stets identisch dargestellt.

Statik Kausalitätsgraphen visualisieren keinerlei Elemente der Applikationsstatik. Entsprechend schwierig ist die Erweiterung der Darstellung um diesen Aspekt.

Dynamik In der Darstellung werden Methodenaufrufe in Form von Knoten symbolisiert. Die Kanten hingegen repräsentieren den Kontrollfluss der Applikation, wobei jeder Pfad vom Initial- zum Terminalknoten dem dynamischen Ablauf eines Threads entspricht. Damit erfüllen Kausalitätsgraphen per se lediglich eines der fünf Dynamikkriterien. Sämtliche übrigen Anforderungen sind ausschließlich durch geeignete Erweiterungen der Visualisierung erfüllbar. So könnte man beispielsweise die Laufzeit einer Methode mittels der Farbe der Knoten codieren. Context-Switches ließen sich hingegen mittels einer fortlaufender Sequenznummer an den Pfeilen kenntlich machen.

Synchronisationspunkte Synchronisationsvorgänge werden auf gleiche Weise wie Methodenaufrufe visualisiert. Da sämtliche, im Rahmen dieser Arbeit betrachteten Synchronisationsmechanismen auf Funktionsausführungen beruhen, unterstützen Kausalitätsgraphen die Darstellung von Semaphor-, Monitor- und Tupelraumoperationen sowie Thread-Joins und den Aufruf von *wait*, *notify*, bzw. *notifyAll*. Kritische Sektionen lassen sich hingegen direkt im Graph markieren (z. B. mittels Farbe oder Umrahmungen von Knoten). Letzteres stellt jedoch eine Erweiterung des Visualisierungskonzepts dar.

Funktionalität Da es sich bei Kausalitätsgraphen um keine eigenständige Softwarelösung sondern lediglich um eine Spezifikation zur Darstellung von Programmabläufen handelt, ist es nicht möglich eine momentane Funktionalität zu beurteilen. Stattdessen bewerten wir lediglich die Eignung für die Integration der genannten Funktionen.
Die Darstellung realisiert insgesamt bereits eine *Übersichtsansicht*, jedoch beinhaltet diese bislang keine Möglichkeit zur detaillierteren Darstellung einzelner Komponenten (*Zoom*). Der Benutzer könnte jedoch beispielsweise einzelne Elemente auswählen und isoliert in einer separaten Ansicht betrachten (*Filter*), sich Hintergrundinformationen (z. B. Laufzeiten von Methoden) beschaffen (*Details-on-Demand*) oder die aktuelle Ansicht speichern (*Extraktion*). Beziehungen zwischen Funktionsaufrufen werden dabei bereits in Form von Kanten in der Visualisierung dargestellt (*Verknüpfung*). Ein Programmablauf ließe sich auch automatisch wiedergeben, indem der Kausalitätsgraph iterativ gebildet wird (*Wiedergabe*). Eine Un- und Redo-Funktion könnte den Benutzer dabei in der Bedienung des graphischen Interfaces unterstützen (*Historie*). Kausalitätsgraphen erfüllen daher potenziell insgesamt sieben der acht Kriterien.

Ästhetik Kausalitätsgraphen sind nicht besonders skalierbar. Zwar erlauben diese die freie Platzierung der Knoten im zweidimensionalen Raum, da jedoch jeder Knoten einen Funktionsaufruf repräsentiert, wird die Darstellung bereits für kleine Programmtraces vergleichsweise groß und kann in der Regel nicht mehr übersichtlich auf einer Bildschirmseite darstellt werden.
Da Knoten vom Benutzer (z.B. in einem entsprechenden graphischen Editor) in der Regel frei im zweidimensionalen Raum platziert werden sowie Kanten nach Belieben gezogen werden können, ist eine weitestgehend kantenkreuzungs- und überlappungsfreie Darstellung leicht erzeugbar. Die Anzahl an optischen Anomalien ist daher minimal.

Fazit Kausalitätsgraphen stellen einen relativ simplen, aber zugleich ästhetisch ansprechenden Visualisierungsansatz dar. Zwar sind diese auf Anhieb in der Lage, sämtliche Synchronisationspunkte zu visualisieren, dennoch zeigen sich große Schwierigkeiten im Bereich der Visualisierung der Applikationsstatik, in welcher sie kein einziges Kriterium erfüllen können. Eine Erweiterung gestaltet sich jedoch aufgrund ihrer ausschließlichen Fokussierung auf die Darstellung der Dynamik als schwierig. Im Bereich der verwendeten Metapher zeigen sich zudem weitere Schwächen, da diese kaum realitätsnah, bzw. divers ist. Insgesamt gehört dieser Darstellungsansatz daher eher zu den qualitativ schwächeren Darstellungen.

4.2.6. Tupelraumdarstellung

Im Gegensatz zu sämtlichen übrigen betrachteten Visualisierungsansätzen wurden Tupelraumvisualisierungen einzig für die Darstellung eines einzigen Synchronisationsmechanismus, nämlich des Namensgebenden, konzipiert. Da jedoch kein Konsens über die exakte Darstellungsform herrscht, gestaltet sich eine ganzheitliche Evaluation dieses Ansatzes als ausgesprochen schwierig. Wir können daher lediglich unsere charakteristische Beispieldarstellung aus Abbildung 17 auf Seite 40 beurteilen.

Metapher Die Tupelraumdarstellung visualisiert einen Tupelraum sowie dessen zugehörigen Operationen auf Basis eines zweidimensionalen, graphenbasierten Ansatzes. Unsere Beispielabbildung stellt den Tupelraum in Form einer Wolke dar. Andere Visualisierungsformen verwenden hingegen auch simplere Formen wie Ellipsen oder Kreise. Derartige Darstellungen verbinden viele Betrachter dabei mit der abstrakten Umgrenzung einer Menge, vergleichbar mit einer Weidefläche, auf der die auf ihr grasenden Nutztiere umzäunt wurden. Diese Assoziation ist jedoch häufig nur sehr vage und geschieht, wenn überhaupt, eher implizit. Eine tatsächliche Nähe zu Objekten der realen Welt ist daher nicht gegeben. Der Realismus der Darstellung ist demnach gering.
Als visuelle Parameter werden Formen und Farben eingesetzt, wobei Letztere lediglich zur leichteren Identifikation der unterschiedlichen Knotentypen dienen und daher keine

Repräsentationsfunktion besitzen. Die verwendete Metapher ist daher nicht sehr ausdrucksstark. Im Sinne einer Erweiterung könnten lediglich unterschiedliche Farben und Arten von Pfeilen sinnvoll verwendet werden, wobei unklar ist, für welche Art von Informationen diese sinnvoll eingesetzt werden könnten. Die Verwendung einer differierenden Knotengröße (bei Tupel- oder Threaddarstellungen) wäre zwar ebenfalls denkbar, jedoch würde dies zum Einen die ohnehin schon geringe Kompaktheit und Skalierbarkeit der Darstellung noch weiter vermindern. Zum Anderen enthalten diese Knoten auch textuelle Komponenten, nämlich den Threadnamen sowie die Tupel-Inhalte. Bei besonders unterschiedlicher Textmenge differieren die Knoten daher ohnehin schon in ihrer Größe, weshalb unklar sein könnte, ob diese oder die beabsichtige Verwendung der Größe als virtueller Parameter die Ursache für die differierenden Knotenmaße ist.
Aufgrund der Verwendung von nur drei verschiedenen Knotentypen und nur einer Kantendarstellung ist die Diversität des Visualisierungsansatzes besonders gering. Lediglich die Einzigartigkeit des zentralen Tupelraumknotens könnte sinnvoll als markanter Bezugspunkt verwendet werden. Dem stehen jedoch eine häufig große Anzahl an uniformen Thread repräsentierenden Knoten sowie Pfeilen gegenüber, in welchen oftmals keine Ordnung oder Struktur erkennbar ist.

Statik Die Tupelraumdarstellung bietet aufgrund ihrer Spezialisierung auf das Linda-Modell keinerlei Integration von statischen Programmelementen. Entsprechend schwierig ist daher deren Erweiterung um diesen Aspekt.

Dynamik Die Tupelraumdarstellung erfüllt lediglich eines der fünf geforderten Kriterien. So zeigt die Darstellung sämtliche beteiligte Klasseninstanzen, d. h. den Tupelraum sowie die Tupel an sich. Bezüglich der Methodenaufrufe werden jedoch ungenügenderweise lediglich Tupelraumoperationen (d. h. *in* und *out*) visualisiert, jedoch keine sonstigen, von einem Thread aufgerufenen Funktionen.
Sämtliche übrigen dynamischen Elemente bleiben in der momentanen Diagrammform gänzlich unvisualisiert. So sind Laufzeiten nicht in der Darstellung codiert sowie Context-Switches oder Threadzustände nicht integriert. Jedoch lassen sich Ausführungszeiten beispielsweise durch eine textuelle Notation direkt an den repräsentierenden Ausführungspfeilen hinzufügen. Context-Switches können demgegenüber durch Sequenznummern codiert werden, wobei jeder Methodenaufruf einer ununterbrochenen Ausführungssequenz eine einheitliche Nummer erhält. Vor einem Context-Switch erhalten beispielsweise sämtliche Aufrufe die Nummer $1.x$, wohingegen die nachfolgenden Operationen mit der $2.x$ versehen werden. Da diese Benennung eindeutig ist, ist auf diese Weise auch eine globale Ordnung der Methodenausführungen definiert.

Threadzustände ließen sich im Zuge einer iterativen Wiedergabe von Programmtraces in Form einer unterschiedlicher Färbung der Thread repräsentierenden Knoten symbolisieren sowie die gesamte Darstellung in UML-Sequenzdiagramme wie in Abschnitt 4.2.1 beschrieben integrieren. Letzteres würde jedoch die Charakteristik der Darstellung verändern, da gegebenenfalls einige Visualisierungselemente des Tupelraums nicht übernommen werden können oder fundamental optisch anders dargestellt werden müssten.

Synchronisation Die Visualisierung stellt eine Repräsentation eines Tupelraum inklusive der zu diesem in Beziehung stehenden Operationen (*in* und *out*) dar. Sämtliche weitere Synchronisationsmechanismen sind von Vornherein aufgrund seines Darstellungsziels nicht in der Visualisierung integriert. Entsprechend schwierig gestaltet sich die Erweiterung der bisherigen Visualisierung um diese Mechanismen.

Funktionalität Da es sich bei der Tupelraumdarstellung um keine eigenständige Softwareanwendung handelt, ist es uns auch hier nur möglich, das Potential einer Applikation abzuschätzen, welche eine Tupelraumdarstellung ähnlich zu unserem Beispiel erzeugt.
Die Darstellung ermöglicht bislang keinerlei Repräsentation von Relationen zwischen Tupeln. Eine Erweiterung um diesen Aspekt scheint jedoch auch nicht notwendig, da häufig keine Abhängigkeiten zwischen Tupeln existieren, bzw. unklar ist, welche Arten von Relationen sinnvoll repräsentiert werden sollten. Das Kriterium der *Verknüpfung* bleibt daher unserer Ansicht nach von der Tupelraumdarstellung unerfüllt.
In sämtlichen übrigen Aspekten entspricht unsere Analyse den Ausführungen zur Funktionalität von Kausalitätsgraphen (siehe Abschnitt 4.2.5) und lassen sich unverändert übernehmen. Die Darstellung erfüllt damit insgesamt sechs der acht funktionalen Kriterien, d. h. sämtliche geforderte mit Ausnahme der *Verknüpfung* und *Zoom* werden befriedigt.

Ästhetik Vergleichbar mit der Darstellung der Kausalitätsgraphen lassen sich die beteiligten Knoten vom Benutzer auf manuelle Weise frei im zweidimensionalen Raum platzieren und die Kanten nach Belieben ziehen. Dies impliziert eine leichte Erzeugbarkeit einer weitestgehend kreuzungs- und überlappungsfreien Visualisierung. Die Darstellung wird jedoch schon bei wenigen integrierten Operationsausführungen unübersichtlich und wächst schnell in seiner Größe. Eine hohe Skalierbarkeit ist daher nicht gegeben.

Fazit Die Tupelraumdarstellung des Linda-Modells ist eine sehr Anwendungsfall spezifische Visualisierung, welche qualitativ ähnliche Schwächen wie die Kausalitätsgraphen besitzt. Einzig im Bereich des Visualisierungsumfangs ergeben sich spürbare Unterschiede. So vermag diese Darstellung bezüglich der Synchronisationspunkte lediglich das Linda-Modell geeignet zu visualisieren. Auch eine Erweiterung erscheint aufgrund der besonderen Fokussierung auf dieses Modell wenig aussichtsreich. Die Darstellung des Linda-

Modells gehört damit ebenfalls zu den eher schwächeren der hier betrachteten Visualisierungsansätze.

4.2.7. Übersicht über die Evaluationsergebnisse

Nachfolgend möchten wir unsere Evaluationsergebnisse in einer handlichen Form zusammenfassen. Die Tabellen 2 und 3 zeigen diese. Erfüllt ein Visualisierungsansatz ein Kriterium erst im Zuge einer im zugehörigen Abschnitt vorgeschlagenen Erweiterung, ist deren Repräsentation, bzw. das Ergebnis der Evaluation, eingeklammert. Andernfalls wurde jeweils der Ist-Zustand referenziert, respektive bewertet. Unsere Ausführungen stellen daher ebenfalls eine Beurteilung des Potentials der betrachteten Visualisierungen im Sinne ihrer Erweiterbarkeit dar.
Leider ist als Endergebnis unserer Beurteilung kein eindeutiger Sieger festzustellen. So lassen sich zwar mit der (erweiterten) UML sämtliche Vollständigkeitskriterien erfüllen, jedoch zeigt dieser Visualisierungsansatz z. T. deutliche Schwächen bezüglich der Ästhetik und der verwendeten Metapher. EvoSpaces schneidet hingegen in diesen beiden Bereichen gut bis sehr gut ab. Es mangelt diesem Ansatz jedoch an der Darstellbarkeit von dynamischen Programmaspekten. So zeigt EvoSpaces diesbezüglich lediglich Methodenaufrufe. Synchronisationspunkte lassen sich symptomatischerweise gar nicht in die Visualisierung integrieren.
Die Tupelraumdarstellung hat über sämtliche Kriterien hinweg am schlechtesten abgeschnitten und ist der eindeutige Verlierer unserer Evaluation. Sie zeigt einzig in den Bereichen der Ästhetik und der Vollständigkeit der Dynamik akzeptable Eigenschaften.

(Unter-)Kategorie	Kriterium	UML inkl. Erweiterungen	EvoSpaces	DyVis
Metapher	Bezeichnung	graphenbasiert	städtisch	städtisch
	Dimensionalität	2D	3D	3D
	Realismus	−	++	+
	Diversität	+/−	+	+
	Ausdrucksstärke	+/−	+	++
Statik	Pakete	Knoten	Distrikte	Distrikte
	Klassen	Knoten	Gebäude	Gebäude
	Assoziationen	Linien	Linien	(Linien)
	Ableitungen	Linien	Linien	(Linien)
	Attribute	textuelle Knoteneinträge	(Gebäudebreite)	Gebäudebreite od. -länge
	Methoden	textuelle Knoteneinträge	Stichmännchen	Gebäudebreite od. -länge
Dynamik	Klasseninstanzen	Knoten	−	Etagen
	Methodenaufrufe	Linien	Linien zw. Gebäuden	Linien zw. Etagen
	Ausführungszeit	Skala	−	Farbe der Linien
	Context-Switches	Linien	−	−
	Threadzustände	Diagramm	−	−
Synchronisationspunkte	Semaphore	Knoten	−	−
	Monitore	Knoten	−	−
	Thread-Joins	Linien	−	−
	Tupelräume	(Knoten)	−	−
	Wait / Notify	Linien	−	−
	Kritische Sektoren	Umrahmung	−	−
Funktionalität	Übersicht	Ja	Ja	Ja
	Zoom	Ja	Ja	Ja
	Filter	(Ja)	Nein	Nein
	Details-on-Demand	(Ja)	Nein	Ja
	Verknüpfung	Ja	Ja	Ja
	Historie	(Ja)	Nein	Nein
	Extraktion	(Ja)	Nein	Nein
	Wiedergabe	(Ja)	Ja	Ja
Ästhetik	Kompaktheit	−−	+	+
	Kantenüberschneidungen	++	+	+
	Grafische Überlappungen	++	+	+/−

Tabelle 2: Übersicht über die Evaluationsergebnisse, Teil 1

(Unter-)Kategorie	Kriterium	TraceCrawler	Kausalitätsgraph	Tupelraumdarstellung
Metapher	Bezeichnung	graphenbasiert	graphenbasiert	graphenbasiert
	Dimensionalität	3D	2D	2D
	Realismus	+/−	−−	−
	Diversität	+	−−	−
	Ausdrucksstärke	++	+/−	+/−
Statik	Pakete	(Verschachtelte Grundflächen)	−	−
	Klassen	Knotenstapel	−	−
	Assoziationen	(Linien zw. Basisknoten)	−	−
	Ableitungen	Linien zw. Basisknoten	−	−
	Attribute	Maße u. Farbe der Basisknoten	−	−
	Methoden	Maße u. Farbe der Basisknoten	−	−
Dynamik	Klasseninstanzen	Knoten	−	−
	Methodenaufrufe	Linien zw. Instanzknoten	Knoten	Linien
	Ausführungszeit	(Farbe der Linien)	(Farbe der Knoten)	Zahl
	Context-Switches	−	(Sequenznummer)	(Sequenznummer)
	Threadzustände	(Farbe der Knoten)	−	Farbe der Knoten
Synchronisationspunkte	Semaphore	(Knoten)	Knoten	−
	Monitore	(Knoten)	Knoten	−
	Thread-Joins	(Knoten u. Linien)	Knoten	−
	Tupelräume	−	Knoten	Knoten
	Wait / Notify	(Knoten u. Linien)	Knoten	−
	Kritische Sektoren	−	(Umrahmung)	−
Funktionalität	Übersicht	Ja	Ja	Ja
	Zoom	Nein	Nein	Nein
	Filter	Nein	(Ja)	(Ja)
	Details-on-Demand	Ja	(Ja)	(Ja)
	Verknüpfung	Ja	Ja	Nein
	Historie	Nein	(Ja)	(Ja)
	Extraktion	Nein	(Ja)	(Ja)
	Wiedergabe	Ja	(Ja)	(Ja)
Ästhetik	Kompaktheit	+	−	−
	Kantenüberschneidungen	−−	++	++
	Grafische Überlappungen	−	++	++

Tabelle 3: Übersicht über die Evaluationsergebnisse, Teil 2

5. Entwicklung einer Visualisierung

Nachdem wir die in Kapitel 3 aufgezeigten, bisherigen Visualisierungsansätze evaluiert haben, können wir nun mit der Entwicklung unser eigenen Darstellungsform der Statik, Dynamik und Synchronisationspunkte beginnen. Aufgrund der mangelnden Skalierbarkeit sämtlicher zweidimensionalen Darstellungen haben wir uns dabei für einen dreidimensionalen Ansatz auf Basis der städtischen Metapher (siehe Abschnitt 3.2) entschieden. Da unsere Visualisierung zudem auch dynamische Thread-Interaktionen, d. h. insbesondere auch Deadlock-Szenarien, leicht verständlich darstellen können muss, ist eine Repräsentation von einzelnen Klasseninstanzen unerlässlich. Dementsprechend ergab sich die Wahl zwischen DyVis [65] und TraceCrawler [32, 33] als Basisvisualisierung. DyVis ist jedoch TraceCrawler gemäß unserer Evaluation qualitativ überlegen. So verfügt DyVis gerade im Bereich der verwendeten Metapher über zahlreiche positive Eigenschaften und schneidet in dieser Kategorie besser ab als TraceCrawler. Besonders negativ sind bei TraceCrawler jedoch seine ästhetischen Charakteristika anzumerken. So stellen häufige Kantenüberschneidungen ein eher typisches Erscheinungsbild dieser Visualisierung dar. Wir haben uns daher letztendlich für DyVis als Grundlagendarstellung in unser eigenen Visualisierung entschieden.
Nachfolgend möchten wir unsere eigene Visualisierung näher vorstellen. Unser Ansatz vereinigt dabei letztendlich die Darstellung von DyVis mit Aspekten von EvoSpaces [13, 14, 15, 27] sowie einigen Erweiterungen der UML, dabei insbesondere derjenigen von Malnati et. al. [48, 49] und Jacot [44, 45, 46]. Unser Softwarewerkzeug *Synchrovis*, kurz für *Synchronization Visualizer*, implementiert diese Darstellungsform prototypisch und ist in der Lage, unsere Visualisierung aus einem *Knowledge Discovery Meta-Modell* [1] und einem mit dem *KIEKER Monitoring & Analysis Framework* [34, 37] aufgezeichneten Programmtrace zu erzeugen. Wir geben im Folgenden sowohl ein Metamodell dieser Visualisierung als auch die zur Speicherung der benötigten Informationen verwendeten Datenstruktur an.

5.1. Eine Visualisierung der Statik, Dynamik und Synchronisationspunkte

Wir untergliedern unsere textuelle Beschreibung der Visualisierung analog zu unseren Evaluationskriterien der Vollständigkeit in die drei Bereiche Statik, Dynamik und Synchronisationspunkte. In jeder Kategorie geben wir dabei eine detaillierte Beschreibung unserer Visualisierung an, welche wir durch verdeutlichende Konzeptdarstellungen ergänzen. Auch obwohl unsere Visualisierung eine dreidimensionale Darstellung ist, werden wir diese Abbildungen aus Gründen der Übersichtlichkeit und Verständlichkeit lediglich in 2D präsentieren.

5.1.1. Visualisierung der Statik

Die Statik einer Anwendung stellen wir weitestgehend identisch zu DyVis dar. Gebäude repräsentieren objektorientierte Klassen, deren Instanzen durch einzelne Stockwerke symbolisiert werden. Das Erdgeschoss zeigt jeweils das Klassenobjekt und wird farblich abgestuft gegenüber den darüber visualisierten, dynamisch erzeugten Objekten dargestellt. Die Gebäude besitzen dabei eine variabel große quadratische Grundfläche, welche sich mit steigender Anzahl der Methoden innerhalb der Klasse stufenweise vergrößert. Im Gegensatz zu DyVis beziehen wir Attribute nicht mittels einer Metrik in die Visualisierung mit ein und lassen diese sogar gänzlich unvisualisiert. So erscheint uns zum Einen die Methodenanzahl ein realistischeres Maß für die Mächtigkeit einer Klasse im Sinne ihres Funktionsumfangs. Zum Anderen halten wir auch die Repräsentation von Attributen für nicht sinnvoll, da unsere Visualisierung vor Allem das Verständnis von nebenläufigen Programmabläufen erhöhen soll und Attribute hierzu keinen Beitrag leisten.
Das Konzept zur Codierung von Ableitungs- und Assoziationsbeziehungen übernehmen wir unmittelbar aus EvoSpaces und zeigen diese mittels Gebäudedächern verbindende, gerichtete Kanten. Schwarze Kanten werden hierbei für Ableitungen, graue für Interface-Implementierungen und weiße für Assoziationen verwendet. Da die Darstellung statischer Relationen zu besonders vielen Kanten(-kreuzungen) führen kann, sollte deren Visualisierung jedoch lediglich optional eingeblendet werden können.
Pakete werden in Form von übereinanderliegenden, rechteckigen Ebenen mit fester Höhe symbolisiert, welche auf einer gemeinsamen Grundfläche aufliegen. An ihrer Farbhelligkeit lässt sich dabei die Verschachtelungstiefe eines Pakets unmittelbar ablesen. Jedes Gebäude wird gemäß seiner Paketzugehörigkeit auf der dazugehörigen Ebene platziert. Abbildung 18 zeigt eine schematische Skizze dieses Aufbaus.

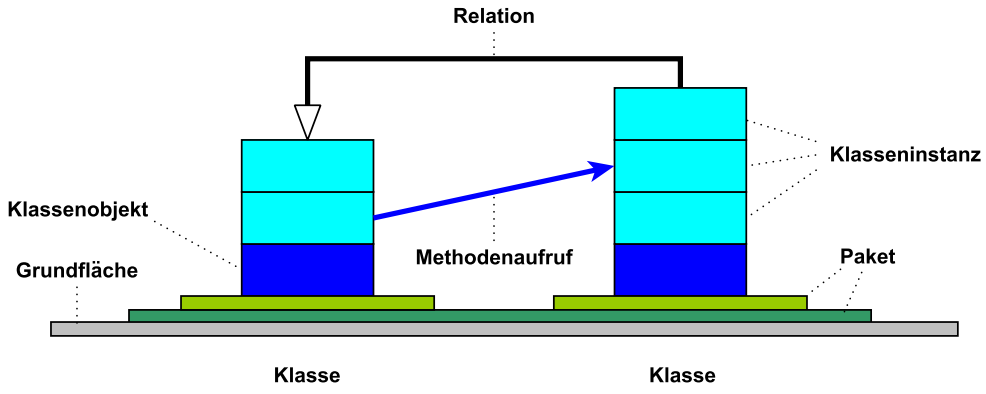


Abbildung 18: Darstellung der Statik und Dynamik in unserer Visualisierung

5.1.2. Visualisierung der Dynamik

Methodenaufrufe visualisieren wir in Form von gerichteten Kanten zwischen aufrufendem und aufgerufenem Objekt, d. h. zwischen zwei nicht notwendigerweise verschiedenen Gebäudeetagen. Eine auf ein Erdgeschoss gerichtete Kante repräsentiert dabei den Aufruf einer statischen Methode, bzw. des Konstruktors. Wird ein Konstruktor aufgerufen und daher eine neue Klasseninstanz zur Laufzeit erstellt, wächst ebenso das korrespondierendes Gebäude dynamisch um eine Etage. Im Vorfeld der Visualisierung haben wir jedem Thread eine eindeutige Farbe zugewiesen. Jede Aufruf symbolisierende Kante wird dabei in der Farbe des ausführenden Threads gefärbt (siehe Abbildung 18).
Zur Repräsentation der aktuellen Threadzustände verwenden wir ein eigenständiges Gebäude, welches auf einer neuen, abseits gelegenen Grundfläche platziert wird und durch ein zusätzliches Pfeilsymbol auf dem Dach leicht von übrigen Gebäuden zu unterscheiden ist. Dessen Erdgeschoss symbolisiert dabei den Hauptausführungsstrang, d. h. den durch die *main*-Methode ausgelösten Programmablauf. Dagegen zeigt jede darüber liegende Etage eine Instanz eines Threads. An den Außenwänden jedes Stockwerks befindet sich dabei ein mittig platziertes Plakat, welches in der Farbe des zugehörigen Threads gefärbt ist, solange dieser aktiv ist. Terminiert ein Thread jedoch, nimmt dieses vollständig die Farbe des Stockwerks an und ist optisch nicht mehr vom Hintergrund zu unterscheiden. Die Erzeugung eines neuen Threads wird ebenfalls durch einen vom aufrufenden Objekt auf das Erdgeschoss gerichteten Pfeil angedeutet.
Der erste Methodenaufruf eines jeden Ausführungsstrangs (Thread oder Hauptprogramm) wird jeweils durch einen vom Thread-Gebäude ausgehenden Aufrufpfeil symbolisiert. Abbildung 19 zeigt dieses Szenario. Die eingezeichnete *umschließende Ebene* besitzt dabei keine Repräsentationseigenschaft und dient lediglich zur Wahrung des Realismus und der Symmetrie der Darstellung. So ist stets jedem Gebäude einem Distrikt zugeordnet.

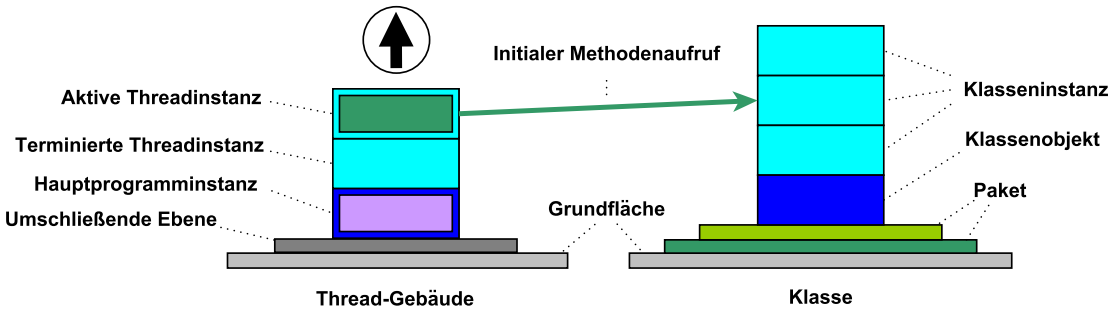


Abbildung 19: Darstellung von Threads inklusive initialem Methodenaufruf

5.1.3. Visualisierung der Synchronisationspunkte

Synchronisationspunkte visualisieren wir mit Hilfe eigenständiger Gebäude, welche auf der selben umschließenden Ebene platziert werden, auf welcher sich bereits das Thread-Gebäude befindet. Da Threads in Kombination mit den Synchronisationsmechanismen sämtliche Programmausführungen initiieren, bzw. steuern, können wir diese Gebäude in Kombination mit der gemeinsamen Grundfläche auf passende Weise mit dem Verwaltungsdistrikt der Stadt assoziieren.

Semaphore und Monitore Im Rahmen dieser Arbeit betrachten wir stets nur binäre Semaphore. Daher sehen wir Semaphore und Monitore als weitestgehend identische Konstrukte an und visualisieren diese dementsprechend in Form eines einzigen, gemeinsamen Gebäudes. Ein Vorhängeschlosssymbol auf dessen Dach ermöglicht dabei dessen sofortige Identifikation innerhalb der Stadt. Jedes Stockwerk repräsentiert eine einzelne Semaphor- bzw. Monitorinstanz, welche als diejenigen Objekte definiert sind, welche zur Absicherung der kritischen Sektion als Sperrvariable verwendet werden. Das Erdgeschoss des Gebäudes wird nicht individuell hervorgehoben, da das Gebäude keine eigenständige Klasse sondern lediglich eine Menge von verwendungsgleichen Objekten repräsentiert. Analog zum Thread-Gebäude wird auch hier auf jede Etagenaußenwand ein Plakat angebracht und szenarioabhängig gefärbt. Ist eine Monitor- oder Semaphor-Instanz belegt, so trägt dieses die Farbe des belegenden Threads. Andernfalls nimmt das Plakat die reguläre (hier blaue) Hintergrundfarbe an.
Der Aufruf der p-Operation (bei Semaphoren), respektive die Belegung der Sperrvariable (bei Monitoren) wird durch einen Methodenaufruf vom initiierenden Objekt zur entsprechenden Etage dieses Gebäudes verstanden und dementsprechend mit Hilfe einer durchgezogenen, gerichteten Kante visualisiert. Analog wird ebenso die v-Operation, bzw. die Freigabe der Sperrvariablen, als Funktionsaufruf dargestellt, jedoch verwenden wir hierfür eine gestrichelte Kante.
Abbildung 20 zeigt ein Beispiel eines Szenarios, in welchem der grüne Thread gerade die mittlere Semaphore belegt hat. Die obere Instanz ist jedoch noch dem purpurnen Thread zugeordnet, sodass der Thread nicht mit seiner Ausführung fortfahren kann. Aus Übersichtsgründen haben wir hierbei das Thread-Gebäude nicht dargestellt. Dieses wird jedoch gemeinsam mit dem Semaphor- und Monitor-Gebäude auf einer gemeinsamen umschließenden Grundfläche platziert.
Zwar werden durch die eigenständige Repräsentation einer Monitor-Instanz in einen separaten Gebäude gegebenenfalls einige (Laufzeit-)Objekte zweifach visualisiert, falls diese ebenso Instanzen der regulären Klassen sind. Da es jedoch beliebig ist, ob das ein oder andere Objekt als Sperrvariable verwendet wird, ist eine (Wieder-)Verwendung der regulären Darstellungen der Objektinstanzen nicht sinnvoll. So ließen sich andernfalls beispielsweise keine *String*-Objekte, welche im Programm nicht als eigenständig deklarierte Klasse vorliegen, als Monitor-Instanz sinnvoll darstellen. Durch die Integration sämtlicher

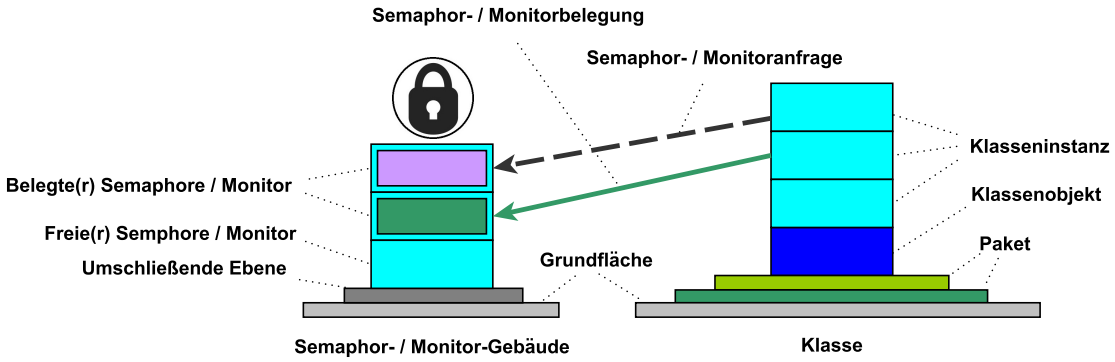


Abbildung 20: Darstellung von Semaphor- und Monitor-Operationen

Semaphore und Monitore in einem einzigen Gebäude wird zudem eine Übersichtlichkeit geschaffen, in welcher Deadlocks auf einen Blick anhand der unterschiedlichen Farbgebung der Plakate und Methodenaufrufpfeile identifiziert werden können.

Wait und Notify Wait- und Notify-Operationen visualisieren wir analog zu den Semaphore- und Monitor-Instanzen, verzichten jedoch auf die Darstellung von Plakaten. In einem weiteren separaten Gebäude repräsentiert jede Etage ein Objekt, auf welches zur Laufzeit entweder die *wait*-, *notify*-, oder *notifyAll*-Methode aufgerufen wurde. Es wird durch ein Handsymbol auf dem Dach gekennzeichnet. Dieses erinnert uns dabei an eine menschliche „Stopp!“-Geste und erzeugt daher, wie bereits das Vorhängeschlosssymbol des Semaphor- und Monitor-Gebäudes und das Pfeilsymbol des Thread-Gebäudes, eine passende Assoziation mit diesem Synchronisationskonstrukt. Potentiell suspendierende Aufrufe, d. h. *wait*-Operationen, visualisieren wir dabei vergleichbar mit der Darstellung der Monitor-/Semaphor-Methoden mittels eines gestrichelten, die beiden übrigen Aufrufe hingegen in Form eines durchgezogenen Pfeils.
Abbildung 21 zeigt ein Beispiel, in welchem der schwarz Thread auf einem von *wait* genutztem Objekt suspendiert wurde. Auf diesem ruft nun der grüne Thread die *notify*-Methode auf. Aus Gründen der Übersichtlichkeit haben wir auch in dieser Darstellung das Thread- und das Semaphor-/Monitor-Gebäude nicht integriert.

Thread-Joins und das Linda-Modell Thread-Joins visualisieren wir unmittelbar mit Hilfe des Thread-Gebäudes. Wartet ein Thread (oder das Hauptprogramm) auf die Terminierung eines Threads, repräsentieren wir dies durch einen Methodenaufruf auf die entsprechende Etage dieses Gebäudes in Form eines gestrichelten Pfeils. Auf die Beendigung des Hauptprogramms kann dabei niemals gewartet werden. Ein Pfeil auf das Erdgeschoss codiert daher den Start eines neuen Threads, worauf das Gebäude um eine

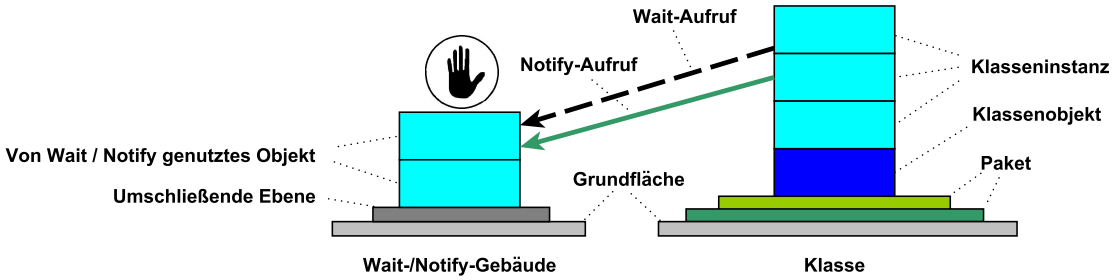


Abbildung 21: Darstellung von Wait und Notify

Etage wächst. Das Linda-Modell ließ sich hingegen nicht sinnvoll in unseren Visualisierungsansatz integrieren.

5.2. Metamodell der Visualisierung

Abbildung 22 zeigt das Metamodell unserer Visualisierung und definiert es auf eine formale Weise. Auch trotz der Verwendung der von DyVis [65] erzeugten Darstellung als Ausgangsbasis, zeigen sich große Unterschiede zwischen dessen und unserem Metamodell.[4]
Jedes visuell sichtbare Element in unserer Visualisierung (in Abbildung 22 gelb eingefärbt) ist von einer der drei abstrakten Entitäten *sichtbares Element*, *2D-Element*, bzw. *3D-Element* abgeleitet. Diese repräsentieren dabei die Grunddatenstrukturen unserer Darstellung und erweitern sich gegenseitig auf eine sinnvolle Weise. So besitzt erstere Klasse eine eindeutige ID, Farbe und Position in der dreidimensionalen Zeichenfläche, Zweitere zusätzlich eine Breite und Länge und Letztere eine zusätzliche Höhe.
Die spätere Visualisierungsinstanz entspricht dabei dem *Stadt*-Objekt, welches im Schaubild grün dargestellt ist. Sämtliche optischen Elemente sind in diesem mittels Kompositionsbeziehungen inkludiert. In unserer Visualisierung sind dabei zwei verschiedene Grundflächen definiert und stets sichtbar: Die *reguläre* und die *Verwaltungsdistriktsgrundfläche*. Erstere dient dabei der Repräsentation der Pakete der betrachteten Applikation. So besitzt jede reguläre Grundfläche mindestens einen *regulären Distrikt*, welche wiederum weitere Distrikte als Untergliederung beinhalten dürfen und als dreidimensionale Objekte visualisiert werden. Jedem regulären Distrikt sind dabei diejenigen *reguläre Gebäude* zugeordnet, welche eine im entsprechenden Paket enthaltene Klasse repräsentieren.
Gebäude verstehen wir dabei initial als zweidimensionale Gebilde. Erst durch deren Komposition mit mindestens einer *Etage* werden aus diesen mittels des Höhenparameters dreidimensionale Objekte. Reguläre Gebäude besitzen dabei in Form des Erdgeschosses stets eine *reguläre Etage*, welche das jeweilige Klassenobjekt visualisiert. Zwischen regulären

[4] vgl. [65, S. 35]

Gebäuden können dabei (u. U. mehrere) *Kabel* verlaufen, welche als gerichtete Kanten ein Start- und ein Zielobjekt besitzen und abhängig von ihrer Färbung Assoziations- (schwarz), Interface-Implementierungen (grau), bzw. Ableitungsbeziehungen (weiß), verkörpern. In der späteren Visualisierung verbinden diese die Dächer zweier (im Fall von Assoziationen nicht notwendigerweise verschiedener) Gebäude.

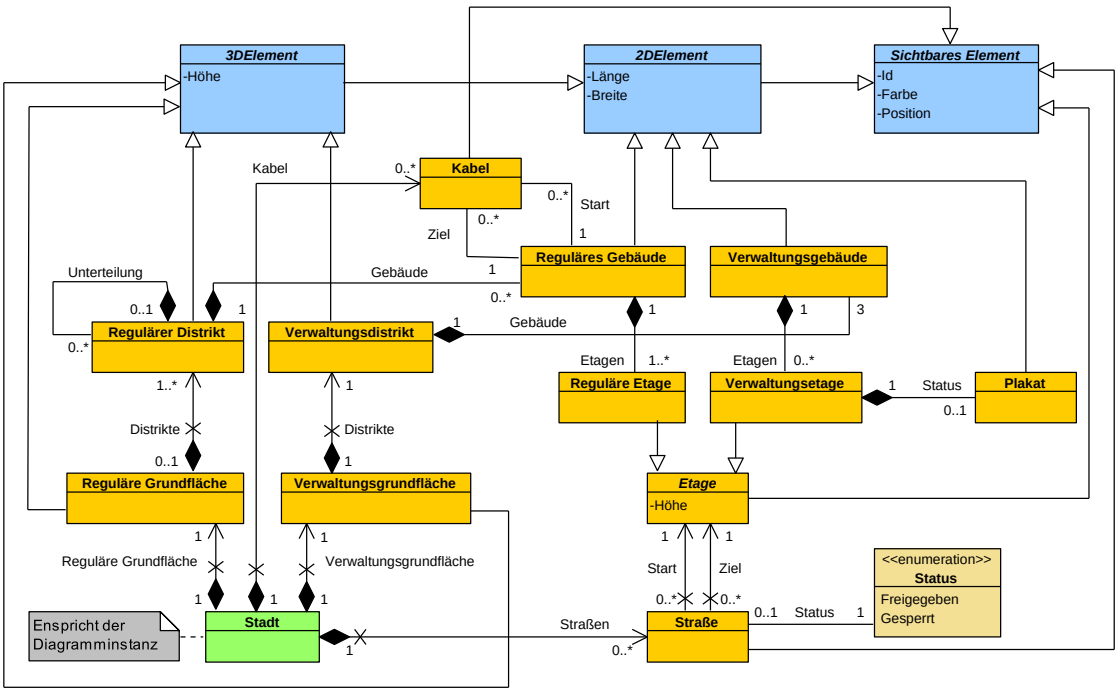


Abbildung 22: Das Metamodell unserer Visualisierung

Die *Verwaltungsdistriktsgrundfläche* besitzt hingegen lediglich eine aufliegende Ebene, den so genannten *Verwaltungsdistrikt*. Diese besitzt an sich keinerlei Repräsentationsfunktion, jedoch wird auf dieser das in Anschnitt 5.1 beschriebene Thread-, Semaphor-/Monitor- sowie das Wait-/Notify-Gebäude platziert. Diese bestehen wiederum aus beliebig vielen (d. h. auch keine) *Verwaltungsetagen* und sind durch ein einzigartiges Symbol auf ihrem Dach gekennzeichnet. An jeder der vier vertikalen Außenwände einer Verwaltungsetage ist dabei ein identisches farbiges *Plakat* angebracht, welches den aktuellen Status des durch das Stockwerk repräsentierten Objekts codiert. Zwischen zwei Etagen eines beliebigen Typs können dabei farbig markierte *Straßen* in Form von gerichteten Kanten verlaufen, welche jeweils einen Methodenaufruf symbolisieren. Jede von diesen besitzt dabei einen *Status*. So werden *gesperrte* Straßen in der späteren Visualisierung in Form von gestrichelten, *freigegebene* hingegen in Form von durchgezogenen Kanten abgebildet.

Gemäß unseres Metamodells stellt bereits die alleinige Darstellung der Applikationsstatik eine gültige Visualisierung dar. In dieser werden Klassen durch ihre Klassenobjekte und Assoziations- und Ableitungsbeziehungen durch Kabel repräsentiert. Diese Darstellung ist vor Allem dann sinnvoll, wenn man lediglich an einer Veranschaulichung des strukturellen Aufbau einer Applikation interessiert ist und die dynamischen Vorgänge bei deren Ausführung vernachlässigen möchte. Die Integration der Dynamik verstehen wir daher stets als ein optionales Feature unserer Visualisierungsform.

5.3. Metamodell der Datenhaltung

Abbildung 23 zeigt unser Metamodell der Datenhaltung, welches wir zur Speicherung der Visualisierung zu Grunde liegenden Applikationsinformationen verwenden. Die *Datensatz*-Klasse repräsentiert hierbei eine Instanz dieses Modells und beinhaltet neben dem Wurzelpaket der Anwendung ebenso sämtliche aus den Kieker-Dateien importierten Programmtraces.
Jeder *Trace* besitzt dabei die angegebenen Attribute. So repräsentiert die *Trace-Id* die ihm eindeutig zugewiesene Identifikationsnummer und der Trace ist demjenigen Thread zugehörig, welcher auf dem Zielcomputer intern durch die angegebene *Thread-Id* bezeichnet wurde. Die *Eltern-Trace-Id* gibt hingegen die Id seines Eltern-Traces an, falls dieser von einem anderen Trace abstammt. Dies ist beispielsweise dann der Fall, wenn er die Ausführung eines Threads darstellt und von einem anderen Thread aus gestartet wurde. Der *Eltern-Ordnungsindex* gibt dann den Ordnungsindex desjenigen Trace-Events an, in welchem es zu dieser Abspaltung des Programmflusses kam. Andernfalls betragen diese beiden Werte minus eins. Die Aufzeichnung des Programmtraces fand dabei auf einem Computer statt, dessen Bezeichnung unter *Hostname* eingetragen ist. Im Fall von Internet basierten Anwendungen ist zudem eine *Session-Id* angegeben, welche die zum Programmablauf zugehörige Client-Server-Kommunikationssitzung eindeutig identifiziert.
Ein Trace bezeichnet dabei eine Sequenz zusammengehöriger Operationsausführungen und Ereignisse, welche wir *Trace-Events* nennen. Wie sämtliche im Diagramm in blau dargestellte Entitäten stellen diese ein abstraktes, d. h. nicht instantiierbares, Konstrukt dar und besitzen neben den klassischen Attributen wie der Id des zugehörigen Traces (*TraceId*), ihrem *Zeitstempel* sowie ihrer *Verschachtelungstiefe* im Aufrufgraph sowohl einen *lokalen* als auch einen *globalen Index*. Ersterer gibt dabei die Trace-lokale Ordnung aller inkludierten Events gemäß ihres Zeitstempels an, wohingegen Letzterer die Events auch über die Tracegrenzen hinweg global chronologisch ordnet.

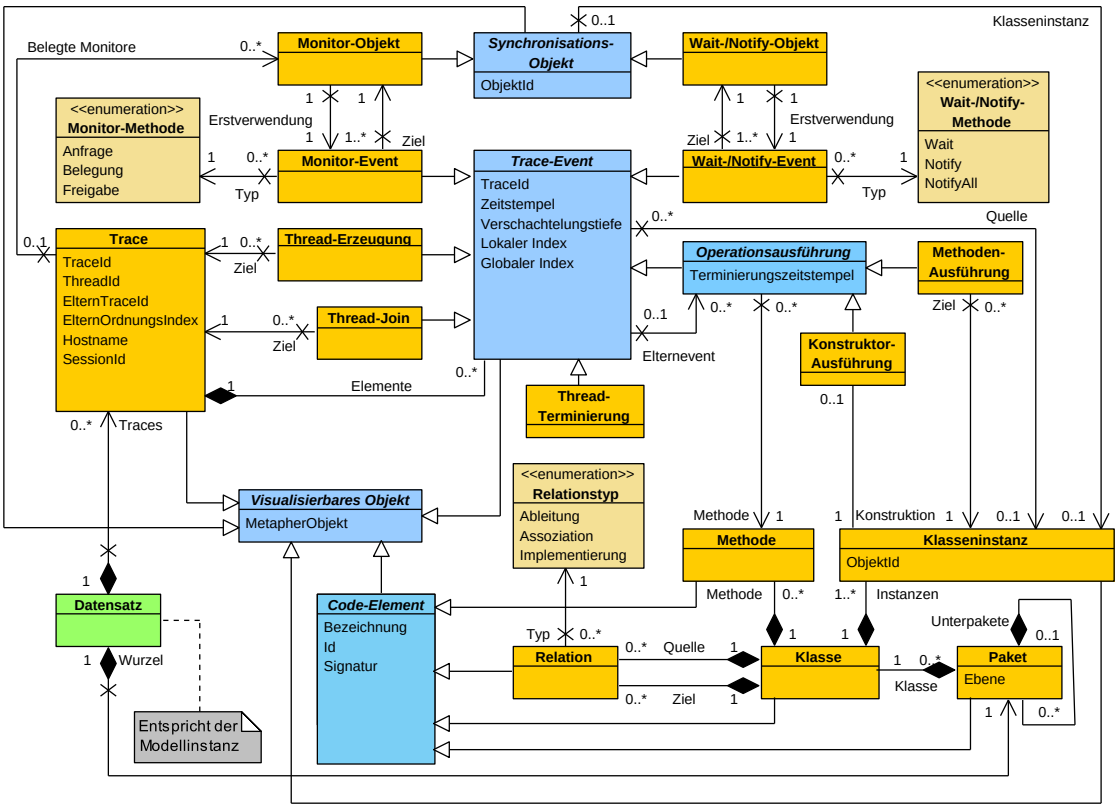


Abbildung 23: Das Metamodell der Datenhaltung

Jedes Event wurde dabei von derjenigen *Klasseninstanz*[5] initiiert, auf welches sein *Quelle*-Attribut verweist. Es definiert im Sinne der Applikationsdynamik dasjenige Objekt, welches während der Ereignisauslösung den Kontrollfluss innehielt. Ist diese Referenz undefiniert (*null*), so identifiziert diese das erste Ereignis im Trace, welches im Zuge der Ausführung häufig unmittelbar von der *main*-, bzw. *run*-Methode des Prozesses oder Threads initiiert wurde.

Ein Trace-Event kann dabei eine von insgesamt sieben Ausprägungen annehmen, welche die gleichnamigen Ereignisse repräsentieren. So codiert ein *Monitor-Event* entweder die *Anfrage*, *Belegung* oder *Freigabe* eines Monitors. Die exakte Operation ist dabei durch den beigefügten Typ *Wait-/Notify-Methode* definiert. Auf analoge Weise sind *Wait-/Notify-Events* mittels der Enumeration *Wait-/Notify-Methode* realisiert. Dasjenige Objekt, auf welches synchronisiert wird dabei in Form eines *Synchronisationsobjekts* im *Ziel*-Attribut gespeichert. Gleichzeitig besitzt dieses Objekt aber auch eine Referenz auf dasjenige Er-

[5] Im Sinne dieses Metamodells bezeichnen wir sowohl die mittels eines Konstruktors erzeugten Instanzen einer Klasse als auch das Klassenobjekt an sich als *Instanz*.

eignis, bei welches es zum ersten mal aufgerufen wurde. Dies ist vor Allem dann nützlich, wenn der Benutzer rückwärts durch den Trace navigiert und zur Realisierung einer konsistenten Abbildung Synchronisationsobjekte aus der Darstellung wieder entfernt werden müssen. Auf vergleichbare Weise besitzt ebenso jede Thread-Erzeugung sowie Thread-Joins eine *Ziel*-Referenz, welche jeweils auf den resultierenden, bzw. „gejointen" Trace zeigen.
Operationsausführungen besitzen neben dem (Initiierungs-) Zeitstempel des Trace-Events ebenso einen *Terminierungszeitstempel*, welcher den Zeitpunkt des Verlassens des Methodenrumpfes angibt. Wir unterschieden bei diesen zwischen den *Methoden-* und *Konstruktorausführungen*. Erstere repräsentieren reguläre Operationsabarbeitungen mit der *Ziel*-Klasseninstanz als dasjenige Objekt, auf welchem dieser Aufruf stattfindet (zum Beispiel o bei $o.foo()$). Letztere hingegen symbolisiert die Erzeugung eines neuen Objekts, dessen Klasse in der Anwendungsstatik definiert ist. Das konstruierte Objekt wird dabei bidirektional mit dem Trace-Ereignis verbunden sowie bei beiden Ausführungsarten die aufgerufene Methode im *Methode*-Attribut referenziert.
Thread-Terminationen markieren das Ende der Operationsausführungen eines Threads und werden als letztes Ereignis an seinen finalen Trace angehängt. Sie repräsentieren dabei die einzigen Ereignisse, welche nicht vom Kieker-Framework aufgezeichnet wurden, sondern von uns eigenständig hinzugefügt werden. Da die Terminierung eines Threads eine Änderung des Visualisierungszustands (genauer: das Ausblenden der Thread-Poster) zur Folge hat, ist deren Addition zur Schaffung einer konsistenten Visualisierung erforderlich.
Sämtliche Elemente der Statik sind von der Klasse *Code-Element* abgeleitet und repräsentieren ihre gleichnamigen Softwareentitäten mit angegebener *ID*, *Bezeichnung* sowie *Signatur*. Das Wurzelpaket definiert dabei den gesamten Datensatz der Statik, da in diesem sämtliche übrige Elemente inkludiert sind. So kann ein *Paket* aus beliebig viele Kindpaketen sowie *Klassen* bestehen, wobei Letztere wiederum aus *Klasseninstanzen* und *Methoden* aufgebaut sind. Zwischen einzelnen Klassen können dabei gegenseitige Abhängigkeiten (*Relationen*) bestehen, welche bidirektional traversiert werden können und deren Typ durch eine Aufzählungskonstante aus der Enumeration *Relationstyp* notiert wird. Unsere Datenstruktur unterscheidet dabei zwischen *Assoziationen*, *Ableitungen* und Interface-*Implementierungen*.
Sämtliche Datenobjekte, welche später mittels eines optischen Elements visualisiert werden können, sind von der Klasse *Visualisierbares Objekt* abgeleitet und verfügen damit über eine direkte Anbindung an ihr jeweiliges Metapher-Objekt. Diese Referenz repräsentiert dabei die einzige feste Verbindung zwischen unseren beiden Metamodellen.

6. Synchrovis - Eine prototypische Implementierung

Im Rahmen dieser Arbeit haben wir den Prototypen *Synchrovis* (kurz für *Synchronization Visualizer*) entwickelt, welcher aus Daten zur Applikationsstatik und -dynamik die in Kapitel 5 beschriebene Visualisierung zu erzeugen vermag und dessen Quellcode dieser Arbeit im Anhang B beigefügt ist. Nachfolgend geben wir zunächst einen kurzen Überblick über diese Anwendung und stellen dessen Funktionalität detailliert vor. Eine Präsentation der Architektur sowie der bei der Implementierung verwendeten Technologien folgt im Anschluss.

6.1. Übersicht

Abbildung 24 zeigt das Hauptfenster unserer Applikation. In dessen oberen Bereich ist die Menüleiste angeordnet ist, welche dem Benutzer die übergeordnete Steuerung von Synchrovis (u. A. das Laden und Speichern der aktuellen Visualisierung) ermöglicht. Auf der rechten Seite befindet sich die *Trace-Ansicht*, welche die aktuell visualisierten Programmabläufe abhängig von der Einstellung der darüberliegenden *Trace-Layout-Leiste* wahlweise in chronologischer oder Aufruf-hierarchischer Sicht (Näheres in Abschnitt 6.2.4) darstellt und so stets eine Übersicht über die aktuell geladen Trace-Elemente bietet. Die *Darstellungszeichenfläche* im linken, mittleren Bereich repräsentiert das größte von ei-

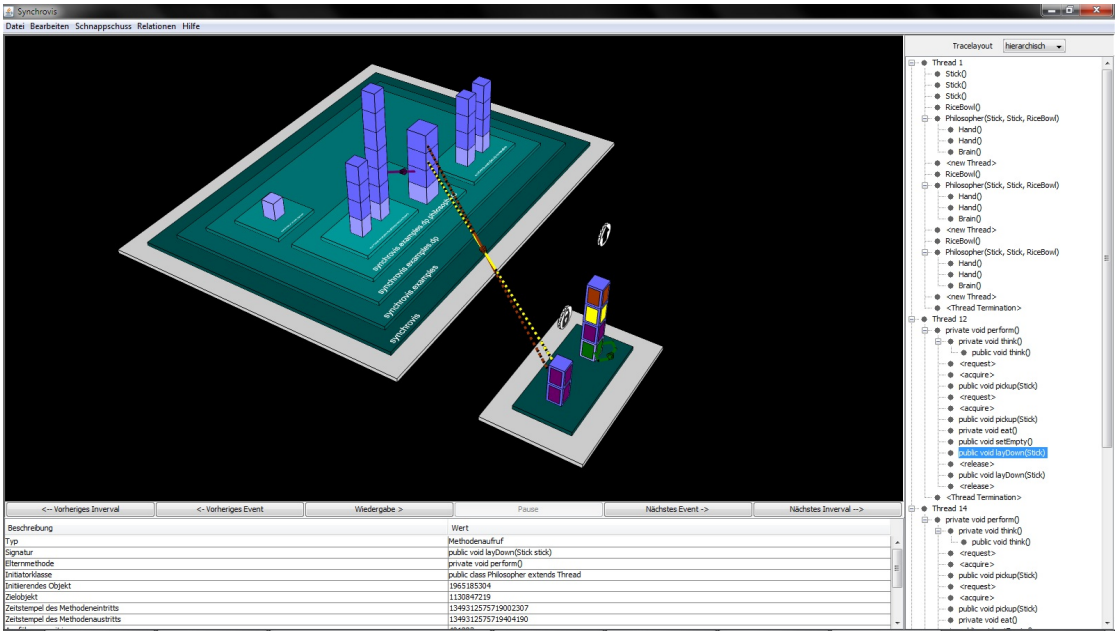


Abbildung 24: Das Hauptfenster von Synchrovis

ner einzigen Komponente belegte Areal und zeigt die dreidimensionale Darstellung einer betrachteten Applikation gemäß unseres beschriebenen Visualisierungsansatzes. Sie wird dabei mittels der darunter befindlichen Steuerungsleiste bedient und ermöglicht sowohl einer manuelle als auch automatisierte Wiedergabe des geladenen Programmablaufs. Der Benutzer kann dabei beinahe jedes optische Element der Darstellung per Mausklick auswählen und dessen Charakteristika in der *Info-Tabelle* am unteren Fensterrand anzeigen lassen.

6.2. Funktionalität

Abbildung 25 zeigt einen Überblick über sämtliche, momentan in Synchrovis implementierten Benutzerinteraktionsmöglichkeiten in Form eines vereinfachten UML-Anwendungsfalldiagramms. Ähnliche Funktionen sind in diesem jeweils zu gleichen Kategorien zusammengefasst. Wir werden diese im Folgenden näher erläutern sowie z. T. kurz beschreiben, wie die Funktionen jeweils aufgerufen werden können.

6.2.1. Im- und Export

Sämtliche Im- und Export-Funktionen von Synchrovis sind unmittelbar über das *Datei*-Menü ansteuerbar. Diese beinhalten neben den Import-Operationen der Applikationsstatik und -dynamik aus KDM-, bzw. Kieker-Dateien (siehe Abschnitt 6.3.1 und 6.3.2) auch Möglichkeiten zur Speicherung und Wiederherstellung der aktuellen Visualisierung. Gerade die beiden letztgenannten Operationen sind dabei besonders nützlich. So lässt sich jede aktuell sichtbare Szene leicht zur späteren Weiterarbeit zwischenspeichern oder zum Zwecke der Kollaboration mit anderen Personen teilen. Es wird dabei nicht der aktuelle Bildausschnitt, sondern lediglich der aktuelle Visualisierungszustand inklusive des zugrunde liegenden Datensatzes gespeichert. Bei der Wahl einer Dateioperationen öffnet sich jeweils ein Fenster, mittels welchem man das auf dem Computer vorhandene Dateisystem durchsuchen und die zu ladende Datei auswählen, bzw. das bei der Speicherung zu verwendende Zielverzeichnis benennen kann.

6.2.2. Schnappschuss

Die Erstellung eines Schnappschusses stellt eine alternative Möglichkeit zur Extraktion der aktuellen Visualisierung aus unserer Applikation dar. So lässt sich der momentan auf dem Bildschirm sichtbare Ausschnitt in Form einer PNG-Datei auf der Festplatte abspeichern oder ohne Zwischenspeicherung direkt über einen auf dem System installierten, frei wählbaren Drucker ausgeben.

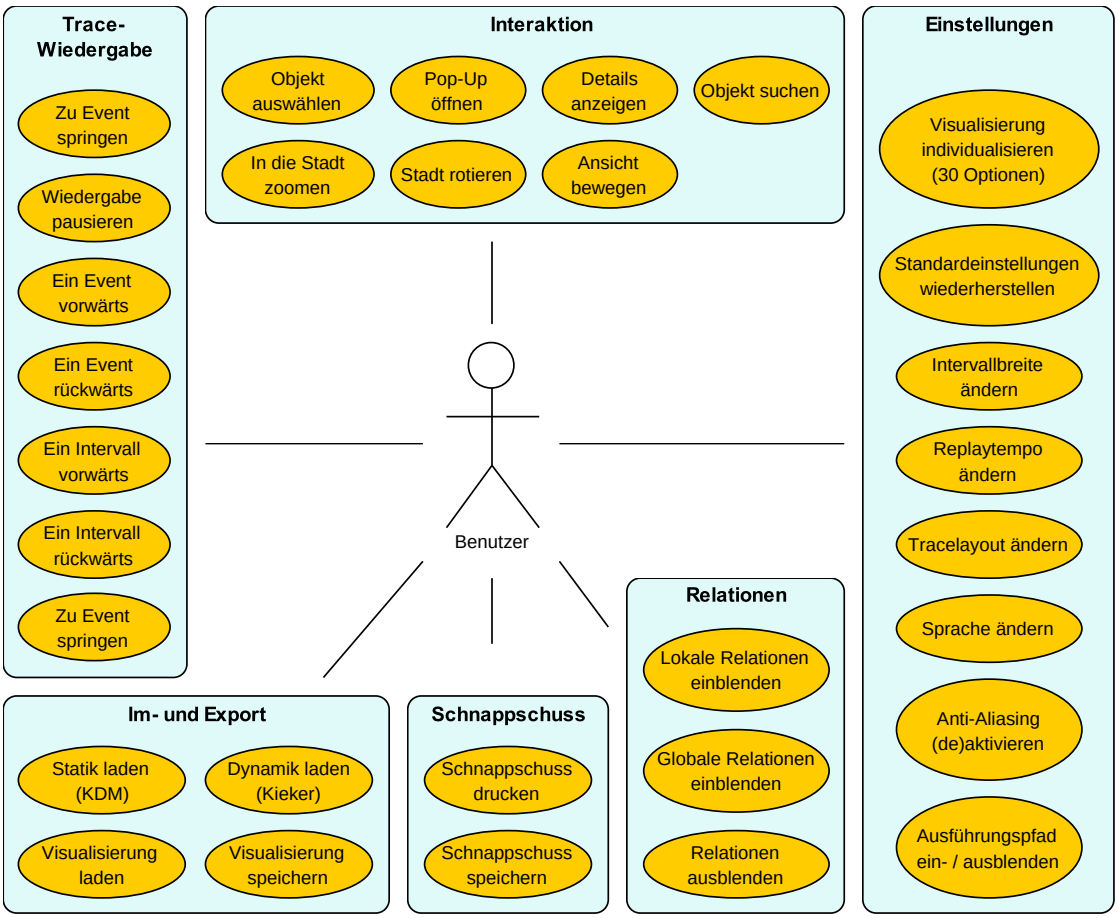


Abbildung 25: Die Funktionalität von Synchrovis im Überblick

6.2.3. Relationen

Das *Relationen*-Menü ermöglicht die Einblendung von Assoziations-, Ableitungs- und Interface-Implementierungsabhängigkeiten in die Visualisierung. Wurde ein Stockwerk eines regulären Gebäudes in der aktuellen Szene ausgewählt, lassen sich die eingehenden und ausgehenden Relationen der repräsentierten Klasse darstellen. Ebenso erlaubt Synchrovis die zeitgleiche Visualisierung sämtlicher in der betrachteten Applikation vorhandenen Beziehungen. Wird der momentane Visualisierungszustand verändert, werden diese automatisch ausgeblendet. Alternativ besitzt auch der entsprechende Menüpunkt diesen Effekt.

6.2.4. Einstellungen

Das Einstellungsmenü von Synchrovis haben wir in Abbildung 26 dargestellt. In diesem lassen sich mittels dreißig Optionen das optische Erscheinungsbild der Visualisierung weitestgehend bestimmen. Die Farbe, Größe und Abstand vieler visuellen Elemente können dabei frei definiert werden. Erstere Eigenschaft wird dabei über separates Farbeingabedialogfenster festgelegt. Letztere werden hingegen unmittelbar in das dafür vorgesehene Textfeld eingetragen, wobei ungültige Werte automatisch erkannt und zurückgewiesen werden.
Auch die Applikation an sich kann über eine Vielzahl an Einstellungsmöglichkeiten kon-

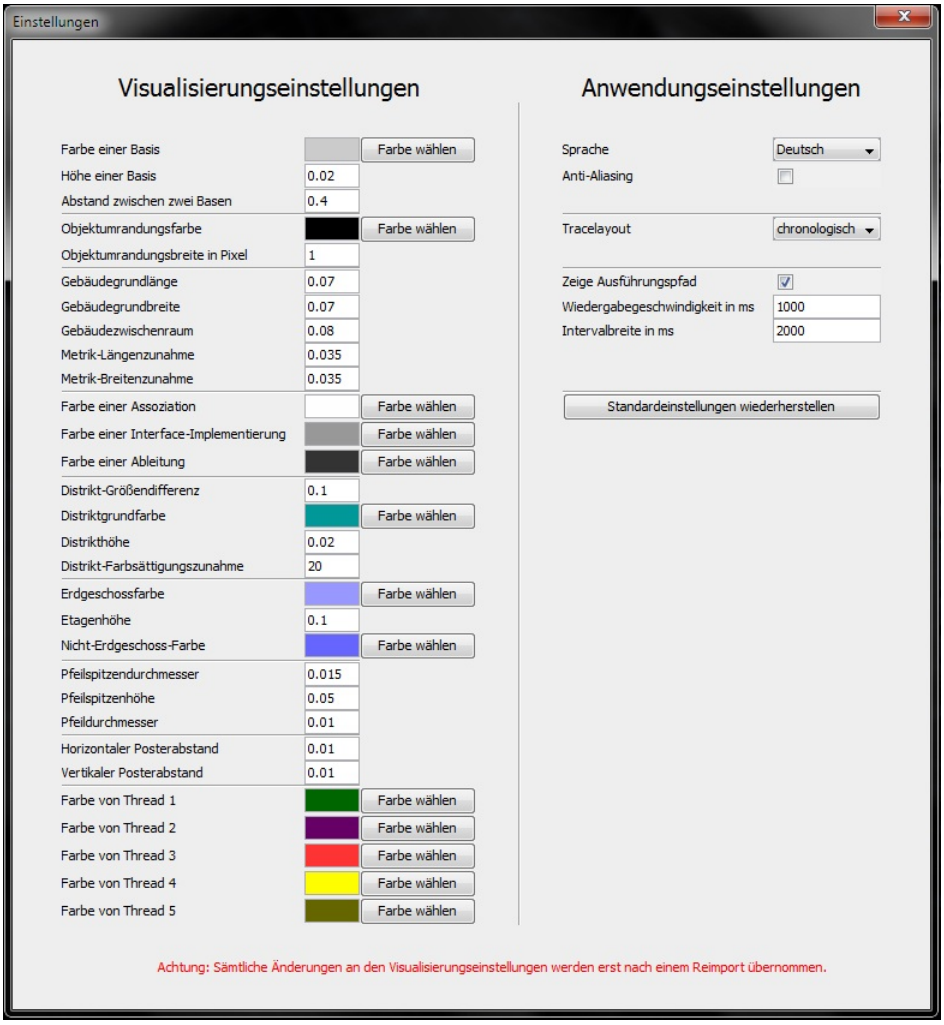


Abbildung 26: Das Einstellungsmenü in Synchrovis

figuriert werden. So lässt sich unter Anderem die Kantenglättung (engl.: *Anti-Aliasing*) aktivieren, die Sprache der Benutzeroberfläche zwischen Deutsch und Englisch umschalten sowie sämtliche Einstellungen auf ihre Standardwerte zurücksetzten. Besonderes Augenmerk möchten wir jedoch auf die Tracelayout- und Wiedergabeoptionen richten, da diese die visuelle Darbietung der Trace-Elemente und den Charakter unserer Visualisierung maßgeblich bestimmen. Diese erlauben dem Benutzer unter Anderem das Darstellungsformat des Applikationsablaufs in der Trace-Ansicht zwischen *chronologisch* und *hierarchisch* hin- und herzuschalten. Ist Ersteres ausgewählt, werden sämtliche Ereignisse gemäß ihres Zeitstempels absteigend geordnet und untereinander dargestellt. Ist hingegen Letzteres aktiviert, wird jedes Ereignis als Kindknoten desjenigen Methodenaufrufes angeordnet, der es initiiert hat. Diese Option kann alternativ auch bequem mittels der Trace-Layout-Leiste im Hauptfenster direkt verändert werden.
Die Visualisierung von nur einem Methodenaufruf pro Thread bedingt zwar eine übersichtliche Darstellung, ist jedoch häufig kontraproduktiv, wenn man Performance-Hotspots in einer betrachteten Applikation identifizieren möchte. Aus diesem Grund erlaubt Synchrovis auch die Darstellung sämtlicher Methodenaufrufe in einem Schaubild. Ist die Option *Zeige Ausführungspfad* aktiviert, werden bei der Wiedergabe des Programmablaufs keine Aufruf symbolisierenden Pfeile mehr aus der Visualisierung entfernt.
Die Trace-Wiedergabe lässt sich zudem unmittelbar beeinflussen. So ist es auch möglich die Geschwindigkeit der automatischen Wiedergabe zu bestimmen sowie die Intervallbreite bei der manuellen, zeitlichen Durchwanderung des Programmablaufs festzulegen.
Bei sämtlichen Änderungen an den Visualisierungseinstellungen ist jedoch zu beachten, dass diese erst nach einem erneuten Import der geladenen Kieker- bzw. KDM-Datei übernommen werden, da gegebenenfalls die gesamte Visualisierung neu aufgebaut werden muss. Veränderungen an der Applikationskonfiguration werden hingegen instantan übernommen.

6.2.5. Interaktionen

Synchrovis eröffnet dem Benutzer eine Vielzahl an Interaktionsmöglichkeiten mit der Visualisierung. So kann dieser mittels des Mausrads schrittweise in die Darstellung rein- oder rauszoomen, die Stadt durch Ziehen mit der rechten Maustaste im oder gegen den Uhrzeigersinn drehen oder mittels der Pfeil- bzw. WASD-Tasten den aktuellen Ausschnitt in alle vier Himmelsrichtungen verschieben. Wählt der Anwender ein Visualisierungselement mit der linken Maustaste aus, wird dieses zur leichteren Identifikation grünlich eingefärbt sowie die charakteristischen Werte des repräsentierten Datenobjekts in der Info-Tabelle eingeblendet. Alternativ zur Darstellung lokaler Relationen über die Menü-Leiste, können diese auch über ein Pop-Up eingeblendet werden, welches sich bei der Markierung eines regulären Gebäudeelements bei gedrückter Steuerungstaste öffnet.

Darüber hinaus kann der Anwender auch nach einem Paket oder einer Klasse in der Visualisierung suchen. Hierzu gibt er lediglich dessen vollständige Bezeichnung (d. h. im Fall von Klassen inklusive der vorangestellten Paketbezeichnung) in die dafür vorgesehene Eingabemaske ein und das gewünschte Objekt wird daraufhin farblich hervorgehoben.

6.2.6. Trace-Wiedergabe

Die Wiedergabe der eingelesenen Programmtraces wird überwiegend durch die Steuerungsleiste unmittelbar unterhalb der Zeichenfläche realisiert. Auf der einen Seite lässt sich die Dynamik einer betrachteten Anwendung automatisiert abspielen, wobei Synchrovis mit einer im Einstellungsfenster definierten Geschwindigkeit vorwärts durch den Programmtrace schreitet. Auf der anderen Seite ist aber auch eine manuelle Wiedergabe des Programmablaufs möglich. So kann sich der Benutzer entweder schritt- oder intervallweise durch die Applikationstraces bewegen, wobei bei Ersterem jeweils der Folge- oder vorangegangene Zustand relativ zum aktuellen Status dargestellt wird. Bei Letzterem hingegen springt der Benutzer einen im Einstellungsmenü festgelegtes Zeitintervall vor oder zurück im Trace. Diese manuellen Operationen sind jedoch nur dann verfügbar, wenn die automatische Trace-Wiedergabe zuvor pausiert wurde. Synchrovis unterstützt darüber hinaus ebenso ein direktes Springen zu einem beliebigen Ereignis. Hierzu wählt der Anwender einfach den entsprechenden Aufruf in der Trace-Ansicht mit der Maus aus.

6.3. Verwendete Technologien

Im Rahmen der Implementierung fanden eine Reihe von Technologien Anwendung, welche wir im Folgenden näher vorstellen möchten. Wir gehen dabei sowohl auf deren individuelle Charakteristika als auch auf ihren jeweiligen Verwendungszweck ein.

6.3.1. Kieker

Das *Kieker Monitoring & Analysis Framework* (kurz: *Kieker*) [34, 37] ist ein von der Arbeitsgruppe Software-Engineering der Christian-Albrechts-Universität zu Kiel entwickeltes, auf Java basiertes Werkzeug zur Performance-Überwachung und Laufzeitanalyse von Softwaresystemen. Diese Anwendung misst dabei die Ausführungszeiten einer Applikation auf Grundlage der sogenannten *Instrumentierungstechnik*. Möchte man die Laufzeit einer Methode bestimmen, wird bei diesem Verfahren unmittelbar vor und nach der Ausführung des Rumpfes die aktuelle Systemzeit ausgelesen. Die Differenz dieser beiden Werte ergibt nun Laufzeit der betrachteten Methode.
Kieker fügt den für die Laufzeitmessung erforderlichen Code automatisiert mittels Werkzeugen der aspektorientierten Programmierung [36] (hier: *AspectJ* [60]) in das Zielprogramm ein und gibt die Ausführungszeiten der betrachteten Methoden in Echtzeit aus. Die Laufzeitinformationen werden dabei in Form von maschinenlesbaren Dateien auf dem Rechner gespeichert, welche mittels Komponenten zur Analyse der beinhaltenden

Daten (z. T. grafisch) aufbereitet und ausgewertet werden können. Es sind dabei bereits verschiedene Ansätze implementiert, zu welchen neben der Darstellung als Aufrufgraph auch die Visualisierung in Form von Sequenzdiagrammen zählen.
Synchrovis liest die von Kieker erzeugten Laufzeitinformationen ein und rekonstruiert den gesamten dynamischen Programmablauf. Der resultierende Programmtrace ist dabei stets in der Trace-Ansicht sichtbar und wird anschließend wie in Kapitel 5 beschrieben visualisiert.

6.3.2. Knowledge Discovery Metamodell

Das *Knowledge Discovery Metamodell (KDM)* [1] ist eine Datenstruktur zur ganzheitlichen, Programmiersprachen und Plattform unabhängigen Beschreibung existierender Softwaresysteme inklusive deren Ausführungsumgebungen [57]. Die KDM definiert dabei vier aufeinander aufbauende Schichten, welche jeweils einen unterschiedlichen Aspekt der betrachteten Anwendung modellieren. So definiert die *Programmelemente*-Schicht die Programmstruktur einer Anwendung, d. h. dessen Quellcode und Dateistruktur. Auf der *Laufzeitressourcen*-Ebene sind dagegen Informationen zur Ausführungsumgebung, d. h. den Eigenschaften des Betriebssystems sowie der Datenhaltung, codiert. Hingegen modelliert die *Ressourcen*-Schicht die Interaktion einer Applikation mit der lokalen Ausführungsumgebung sowie die nach außen gerichteten Schnittstellen und die *Infrastruktur*-Schicht enthält vor Allem Pakete, deren Elemente von den übrigen drei Schichten als gemeinsame Basisentitäten weiterverwendet werden.
Zur Wiedergabe eines gegebenen Programmtraces benötigt Synchrovis stets ein KDM-Modell der darzustellenden Anwendung. Synchrovis liest aus dem *Code-Modell* der Programmelemente-Schicht die statischen Informationen der Applikation aus und generiert aus diesen die Distrikt- und Gebäudestruktur der Visualisierung sowie ordnet jeder mittels Kieker erfassten Methodenausführung der jeweiligen Methode zu. Aus einer gegebenen Java-Anwendung lässt sich ein solches KDM-Modell mit jedem beliebigen KDM-konformen Parser, z. B. *KADis*[6] oder *MoDisco*[7], generieren.

6.3.3. Eclipse Modeling Framework

Das *Eclipse Modeling Framework (EMF)* [4] ist ein quelloffenes Java-Framework basierend auf der Entwicklungsumgebung *Eclipse* [3] zur automatisierten Generierung von Java-Quellcode aus benutzerdefinierten Modellen. Diese können dabei sowohl als XML-Schema, UML-Diagramme oder auch als annotierte Java-Interfaces vorliegen. Auch die Erzeugung eines Modells per Hand mittels eines integrierten Editors ist möglich.

[6] `http://sourceforge.net/projects/kadis/`
[7] `http://eclipse.org/MoDisco/`

Die Modellierung findet dabei auf Basis von *Ecore* statt, welches als objektorientiertes Metamodell sowohl Standardelemente wie Pakete, Klassen, Attribute und Operationen als auch spezifischere Konstrukte wie Interfaces und Enumerationen umfasst. Ecore bietet dabei umfangreiche Code-Generierungsmöglichkeiten. So lassen sich neben dem eigentlichen Quellcode unter Anderem auch *JUnit*-Tests [10] zur Validierung der erzeugten Klassen oder Editor-Code zur Manipulation des Modells erstellen. Als zusätzliche Funktionalität unterstützt EMF zudem XML basierte Konvertierungsoperationen. So ermöglicht diese Technologie beispielsweise auch die bidirektionale Umwandlung von XML-Dateien in entsprechende Java-Klassen unter Zuhilfenahme der entsprechenden Ecore-Instanzen sowie das Generieren zugehöriger XML-Schema-Dateien.
Im Rahmen unser Implementierung haben wir EMF für die Konvertierung unseres Metamodells der Visualisierung (siehe Abschnitt 5.2) und der Datenhaltung (siehe Abschnitt 5.3) in Java-Quellcode verwendet. Da im Zuge der Entwicklung häufige Änderungen an diesen beiden Datenstrukturen zu erwarten waren, bot sich ein Modell getriebener Entwicklungsansatz für dessen Implementierung an. Etwaige Änderungen konnten auf diese Weise sehr leicht durch Anpassung der entsprechenden Ecore-Instanzen inklusive Code-Neugenerierung auf das Gesamtprojekt übertragen werden.

6.3.4. Java 3D

Java 3D [8] ist eine Java-Klassenbibliothek zur Erzeugung, Visualisierung und Manipulation dreidimensionaler graphischer Objekte. Java 3D kapselt dabei unmittelbar die auf dem Zielcomputer vorhandene *Direct3D*- [23], bzw. *OpenGL*-Schnittstelle [12] zu einem leicht verwendbaren, objektorientierten Benutzerinterface. Da dabei stets nur Betriebssystem spezifische Grafikbibliotheken sowie die auf die auf dem System installierte Grafikhardware verwendet werden, wird ein besonders performantes Rendering der visuellen Elemente ermöglicht. Ein direkter Zugriff auf Direct3D-, bzw. OpenGL-Funktionen ist dabei jedoch nicht möglich.
Java 3D ordnet die darzustellenden Objekte in Form eines Szenegraphen an, dessen Wurzelknoten der vollständigen Visualisierung entspricht. Auf diese Weise ist die logische Struktur des Dargestellten stets anhand der Baumstruktur ersichtlich und lässt sich nach Belieben manipulieren. Java 3D verwendet dabei ein rechtshändiges, zentriertes Koordinatensystem. Erstellt man ein neues Objekt im Punkt $(0, 0, 0)$ so wird dieses mittig im Ursprung platziert, wobei die x-Achse dessen Breite, die y-Achse dessen Höhe und die z-Achse dessen Tiefe beschreibt.
Synchrovis verwendet Java 3D in der aktuellen Version 1.5.2 zur Realisierung unseres in Kapitel 5 beschriebenen dreidimensionalen Visualisierungsansatzes. Die gesamte, auf dem Bildschirm sichtbare städtische Darstellung besteht aus den graphischen Elementen dieser Technologie.

6.4. Architektur

Nachfolgend präsentieren wir detailliert die Architektur von Synchrovis mittels Komponenten- und Paketdiagrammen. Mit Ausnahme der Beschreibung der Modell-zu-Modell-Transformationskette in Abschnitt 6.4.3 verzichten wir dabei auf die Darstellung von Klassen. Klassendiagramme lassen sich zum einen mit einer Vielzahl von Reverse-Engineering-Werkzeugen, z. B. *STAN4J*[8], aus dem bestehenden Quellcode rekonstruieren. Zum Anderen bietet aber auch bereits unsere detaillierte JavaDoc-Dokumentation eine ausreichende Beschreibung der Funktionalität und Aufgabe jeder Klasse unserer Anwendung. Zur leichteren Lesbarkeit können diese zudem in HTML konvertiert werden.

6.4.1. Komponentenübersicht

Abbildung 27 zeigt die Kernkomponenten von Synchrovis in Form eines Komponentendiagramms. Unsere Applikation realisiert dabei das *Modell-Präsentation-Steuerungs-Muster* (engl.: *Model View Controller Pattern*) [30]. So dient die *Logik*-Komponente der Steuerung, *Ansicht* der Erzeugung der Benutzeroberfläche sowie *Modells* der (temporären) Speicherung sämtlicher zugrunde liegender Informationen in Synchrovis. *Importeure* liest diese aus KDM-, respektive Kieker-Dateien ein und erzeugt so die Datenbasis unserer Visualisierung. Die Komponenten *Modelle* und *Importeure* werden wir in Abschnitt 6.4.3 jedoch noch etwas genauer betrachten.

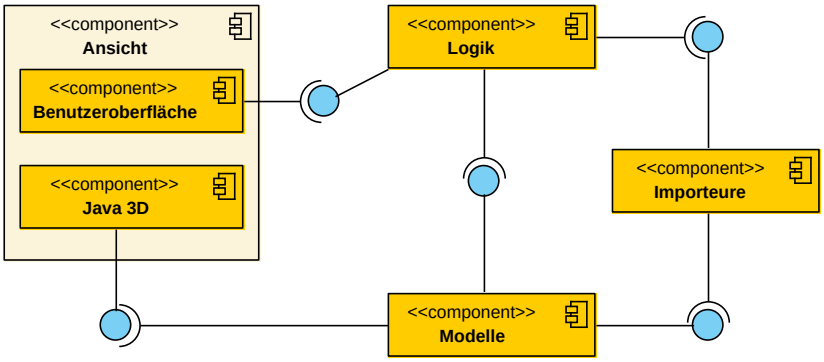


Abbildung 27: Die Komponenten von Synchrovis

[8] http://stan4j.com/

6.4.2. Paketübersicht

Abbildung 28 zeigt eine Übersicht über die Kernpakete unserer Anwendung inklusive ihrer gegenseitigen Abhängigkeiten. Aus Gründen der Übersichtlichkeit haben wir in dieser Darstellung lediglich die wesentlichen Nutzungsrelationen visualisiert. Für weitere Abhängigkeiten verweisen wir daher auf den Quellcode (siehe Anhang B).
Die Pakete *models*, *visual*, *logic* und *importer* stimmen funktional weitestgehend den in Abbildung 27 dargestellten Komponenten überein. So beinhaltet Ersteres die Implementierung unserer Metamodelle der Datenhaltung und der Metapher (Abschnitt 5.2, bzw. 5.3) sowie das Stadtmodell, welche den aktuellen Visualisierungszustand widerspiegelt. Einzelne Modelle, bzw. deren Importierungsklassen sind dabei mittels des Beobachter-Musters [30] (*observer*-Paket) verbunden. Eine Beschreibung dieses Transformationsprozesses findet in Abschnitt 6.4.3 statt.
Das *visual*-Paket beinhaltet die Implementierungen sämtliche auf dem Bildschirm sichtbarer Komponenten. So befindet sich in diesem beispielsweise der Quellcode der Menüleiste oder des Einstellungsfensters, aber auch der Java 3D-Objekte wie z. B. *Rechteck* oder *Pfeil*. Diese Elemente werden dabei von den Steuerungselementen in *logic* verwaltet und führen die von Benutzer gewünschte Aktion aus. Für jede funktional zusammenhängende Gruppe von Aufgaben ist dabei eine eigene Steuerungsklasse zuständig. So initiiert beispielsweise eine Klasse den KDM-, bzw. Kieker-Import mittels der im *importer*-Paket zur Verfügung gestellten Methoden. Eine andere Steuerungsinstanz übernimmt hingegen Einstellungsänderungen in die Konfigurationsdateien, welche über die Klassen des *configuration*-Pakets eingelesen und modifiziert werden können.
Die Pakete *main* und *utils* erfüllen keine eigenständige Aufgabe. So enthält Ersteres lediglich die ausführbare Hauptklasse, Letztere hingegen Hilfsklassen, u. A. zur Manipulation von Zeichenketten, welche von vielen Komponenten der Applikation gemeinsam genutzt werden.

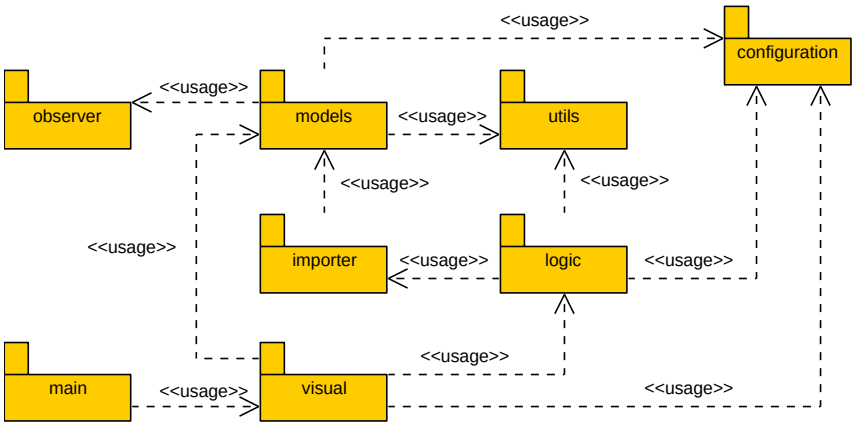


Abbildung 28: Die Paketstruktur von Synchrovis

6.4.3. Modell-zu-Modell-Transformationen

Synchrovis führt eine Reihe von Model-Transformationen durch bis die Darstellung auf dem Bildschirm erscheint. Abbildung 29 zeigt diesen Umwandlungsprozess in Form eines modifizierten Klassendiagramms. Die Struktur entspricht dabei dem von Panas et al. [56] vorgeschlagenem Model-View-Scene-Architekturmuster zur Separation der Daten- und Metapherelemente von der eigentlichen Visualisierung. So entspricht das *Data Model* und *Metaphor Model* unserem Metamodell der Datenhaltung, bzw. der Metapher. Wird eine KDM- oder Kieker-Datei vom Benutzer importiert, wird eine Instanz ersteren Modells mit den eingelesenen Daten befüllt und verfügt daraufhin über sämtliche Informationen, welche Synchrovis zur Erzeugung der Visualisierung benötigt.

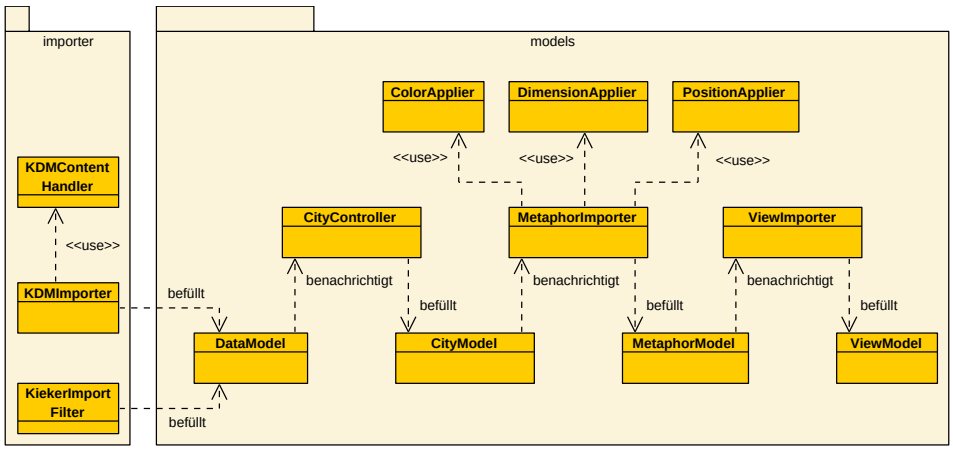


Abbildung 29: Der Modell-zu-Modell-Transformationsprozess in Synchrovis

Das *City Model* beinhaltet eine Teilmenge der Daten des Data Models und umfasst sämtliche Quellcode- und Programmtrace-Elemente, welche aktuell in der Visualisierung sichtbar sind. Wählt der Benutzer ein Methodenaufruf in der Trace-Ansicht aus oder initiiert er eine entsprechende Operation in der Steuerungsleiste, werden vom *City Controller* die korrespondierenden Datenelemente in das City Model eingefügt oder gelöscht. Ändern sich die statischen oder dynamischen Informationen des Data Models (z. B. im Zuge des Einlesens einer neuen KDM-Datei), wird der City Controller automatisch benachrichtigt.
Der *Metaphor Importer* dient der Befüllung des Metaphor Models mit Informationen. Wird dieser von dem City Model über eine Änderung des Datenbestandes informiert, erzeugt dieser die entsprechenden Metapher-Elemente und fügt sie zum Metaphor Model hinzu, bzw. entfernt diese. Für erstere Operation nutzt der Importeur hierbei den *Color*, *Dimension* und *Position Applier*. Ersterer bestimmt die Färbung, Zweiterer die Maße und Letzterer die Koordinaten des zu erschaffenden Objekts.

Das *View Model* entspricht dem Java-3D-Szenegraphen, welcher aktuell auf dem Bildschirm sichtbar ist. Es wird vom *View Importer* mit Daten bestückt, welches es vom Metaphor Model erhalten hat. Wird ein Element in das Metaphor Model eingefügt, erzeugt der Importeur automatisch das entsprechendes Java-3D-Objekt. So wird beispielsweise ein Rechteck für einen Distrikt oder ein Pfeil für einen regulären Methodenaufruf generiert und auf dem Bildschirm dargestellt.
Sämtliche Modell-Instanzen sind dabei mittels des Beobachter-Musters mit dem Importeur der jeweils nachfolgenden Instanz verbunden, d. h. jegliche *benachrichtigt*-Operationen basieren auf dieser Form des Nachrichtenaustausches.
Diese Konzipierung des Modell-zu-Modell-Transformationsprozesses bringt dabei mehrere Vorteile mit sich. So erzeugt die Transformationsstruktur zum einen übersichtlichen Informationsfluss innerhalb des Programms, ermöglicht aber auch den leichten Austausch einzelner Module ohne angrenzende Klassen verändern zu müssen. Möchte man beispielsweise statt Java 3D ein anderes Grafikframework verwenden, reicht es aus, lediglich das *View Model* inklusive der Implementierung der visuellen Objekte (d. h. zum Beispiel Rechteck und Pfeil) auszutauschen. Das Beobachter-Muster sorgt dabei zusätzlich für eine geringe Kopplung zwischen den Modellinstanzen und definiert eine implementierungsunabhängige Schnittstelle für die Integration weiterer Transformationsschritte. Ist beispielsweise die Erweiterung des Funktionsumfangs von Synchrovis um eine Filterfunktion geplant, empfiehlt es sich, einen zusätzliche Filterklasse zwischen dem City Model und Metaphor Importer zwischenzuschalten.

7. Evaluation von Synchrovis

Nachdem wir nun die Entwicklung unserer prototypischen Implementierung abgeschlossen haben, gilt es nun Synchrovis hinsichtlich seiner qualitativen Eigenschaften zu beurteilen. Wir bewerten unsere Applikation zunächst anhand unserer Evaluationskriterien aus Abschnitt 4.1 und holen anschließend im Rahmen einer Expertenbefragung die Meinung von Fachkundigen ein. Da eine Implementierung typischerweise stets Raum für Verbesserungen lässt, rundet eine Erörterung möglicher, zukünftiger Verbesserungen und Erweiterungen dieses Kapitel ab.

7.1. Evaluation nach Bewertungskriterien

Ziel der Evaluation gemäß unserer Bewertungskriterien ist es, eine Vergleichbarkeit zwischen Synchrovis und den in Kapitel 3 vorgestellten Visualisierungsansätze zu schaffen. Auf diese Weise lassen sich sowohl dessen Stärken und Schwächen im namhaften Vergleich identifizieren als auch Bereiche mit Verbesserungspotential aufzeigen.

Metapher Synchrovis basiert auf der städtischen Metapher und visualisiert Klassen in Form von identisch gefärbten Gebäuden unterschiedlicher Etagenzahl, Länge und Breite. Pakete werden durch übereinanderliegende Rechtecke gleicher Grundfarbe, aber abgestufter Farbsättigung dargestellt. Zwischen den Stockwerken, bzw. Gebäudedächern können dabei unterschiedlich gefärbte und gestaltete, gerichtete Kanten verlaufen, von welchen Erstere einen Methodenaufruf des codierten Threads, Letztere hingegen eine Abhängigkeitsbeziehung zwischen Quell- zur Zielklasse symbolisieren. An den Außenseiten der Verwaltungsetagen können zudem verschiedenfarbige Plakate angebracht sein, welche den aktuellen Zustand des von dem Stockwerk repräsentierten Objekts, respektive Threads angeben. Deren Gebäude sind dabei durch ein Symbol über dem Dach eindeutig gekennzeichnet und geben auf eingängige und intuitive Weise dessen Repräsentationsfunktion wieder. Die verwendete Metapher ist damit relativ ausdrucksstark und würde beispielsweise durch die Verwendung einer unterschiedlichen Etagen- oder Distrikthöhe weiteres Potential für Erweiterungen bieten. Dennoch würde dies den vergleichsweise geringen Realismus der Darstellung weiter senken und sollte daher lediglich mit Bedacht gewählt werden. Die Verwendung wirklichkeitsnaher Gebäude- und Straßentexturen könnte jedoch den Realismus der Darstellung und damit dessen Gesamtqualität deutlich erhöhen. Durch die Visualisierung einer Klasseninstanz in Form eines Stockwerks und der damit verbundenen, z. T. stark differierenden Gebäudehöhe entsteht häufig eine charakteristische Stadtstruktur. Als markante Landmarken fallen zudem der separate Verwaltungsbezirk rechts der regulären Grundfläche sowie die Symbole über dessen Gebäude sofort ins Auge und können von Betrachter als Orientierungs- oder Wegpunkte innerhalb der Stadt verwendet werden. Somit ist der Darstellung eine hohe Diversität zuzusprechen. Dennoch wird das Wiederfinden gesuchter Objekte bei der Wiedergabe von Programmtraces stark

erschwert, da die Anzahl der Etagen und damit die Gebäudehöhe über die Zeit variiert.

Statik Synchrovis codiert Pakete in Form von Distrikten und Klassen in Form von Gebäuden, deren Länge und Breite die Anzahl der Methoden, bzw. Attribute wahlweise widerspiegeln. Relationen lassen sich optional in der Darstellung einblenden und werden durch unterschiedlich gefärbte, gerichtete Kanten zwischen den Dächern nicht notwendigerweise verschiedener Gebäude visualisiert. Eine weiße Kante repräsentiert dabei eine Assoziationsbeziehung, eine graue eine Interface-Implementierung. Eine schwarze Kante symbolisiert hingegen ein Ableitungsverhältnis. Somit erfüllt Synchrovis sämtliche Kriterien aus dem Bereich der Statikdarstellung.

Dynamik Jede nicht-erdgeschössige Etage eines Gebäudes der regulären Grundfläche repräsentiert eine Klasseninstanz. Methodenaufrufe werden dabei als Kanten zwischen diesen dargestellt, deren Färbung die Zugehörigkeit zu einem der visualisierten Threadausführungen angibt. Context-Switches lassen sich jedoch lediglich während der Trace-Wiedergabe anhand der chronologischen Reihenfolge der Kanteneinblendungen erkennen. Wird beispielsweise eine rote Methodenaufruf repräsentierende Kante durch eine blaue abgelöst, so repräsentiert dies einen Context-Switch vom roten zum blauen Thread. Threadzustände lassen sich hingegen jeder Zeit anhand der unterschiedlichen Farbgebung und Vorhandensein der Poster und Etagen der Verwaltungsgebäude sowie der auf diese gerichteten Kanten ablesen. Es werden dabei die klassischen Zustände *running*, *waiting*, *waiting on a monitor* und *terminated* im Rahmen der zuvor beschriebene Synchronisationsmechanismen unterstützt. Eine Visualisierung von Methodenausführungszeiten findet jedoch nicht statt und lässt sich ebenso nicht ohne Weiteres in die Darstellung integrieren.

Synchronisationspunkte Synchrovis repräsentiert Semaphor- und Monitor-Instanzen in Form der Stockwerke des korrespondierenden Semaphor-/Monitor-Gebäudes. Aufrufe ihrer Methoden werden dabei durch eine auf das entsprechende Stockwerk gerichtete Kante in der Farbe des ausführenden Threads symbolisiert. Auf vergleichbare Weise werden Wait-/Notify-Operationen sowie Thread-Joins visualisiert. Kritische Sektionen werden hierbei nicht gesondert markiert, lassen sich jedoch anhand der Aufrufsequenz der Semaphor- und Monitor-Methoden in der Trace-Wiedergabe erkennen. Eine spezifische Hervorhebung dieser Sektionen, beispielsweise durch eine spezielle Kantenform, wäre denkbar. Tupelräume bleiben in Synchrovis gänzlich unvisualisiert und lassen sich jedoch nur schwer in das bisherige Darstellungskonzept integrieren.

Funktionalität Synchrovis erfüllt insgesamt sechs unser acht Funktionalitätskriterien. So repräsentiert bereits die einzig implementierte Ansicht eine *Übersichtsansicht*. Mittels einer *Zoom*-Funktion kann sich der Anwender frei in der erzeugten Darstellung bewegen (*Navigation*) sowie per Mausanwahl nähere Informationen zu beinahe jedem optischen Element abrufen (*Details-On-Demand*). Relationen können dabei über ein Kontext-, bzw. das Hauptmenü optional eingeblendet werden (*Verknüpfung*) sowie die gesamte Visualisierung zur späteren Weiterarbeit gespeichert werden (*Extraktion*). Lediglich die Anwenderbedürfnisse nach einer *Filterfunktion* und einer *Historie* befriedigt Synchrovis nicht.

Ästhetik Die von Synchrovis erzeugte Visualisierung wirkt aufgrund der Verwendung der städtischen Metapher sehr kompakt. Auch bei der Darstellung von größeren Applikationen kann man diese noch als annäherungsweise quadratisch und damit umfangsminimal bezeichnen. Eine leichte Überblickbarkeit der gesamten Abbildung ist die positive Folge.
Bei der Visualisierung von Methodenaufrufen werden bei deaktivierter *Zeige-Ausführungspfad*-Option stets nur eine begrenzte Anzahl gleichzeitig visualisiert. Somit bleibt die Zahl an Kantenüberschneidungen gering. Da jedoch bislang noch keine Kollisionserkennung implementiert ist, könnten Kanten bestehende Etagen auf ihrem Weg durchkreuzen, d. h. zu grafischen Überlappungen führen, und damit die Identifizierbarkeit der beteiligten Objekte vermindern. Blendet man sämtliche Relationen einer betrachteten Applikation gleichzeitig ein oder zeigt den gesamten Programmtrace auf ein mal (Option: *Zeige Ausführungspfad*), sind Kantenüberschneidungen und grafische Überlappungen keine Ausnahme mehr, sodass sich einzelne Kanten lediglich durch Auswahl mit der Maus und damit farbliche Hervorhebung in der Darstellung auseinanderhalten lassen.

Fazit Tabelle 4 zeigt eine zusammenfassende Übersicht über die Evaluationsergebnisse unserer Applikation. Synchrovis bietet dabei nur wenig Anlass für Tadel, was sicherlich auch damit zusammenhängt, dass wir aus den Mängeln der vorangegangenen Applikationen gelernt haben. So besitzt sie im Bereich der verwendeten Metapher sehr gute Qualitäten und erfüllt beinahe sämtliche Vollständigkeitskriterien der Statik, Dynamik und Synchronisationspunkte. Die Implementierung von sechs der acht funktionalen Anforderungen ist ebenso positiv zu bewerten. Lediglich die Häufigkeit von grafischen Überlappungen müsste dringend verringert werden.
Auch im unmittelbaren Vergleich mit den Applikationen und Ansätzen aus Abschnitt 3 schneidet unsere Anwendung sehr gut ab. Dessen qualitativen Eigenschaften lassen Synchrovis sogar zum neuer Sieger unseres Evaluationsverfahrens werden.

(Unter-)Kategorie	Kriterium	Ausprägung
Metapher	Bezeichnung	städtisch
	Dimensionalität	3D
	Realismus	+
	Diversität	++
	Ausdrucksstärke	++
Statik	Pakete	Repräsentation
	Klassen	Distrikte
	Assoziationen	Linien
	Ableitungen	Linien
	Attribute	Gebäudebreite od. -Länge
	Methoden	Gebäudebreite od. -Länge
Dynamik	Klasseninstanzen	Etagen
	Methodenaufrufe	Linien zw. Etagen
	Ausführungszeit	–
	Context-Switches	Abfolge d. Methodenaufrufe
	Threadzustände	Farbe der Etagen u. Plakate
Synchronisationspunkte	Semaphore	Etagen
	Monitore	Etagen
	Thread-Joins	Kanten zw. Etagen
	Tupelräume	–
	Wait/Notify	Kanten z. Etagen
	Kritische Sektionen	(Kantenform)
Funktionalität	Übersicht	Ja
	Zoom	Ja
	Filter	Nein
	Details-on-Demand	Ja
	Verknüpfung	Ja
	Historie	Nein
	Extraktion	Ja
	Wiedergabe	Ja
Ästhetik	Kompaktheit	++
	Kantenüberschneidungen	+
	Grafische Überlappungen	+/–

Tabelle 4: Evaluation von Synchrovis anhand unserer Bewertungskriterien

7.2. Expertenbefragung

Eine Expertenbefragung beschreibt eine Methode des Erkenntnisgewinns, welche häufig dann eingesetzt wird, wenn das eigene Wissen oder Erfahrung nicht ausreicht, um einen betrachteten Sachverhalt umfassend beurteilen zu können. Gerade in einem Gespräch mit Fachkundigen können häufig neue Anregungen für Verbesserungen entstehen, welche in einer internen Evaluation bisher unidentifiziert blieben. Wir erhoffen uns daher von der Expertenbefragung vor Allem eine qualitative Bewertung unseres Visualisierungsansatzes. Aber auch auf eine Rückmeldung zur Benutzerfreundlichkeit und Funktionalität unserer Implementierung legen wir großen Wert, da gerade diese Faktoren oftmals darüber entscheiden, für welche Visualisierungsapplikation sich ein Anwender entscheidet.

7.2.1. Durchführung

Im Rahmen unserer Expertenbefragung wurden insgesamt vier Doktoranden der Arbeitsgruppe Software-Engineering der Christian-Albrechts-Universität zu Kiel zur Qualität von Synchrovis und der von dieser Applikation erzeugten Visualisierung befragt. Diese Personen wurden dabei so ausgewählt, dass sie aufgrund ihres Forschungsschwerpunkts oder ihrer bisherigen Tätigkeit ausreichend Erfahrung mitbrachten, um Synchrovis kompetent beurteilen zu können. Unter diesen war mit C. Wulf ebenso der Entwickler des in Abschnitt 3.2.2 beschriebenen Softwarewerkzeugs DyVis [65], auf dessen Visualisierung unser Darstellungsansatz aufbaut.
Jedem Teilnehmer wurde Synchrovis im Einzelgespräch vorgestellt sowie anhand eines Fragebogens, welcher dieser Arbeit im Anhang A beigefügt ist, befragt. Die Dauer jedes Interviews betrug jeweils zwischen 20 und 45 Minuten. Die Darstellung der Statik wurde dabei anhand der Visualisierung des *JPetStores*[9] durchgeführt. Zur Beurteilung der Dynamik und der Synchronisationspunkte wurde hingegen ein Demonstrationsprogramm, bestehend aus insgesamt zwei Klassen, drei Paketen und dreizehn Methodenaufrufen in zwei Threads, verwendet, welches im Zusammenhang mit dem aktuellen Implementierungsfortschritt in Kieker den gesamten Darstellungsumfang von Synchrovis abdeckte.
Da im Rahmen dieser Befragung auch um eine Bewertung der Benutzerfreundlichkeit unserer Anwendung gebetet wurde, erhielt jeder Teilnehmer während der Befragung die Möglichkeit, Synchrovis probeweise selbstständig zu bedienen.

[9] `http://sourceforge.net/projects/ibatisjpetstore/`

7.2.2. Ergebnisse

In den nachfolgenden Abschnitten präsentieren wir die Ergebnisse unserer Expertenbefragung. Aufgrund der häufig sehr speziellen und kleinschrittigen Frageweise möchten wir dabei nicht die Antworten auf jede Frage einzeln vorstellen sondern ziehen stattdessen je Fragekategorie ein zusammenfassendes Resümee der gewonnenen Erkenntnisse.

Darstellung der Statik Die Darstellung der Statik wurde von sämtlichen Befragten als positiv empfunden. Insbesondere würde dabei die Übersichtlichkeit der Darstellung positiv hervorgehoben. Zwei der vier Befragten vermissten in der Darstellung jedoch die Visualisierung statischer Methodenaufrufbeziehungen. Diese könnten beispielsweise in Form weiterer Dächer verbindender Kanten in bisher noch nicht verwendeten Farben in der Visualisierung dargestellt werden und ähnlich wie Relationen optional eingeblendet werden.
Zwei Teilnehmer wünschten sich zudem Orientierungshilfen innerhalb der Darstellung. So schlug ein Befragter vor, auf den überstehenden, sichtbaren Rand jedes Distrikts die Bezeichnung des zugehörigen Pakets zu notieren. Der andere Teilnehmer sprach sich hingegen für die Verwendung von Tooltips aus, welche beim Kontakt eines graphischen Elements mit der Maus jeweils den Namen des dargestellten Pakets oder der Klasse zeigen. Den Vorschlag der Distriktbeschriftungen haben wir dabei bereits umgesetzt.[10] Im direkten Vergleich mit einem UML-Klassendiagramm eines Ausschnitts der betrachteten Applikation sahen sämtliche Befragten die Vorteile der UML in der sofortigen Erkennung der Bezeichnung einer jeweiligen Entität.

Darstellung der Dynamik Die Visualisierung der Dynamik erachteten sämtliche Teilnehmer als intuitiv und leicht verständlich. Auch im unmittelbaren Vergleich unserer Darstellung mit einem UML-Sequenzdiagramm zeigten sich die Vorzüge von Synchrovis. So bezeichnete beispielsweise ein Teilnehmer das UML-Diagramm als *überladen* und empfand demgegenüber die sequenzielle Darbietung des Programmtraces in Synchrovis als übersichtlicher und leichter nachvollziehbar.
Zwei der Befragten bemängelten jedoch die fehlende Skalierbarkeit der Synchrovis-Darstellung bei der Visualisierung des gesamten Ausführungspfades. So fiele es ihnen schwer, in dieser Darstellung den Überblick zu bewahren. Als Gegenmaßnahme schlugen sie vor, mehrere Aufruf repräsentierende Kanten zwischen zwei gleichen Stockwerken zu einer einzigen Kante zu akkumulieren und dessen Multiplizität durch eine beistehende Zahl zu notieren.

[10] In der von den Experten evaluierten Version waren die Distrikte noch unbeschriftet, keine Suchfunktion implementiert und die Markierung eines Objekts wurde noch mit der rechten, bzw. die Drehung der Stadt mit der linken Maustaste ausgelöst.

Ein weiterer Teilnehmer regte zudem an, in diesem Darstellungsmodi sämtliche nicht mehr aktiven Methodenaufrufe in einer einheitlichen, neutralen Farbe, z. B. Grau, zu färben, da andernfalls der jeweils neu hinzugekommene Methodenaufrufpfeil bei besonders vielen Kanten nur schwer zu identifizieren sei. Diese Maßnahme würde jedoch unserer Intention dieser Darstellungsform entgegenwirken, da man auf diese Weise nicht mehr zwischen den Ausführungspfaden der einzelnen Threads unterscheiden könnte.

Darstellung der Synchronisationspunkte Die Teilnehmer beurteilten die Darstellung der Synchronisationspunkte als positiv. Dabei wurde besonders die Übersichtlichkeit der Belegungsstruktur der Semaphore, bzw. Monitore hervorgehoben. Ein Teilnehmer empfand jedoch die konzeptionelle Trennung der Monitor-Instanzen zu ihren (gegebenenfalls real existierenden) Klasseninstanzen als vergleichsweise unintuitiv und tendierte zur Verwendung von Postern auch bei regulären Etagen, falls das repräsentierte Objekt ebenfalls als Semaphor, bzw. Monitor angesteuert wird. Dies hätte jedoch aus unserer Sicht zur negativen Folge, dass Deadlock-Szenarien. bzw. die gesamte Monitor-Belegungsstruktur nicht mehr so leicht zu erkennen wären, da die beteiligten Monitor-Instanzen gegebenenfalls über die gesamte Stadt verteilt wären.
Ein weiterer Teilnehmer bemängelte darüber hinaus die Verwendung eines Pfeils zur Kennzeichnung des Thread-Gebäudes, da er dieses Symbol nicht auf Anhieb mit einem Thread in Verbindung bringe. Stattdessen schlug er den Buchstaben T als Akronym für *Thread* als einleuchtenderes Identifikationssymbol vor. Dagegen lobte er ausdrücklich die Markierung des Semaphor-/Monitor-Gebäudes mittels eines Vorhängeschloss-Symbols. Durch dieses würden seiner Ansicht nach selbst Personen ohne Vorherige Einführung in Synchrovis unmittelbar erkennen, welches Konstrukt die Stockwerke dieses Gebäudes repräsentieren.

Synchrovis und dessen Funktionalität Auch zur optischen und räumlichen Gestaltung der graphischen Benutzeroberfläche und der zur Verfügung gestellten Funktionen von Synchrovis erhielten wir positive Reaktionen. Ein Befragter wünschte sich über den bisher implementierten Funktionsumfang hinaus zusätzlich eine Suchfunktion zum schnellen Auffinden gesuchter Pakete und Klassen, welche bereits implementiert wurde, und ein weiterer Teilnehmer eine farbliche Legende, welche jedem Thread seiner Farbe in der Visualisierung zuordnet.[10]
Ebenso die Trace-Ansicht würde von sämtlichen Experten als übersichtlich empfunden. Ein Teilnehmer regte jedoch die Verwendung unterschiedlicher Farben oder einer Symbolik zur Markierung der unterschiedlichen Methodenarten (z. B. statische Methode oder Konstruktor) an.

Zwei Teilnehmer kritisierten hingegen die vergleichsweise unintuitive Bedienung der Markierungs- und Rotationsfunktion. Sie sprachen sich dabei für die Verwendung der linken Maustaste zur Selektion eines graphischen Elements und der rechten Maustaste zur die Ansteuerung der Rotation der Gesamtansicht aus. Dieser Vorschlag wurde in der aktuellen Version von Synchrovis bereits umgesetzt.[10]

Fazit Die Rückmeldungen der vier befragten Experten zu Synchrovis und dessen Visualisierung fielen ausgesprochen positiv aus. So fand unser Gesamtvisualisierungskonzept bei sämtlichen Teilnehmern Anklang. Der einzig fundamentale Kritikpunkt stellte aus unserer Sicht dabei die fehlenden initialen Orientierungsmöglichkeiten innerhalb der Stadt dar. Mit der Implementierung einer Suchfunktion für Klassen und Paketen sowie der Beschriftung jedes Distrikts mit der Bezeichnung des repräsentierten Pakets haben wir diesem Makel bereits entgegengewirkt und hoffen, dass dem Anwender die Zuordnung der einzelnen Distrikte und Gebäude zu den repräsentierten statischen Elementen nun leichter fällt. Durch die Implementierung entsprechender Tooltips könnte dieses Problem noch weitergehender gelöst werden und damit die Attraktivität unserer Darstellung weiter gesteigert werden.

7.3. Mögliche Erweiterungen

Neben den in Abschnitt 7.2 beschriebenen möglichen Erweiterungen des Funktionsumfangs (Verwendung von Tooltips, Implementierung einer farblichen Legende für Threads) existieren noch weitere potentielle Erweiterungen und Verbesserungen, welche die Aussagekraft der Darstellung und Benutzerfreundlichkeit unserer Applikation deutlich erhöhen könnten, welche wir nachfolgend vorstellen möchten.

7.3.1. Erhöhung des Realismus

Synchrovis basiert auf der städtischen Metapher und erzeugt damit bereits eine sehr realitätsnahe Visualisierung. Dennoch haben wir mit unserer Implementierung das gesamte Potential an Realismus noch nicht einmal ansatzweise ausgeschöpft. So könnte man außerhalb der Stadt beispielsweise durch die Darstellung einer wirklichkeitsnahen urbanen Umgebung (z. B. mittels Grünflächen und Wälder), eines klar erkennbaren Horizonts sowie des Himmels die Eingängigkeit der Visualisierung deutlich steigern. Auch innerhalb der Stadt lassen sich einige Verbesserungen umsetzen. So könnte man beispielsweise realistische Gebäudetexturen auftragen, um die Assoziation der Visualisierungsform mit dem Bildnis einer (Groß-)Stadt weiter zu verstärken, oder die Zwischenräume zwischen den Häusern durch die Andeutung eines Straßennetzes wirklichkeitsgetreuer darstellen. Durch letztere Erweiterung wird ebenso die Ersetzung der Relationen repräsentierenden Kanten durch Fahrzeuge ermöglicht, welche auf diesem Straßennetz zwischen Start- und Zielgebäude verkehren könnten.

7.3.2. Integration weiterer Synchronisationsmechanismen

Synchrovis wurde mit dem Ziel entwickelt, möglichst viele Synchronisationsmechanismen gleichzeitig abbilden zu können. Mit der Visualisierung von Monitoren, (binären) Semaphoren sowie den Wait-/Notify- und Join-Operationen ist bereits die Darstellung der gängigsten, in Java implementierten Synchronisationskonzepte aus dem Bereich der nebenläufigen Anwendungen möglich, sodass in diesem Kontext lediglich Zählsemaphore (siehe Abschnitt 2.2.1) sowie das Linda-Modell (Abschnitt 2.2.5) in dessen Darstellungsumfang noch fehlen.
Erweitert man den Fokus auf verteilte Softwaresysteme, erkennt man jedoch schnell die ersten Schwächen von Synchrovis. Vor Allem mangelt es hier an der Visualisierbarkeit der synchronen Interprozess-Kommunikation (insbesondere des Rendezvous-Konzepts). Hierfür müssten jedoch zunächst verteilte Multiprozess-Systeme geeignet dargestellt werden können. Dies ließe sich beispielsweise durch die Abbildung mehrerer Städte, welche jeweils einen einzigen Prozess repräsentieren, realisieren. Straßen bzw. Luftverkehrswege könnten diese verbinden und jeweils einen Methodenfernaufruf (z. B. durch *Remote Method Invocation* [6]), einen allgemeinen Nachrichtenaustausch (z. B. durch *Transmission Control Protocol* [22]) oder sonstige gegenseitige Abhängigkeiten symbolisieren.

7.3.3. Integration weiterer Applikationseigenschaften

Neben der Darstellung konkreter Methodenaufrufe und deren Aufrufpfade sollte unser Darstellungsansatz auch gemessene Performancedaten zukünftig geeignet visualisieren können. Die rein textuelle Beschreibung von Ausführungszeiten ist allein nicht ausreichend, um Performance-Hotspots und damit eventuelle Schwächen in der Architektur oder Implementierung der betrachteten Applikation erkennen zu können. Besonders frequentiert aufgerufene oder Laufzeit intensive Klassen könnte man beispielsweise mittels eines brennenden Gebäudes visualisieren. Auf der anderen Seite könnten längere Zeit nicht mehr verwendete Klassen oder Instanzen durch zunehmend verfallende Gebäude, bzw. Stockwerke symbolisiert werden.
Auch die aktuelle Prozessorlast ließe sich auf einfache und zugleich passende Weise in die Applikation beispielsweise in Form einer Wetterdarstellung integrieren. So könnte ein Gewittersturm eine Periode mit hoher Last repräsentieren, wohingegen ein wolkenfreier Himmel, bzw. Sonnenschein eine geringe Prozessorauslastung symbolisiert.
In Synchrovis existiert noch kein Konzept zur Darstellung von Etageninterna. Hier bietet sich die Abbildung der Methoden der Klasse an und könnten nach dem Vorbild von EvoSpaces [13, 14, 15, 27] (siehe Abschnitt 3.2.1) durch Strichmännchen oder sonstigen passenden Assoziationen innerhalb der Stockwerke repräsentiert werden.In unserer

Visualisierung ist die dynamische Elimination einzelner Etagen darüber hinaus bislang noch nicht vorgesehen. Jedoch ist dies gerade dann sinnvoll, wenn einzelne Objekte nicht mehr in Verwendung sind und bereits durch die automatische Speicherverwaltung aus dem Speicher entfernt wurden. Erfolgt diese Bereinigungsaktion durch den Java Garbage-Collector, wird automatisch die *finalize()*-Methode auf dem zu entfernenden Objekt aufgerufen [5]. So könnte man leicht die aktuelle Implementierung abändern, um auch (näherungsweise) den aktuellen Speicherzustand der Java Virtual Machine abbilden zu können. Das verwendete Kieker-Framework müsste hierfür jedoch modifiziert werden, um auch solche Methodenaufrufe detektieren und aufzeichnen zu können.

7.3.4. Erhöhung der Programmfunktionalität

Auch im Bereich der Programmfunktionalität bietet Synchrovis noch einiges an Verbesserungspotential. Zwar sind in unserer Anwendung bereits einige Metriken als Maß für die Größe von Klassen implementiert, jedoch lassen sich diese noch nicht dynamisch vom Benutzer, d. h. beispielsweise im Einstellungsfenster, auswählen oder anpassen. Die Wahl ist bislang lediglich durch Modifikation des Synchrovis-Quellcodes möglich.
Ebenso besitzt der Anwender bislang noch keine Möglichkeit zur Herausfilterung spezieller Thread-Ausführungen oder Methodenarten. Doch gerade durch diese Funktionalität könnte die Benutzerfreundlichkeit von Synchrovis deutlich steigern, da der Anwender auf diese Weise für ihn unwichtige Informationen ausblenden kann.
Ist in einer eingelesenen KDM-Datei zudem der aktuelle Speicherort einer jeden Java-Datei inkludiert, könnte man auf Anfrage zudem auch den Quellcode einer betrachteten Klasse öffnen und dem Benutzer eine bessere Nachvollziehbarkeit der Darstellung ermöglichen.

8. Themenverwandte Arbeiten

In Kapitel 3 haben wir bereits eine Reihe themenverwandter Arbeiten vorgestellt. So stellt die *Unified Modelling Language (UML)* [54] (siehe Abschnitt 3.1.1) heutzutage den De-facto-Standard im Bereich der Modellierung von Softwaresystemen dar und basiert auf einem graphenbasierten Visualisierungsansatz.
Alam und Dugerdil [13, 14, 15, 27] präsentierten mit *EvoSpaces* (siehe Abschnitt 3.2.1) ebenso wie Wulf [65] mit *DyVis* (siehe Abschnitt 3.2.2) einen Darstellungsansatz auf Basis der städtischen Metapher [24]. Sie visualisieren die Applikationsstatik und -dynamik dabei in Form einer Großstadt und bilden Klassen auf Gebäude und Pakete auf Distrikte ab. *TraceCrawler* von Greevy et al. [32, 33] (siehe Abschnitt 3.3.1) erzeugt eine mit DyVis vergleichbare Darstellung, verzichtet dabei jedoch auf die Darstellung einer Distriktstruktur.
Kausalitätsgraphen [66] (siehe Abschnitt 3.3.2) sowie die Tupelraumdarstellung des Linda-Modells (siehe Abschnitt 3.3.3) zeigen hingegen lediglich einzelne Aspekte einer betrachteten Softwareanwendung. So visualisieren Erstere die Anwendungsdynamik in Form eines graphenbasierten Ansatzes, Letztere zeigen dagegen nur die Interaktionen eines Programms mit einem Tupelraum.
Über diese Arbeiten hinaus existieren jedoch noch weitere Veröffentlichungen, welche sich thematisch mit der Visualisierung existierender Softwaresysteme beschäftigen. So betonen Knight und Munro [38] den besonderen Nutzen von Metaphern in der Visualisierung von Softwareanwendungen. Sie vergleichen dabei unterschiedliche dreidimensionale Darstellungsansätze miteinander und präsentieren ihr Visualisierungswerkzeug *Software World*, welches auf Basis der städtischen Metapher die Statik einer Java-Anwendung zu veranschaulichen vermag. Dieses stellt eine Klasse in Form eines separaten Distrikts dar. Ein Gebäude repräsentiert hingegen eine einzelne Methode, dessen Parameteranzahl identisch mit der Zahl der Eingangstüren in das Gebäude ist. Die Höhe des Gebäudes spiegelt dabei die Zeilen an Code der betrachteten Methode wieder. Ansprechende Gebäude- und Straßentexturen verleihen der Darstellung ein realitätsnahes Äußeres.
3D City von Panas et al. [55] bietet einen noch höheren Realismus. Die Autoren bilden Klassen auf Gebäude und Pakete auf Städte ab. Innerhalb einer Stadt werden dabei Methodenaufrufe durch sich bewegende Fahrzeuge modelliert, wohingegen Paket übergreifende Kommunikationsvorgänge mittels Verkehr über Land- und Schifffahrtsstraßen realisiert werden. Ersteres symbolisiert einen bidirektionalen, Letzteres hingegen einen unidirektionalen Nachrichtenaustausch. Die Verkehrsdichte sowie die Art und Geschwindigkeit der Land- und Wasserfahrzeuge dient hierbei als Maß für die Laufzeit und Priorität der repräsentierten Methodenausführungen. Als zusätzliche Funktionalität bietet 3D City die Visualisierung von Quellcodemetriken an. So repräsentiert beispielsweise ein brennendes Gebäude eine häufig aufgerufene Klasse, wohingegen ein verfallende Gebäudefassade eine qualitativ eher schwache Implementierung impliziert.

Wettel und Lanza [62, 63] präsentieren mit *CodeCity* ihr eigenes Visualisierungswerkzeug auf Basis der City-Metapher. Sie bilden Klassen auf Gebäude ab, deren Höhe und Breite ein Maß für die Anzahl der Attribute und Methoden der betrachteten Klasse ist. Pakete werden dabei in Form von Distrikten dargestellt. In einer von den Autoren durchgeführten Studie [64], in welcher CodeCity mit einigen nicht-visuellen Reverse-Engineering-Werkzeugen verglichen wurde, hat sich ihre Applikation als außerordentlich nützlich bei dem Verständnis der statischen Eigenschaften von Softwaresysteme erwiesen.
Balzer et al. [18, 19] beschreiben einen Darstellungsansatz der Anwendungsstatik auf Grundlage der landschaftlichen Metapher. In diesem werden Pakete durch ineinander verschachtelten Sphären symbolisiert. Die äußerste Struktur repräsentiert den Wurzelknoten in der Pakethierarchie. Zoomt man auf eine Sphäre ein, so werden die innen liegenden Pakete der darunter liegenden Hierarchieschicht nach und nach sichtbar. In jeder Sphäre ist dabei eine zweidimensionale Grundfläche integriert, auf welcher Klassen in Form von Kreisen platziert werden, deren Größe abhängig von der Anzahl der enthaltenen Methoden und Attribute ist, welche auf diesen durch unterschiedlich gefärbte Quader repräsentiert werden. Abhängigkeiten zwischen diesen werden dabei durch adjazente Kanten codiert.
Marcus et al. [50] präsentieren mit *sv3D* (kurz für *source viewer 3D*) ein eigenständiges Framework zur Erstellung dreidimensionaler Softwarevisualisierungen. Sie definieren dabei eine Applikation, welches dieses Framework nutzt (eine sogenannte *sv3D-Anwendung*) als ein Quadrupel $\{V, D, S, M\}$. V repräsentiert hierbei die Menge der verwendeten Metaphern, D die Dateien, welche die zu visualisierenden Daten enthalten und aus den Quellcodedateien S gewonnen wurden, und M die Abbildungen von S auf D auf V. sv3D visualisiert diese Informationen anschließend in Form von Quadern unterschiedlicher Höhe, Farbe und Platzierung auf einer zweidimensionalen virtuellen Ebene.
Neben den genannten Veröffentlichungen existieren ebenso einige Arbeiten, welche lediglich eine Teilmenge einer UML-Erweiterungen aus Abschnitt 3.1 beschreiben. So visualisiert *JaVis* von Mehner et al. [51, 53] Programmtraces in Form von Sequenzdiagrammen, identifiziert dabei automatisch Deadlocks und macht diese in der Darstellung kenntlich. Auch Leroux und Exton [42, 43] nutzen Sequenzdiagramme zur Darstellung von Applikationsabläufen, erhöhen deren Aussagekraft jedoch mittels Zustandsdiagrammen. Ihr Visualisierungswerkzeug *COOPE* ermöglicht zwar noch nicht die Darstellbarkeit von Nebenläufigkeit, jedoch sei ein entsprechendes Konzept bereits in der Entwicklung.

9. Zusammenfassung und Fazit

Die Visualisierung von Softwaresystemen ist keine triviale Aufgabe. So muss eine gute Darstellung neben der weitest möglichen Erfüllung ästhetischer Kriterien auch über eine realistische Metapher verfügen, in welcher sich der Betrachter leicht zurechtfindet und aufgrund der Analogie zur realen Welt intuitiv navigieren kann. Doch auch der Umfang der unterstützten Anwendungsaspekte spielt eine große Rolle für dessen Nutzbarkeit. So sollte eine gute Visualisierung auch mindestens die Integration der Statik und Dynamik, nach Möglichkeit sogar von Synchronisationspunkten erlauben.
Im Rahmen dieser Arbeit haben wir verschiedene bestehende Darstellungsansätze präsentiert und anhand eigens aufgestellter Beurteilungskriterien evaluiert. Auffallend viele dieser Ansätze zeigen in einzelnen Bereichen deutliche Schwächen und entsprechend groß ist deren qualitative Diversität. Als eindeutiger Sieger unseres Bewertungsprozesses ging dabei DyVis [65] hervor. Auf Grund seiner Vielzahl positiver Eigenschaften sowie des hohen Erweiterungspotenzials diente die Applikation uns als Ausgangsbasis für die Entwicklung unseres eigenen, ganzheitlichen Visualisierungsansatzes der Statik, Dynamik und Synchronisationspunkte, welchen wir ausführlich vorstellten.
Unser Softwarewerkzeug Synchrovis implementiert diesen Ansatz prototypisch und verfügt neben der reinen Darstellungsfähigkeit über viele nützliche Funktionen wie beispielsweise der automatisierten Trace-Wiedergabe oder der Schnappschuss-Speicherung. Die erzeugte Visualisierung lässt sich dabei mittels zahlreicher Konfigurationseinstellungen individualisieren und in vielen Bereichen an den jeweiligen Geschmack des Betrachters in puncto Form und Farbe anpassen. Darüber hinaus unterstützen umfangreiche Navigations- und Interaktionsmöglichkeiten den Anwender dabei, sich in der Darstellung zurechtzufinden sowie die von ihm gewünschte Information leicht extrahieren zu können. Sowohl bei der Evaluation anhand unserer Bewertungskriterien als auch bei der von uns durchgeführten Expertenbefragung schnitt unsere Applikation durchweg positiv ab.
Wir hoffen, mit dieser Arbeit einen wichtigen Beitrag zur Visualisierung von Softwaresystemen sowie eine gewisse Pionierarbeit geleistet zu haben. So existiert nach unseren Recherchen bislang noch kein weiterer Ansatz, welcher die Facetten einer Anwendung auf vergleichbar ganzheitlicher Weise zu visualisieren vermag. Doch auch Synchrovis ist noch nicht perfekt. So steht beispielsweise die Integration von Synchronisationsmechanismen verteilter Systeme sowie die Erfassung von Speicherbereinigungsaktivitäten noch aus und wären priorisierte Ziele im Rahmen einer nächsten Erweiterung.

A. Fragekatalog der Expertenbefragung zu Synchrovis

A.1. Fragen zur Darstellung der Statik

- Empfinden Sie die Darstellung der Statik als gelungen? Falls nein, in welchen Bereichen sollten Verbesserungen angebracht werden?
- Welche statischen Informationen vermissen Sie?
- Wie empfinden Sie die Darstellung von Relationen?
- Wir vergleichen nun die Darstellung eines Pakets des JPetStores in Synchrovis und in einem UML-Klassendiagramm: Wo sehen die Vor- und Nachteile der Visualisierung von Synchrovis?

A.2. Fragen zur Darstellung der Dynamik

- Empfinden Sie die Darstellung der Dynamik als gelungen? Falls nein, in welchen Bereichen sollten Verbesserungen angebracht werden?
- Welche dynamischen Informationen vermissen Sie?
- Wir vergleichen nun die Darstellung des Demonstrationsbeispiels in Synchrovis und in einem Sequenzdiagramm: Wo sehen Sie die Vorteile und wo die Nachteile von Synchrovis ?

A.3. Fragen zu Darstellung von Synchronisationspunkten

- Empfinden Sie die Darstellung von Monitor-Operationen als gelungen? Falls nein, wie könnte diese verbessert werden?
- Empfinden Sie die Darstellung von Wait-/Notify-Operationen als gelungen? Falls nein, wie könnte diese verbessert werden?
- Empfinden Sie die Darstellung von Thread-Joins als gelungen? Falls nein, wie könnte diese verbessert werden?
- Empfinden Sie die Darstellung der Thread-Zustände (*running*, *waiting on a monitor* und *terminated*) als gelungen? Falls nein, wie könnte diese verbessert werden?

A.4. Allgemeine Fragen zu Synchrovis

- Wie empfinden Sie das Layout der Benutzeroberfläche. Wie könnte dieses gegebenenfalls verbessert werden?
- Wie beurteilen Sie den Funktionsumfang von Synchrovis? Welche Funktionen vermissen Sie?
- Empfinden Sie die Bedienung von Synchrovis als intuitiv? Falls nein, welche Funktionen sollten anders angesteuert werden?
- Empfinden Sie die Gestaltung der Trace-Ansicht als übersichtlich? Falls nein, wie könnte diese verbessert werden?
- Haben Sie sonstige Anmerkungen zu Synchrovis oder dessen Darstellung?

B. Beifügungen

Dieser Arbeit ist eine CD beigefügt, welche die folgenden Daten enthält:

- eine elektronische Fassung dieser Arbeit im PDF-Format (*Thesis.pdf*)
- der Quellcode von Synchrovis
- beispielhafte KDM-Modelle mit dazugehörigen Kieker-Traces zur Demonstration von Synchrovis

Literatur

[1] *Architecture-Driven Modernization: Knowledge Discovery Meta-Model, v1.3.* http://www.omg.org/spec/KDM/1.3/PDF/. Zuletzt aufgerufen: 7. Oktober 2012

[2] *E. W. Dijkstra Archive.* http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF. Zuletzt aufgerufen: 7. Oktober 2012

[3] *Eclipse.* http://www.eclipse.org/. Zuletzt aufgerufen: 7. Oktober 2012

[4] *Eclipse Modeling Framework Project (EMF).* http://www.eclipse.org/modeling/emf/. Zuletzt aufgerufen: 7. Oktober 2012

[5] *Java Platform Standard Edition 6 API Specification.* http://docs.oracle.com/javase/6/docs/api/. Zuletzt aufgerufen: 7. Oktober 2012

[6] *Java Remote Method Invocation (Java RMI) Specification.* http://docs.oracle.com/javase/6/docs/platform/rmi/spec/rmiTOC.html. Zuletzt aufgerufen: 7. Oktober 2012

[7] *Java Virtual Machine Debug Interface Reference.* http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jvmdi-spec.html. Zuletzt aufgerufen: 7. Oktober 2012

[8] *Java3D.* http://java3d.java.net/. Zuletzt aufgerufen: 7. Oktober 2012

[9] *JavaSpaces Service Specification. Version 2.2.* http://river.apache.org/doc/specs/html/js-spec.html. Zuletzt aufgerufen: 7. Oktober 2012

[10] *JUnit Testing Framework.* http://www.junit.org/. Zuletzt aufgerufen: 7. Oktober 2012

[11] *MetricView and MetricView Evolution.* Verfügbar unter: http://www.win.tue.nl/empanada/metricview/. Zuletzt aufgerufen: 7. Oktober 2012

[12] *The OpenGL Graphics System Version 4.3: A Specification.* http://www.opengl.org/registry/doc/glspec43.core.20120806.pdf. Zuletzt aufgerufen: 7. Oktober 2012

[13] Alam, Sazzadul; Boccuzzo, Sandro; Wettel, Richard; Dugerdil, Philippe; Gall, Harald und Lanza, Michele (2009). *Human Machine Interaction.* Kapitel EvoSpaces - Multi-dimensional Navigation Spaces for Software Evolution, S. 167–192. Springer-Verlag

[14] Alam, Sazzadul und Dugerdil, Philippe (2007). *Evospaces: 3D Visualization of Software Architecture.* In *Proceedings of the IEEE International Conference on Software Engineering and Knowledge Engineering*

[15] Alam, Sazzadul und Dugerdil, Philippe (2007). *EvoSpaces Visualization Tool: Exploring Software Architecture in 3D.* In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, S. 269–270. IEEE Computer Society

[16] Artho, Cyrille; Havelund, Klaus und Honiden, Shinichi (2007). *Visualization of Concurrent Program Executions.* In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, S. 541–546. IEEE Computer Society

[17] Balzarotti, Davide; Costa, Paolo und Picco, Gian Pietro (2007). *The LighTS Tuple Space Framework and its Customization for Context-aware Applications. Web Intelligence and Agent Systems*, 5(2):215–231

[18] Balzer, Michael; Noack, Andreas; Deussen, Oliver und Lewerentz, Claus (2004). *Software Landscapes: Visualizing the Structure of Large Software Systems.* In *VisSym, Symposium on Visualization*, S. 261–266. Eurographics Association

[19] Balzer, Michael; Noack, Andreas; Deussen, Oliver und Lewerentz, Claus (2004). *Software Landscapes: Visualizing the Structure of Large Software Systems.* In *VisSym, Symposium on Visualization*, S. 261–266

[20] Ben-Ari, Mordechai (1990). *Principles of Concurrent and Distributed Programming.* Prentice-Hall, Inc.

[21] Carriero, Nicholas und Gelernter, David (1989). *Linda in context. Commun. ACM*, 32(4):444–458

[22] Comer, Douglas (Hg.) (2000). *Internetworking with TCP/IP - Principles, Protocols, and Architectures, Fourth Edition.* Prentice-Hall

[23] Corporation, Microsoft, *Direct3D 11 Reference.* `http://msdn.microsoft.com/en-us/library/windows/desktop/ff476147%28v=vs.85%29.aspx`. Zuletzt aufgerufen: 7. Oktober 2012

[24] Dieberger, Andreas (1997). *A City Metaphor to Support Navigation in Complex Information Spaces.* In *Proceedings of the International Conference on Spatial Information Theory: A Theoretical Basis for GIS*, COSIT '97, S. 53–67. Springer-Verlag

[25] Dijkstra, Edsger W. (1965). *Cooperating Sequential Processes.* Technischer Bericht, Technische Universität Eindhoven, Niederlande

[26] Dijkstra, Edsger W. (1971). *The Origin of Concurrent Programming.* Kapitel Hierarchical Ordering of Sequential Processes, S. 198–227. Springer-Verlag

[27] Dugerdil, Philippe und Alam, Sazzadul (2008). *Execution Trace Visualization in a 3D Space.* In *Proceedings of the Fifth International Conference on Information Technology: New Generations*, ITNG '08, S. 38–43. IEEE Computer Society

[28] Exton, Chris und Kölling, Michael (2000). *Concurrency, Objects and Visualisation.* In *Proceedings of ACM SIGCSE Fourth Australian Computing Education Conference*, ACE 2000, S. 109–115. ACM Press

[29] Freitas, Carla M.D.S.; Luzzardi, Paulo R.G.; Cava, Ricardo A.; Winckler, Marco; Pimenta, Marcelo S. und Nedel, Luciana P. (2002). *On Evaluating Information Visualization Techniques.* In *Proceedings of the working conference on Advanced Visual Interfaces*, S. 373–374. ACM

[30] Gamma, Erich; Helm, Richard und Johnson, Ralph (2009). *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley

[31] Gansner, E. R.; Koutsofios, E.; North, S. C. und Vo, K.-P. (1993). *A Technique for Drawing Directed Graphs. IEEE Transactions on Software Engineering*, 19(3):214–230

[32] Greevy, Orla; Lanza, Michele und Michele, Christoph (2006). *Visualizing live Software Systems in 3D.* In *Proceedings of the 2006 ACM symposium on Software visualization*, SoftVis '06, S. 47–56. ACM

[33] Greevy, Orla; Lanza, Michele und Wysseier, Christoph (2005). *Visualizing Feature Interaction in 3-D.* In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '05, S. 30–35. IEEE Computer Society

[34] van Hoorn, André; Waller, Jan und Hasselbring, Wilhelm (2012). *Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis.* In *Third Joint WOSP/SIPEW International Conference on Performance Engineering*, ICPE '12

[35] Kaufmann, Michael und Wagner, Dorothea (Hg.) (2001). *Drawing Graphs: Methods and Models.* Nummer 2025 in LNCS. Springer-Verlag

[36] Kiczales, Gregor; Lamping, John; Mendhekar, Anurag; Maeda, Chris; Lopes, Cristina; Loingtier, Jean-Marc und Irwin, John (1997). *Aspect-Oriented Programming.* In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Springer Verlag

[37] Kieker Project (2012). *Kieker 1.5 User Guide*. Arbeitsgruppe Software-Engineering, Christian-Albrechts-Universität zu Kiel
URL `http://kieker-monitoring.net/documentation/`

[38] Knight, Claire und Munro, Malcolm (2000). *Virtual but Visible Software*. In *Proceedings of the International Conference on Information Visualisation*, IV '00, S. 198–205. IEEE Computer Society, Washington, DC, USA

[39] Lange, Christian und Chaudron, Michel (2005). *Combining Metrics Data and the Structure of UML Models using GIS Visualization Approaches*. In *Proceedings of the International Conference on Information Technology: Coding and Computing*, Band II von *ITCC '05*, S. 322–326. IEEE Computer Society

[40] Lange, Christian F. J. und Chaudron, Michel R. V. (2007). *Interactive Views to Improve the Comprehension of UML Models - An Experimental Validation*. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, S. 221–230. IEEE Computer Society

[41] Lange, Christian F. J.; Wijns, Martijn A. M. und Chaudron, Michel R. V. (2007). *A Visualization Framework for Task-Oriented Modeling Using UML*. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, S. 289–298. IEEE Computer Society

[42] Leroux, Hugo und Exton, Chris (2001). *COOPE: A Tool for Representing Concurrent Object-oriented Program Execution through Visualisation*. In *Ninth Euromicro Workshop on Parallel and Distributed Processing 2001*, S. 71–76

[43] Leroux, Hugo und Exton, Chris (2001). *Visualising the Execution of Concurrent Object Oriented Programs dynamically using UML*. In *Short Communications and Posters*, Wscg '2001, S. 114–119

[44] Leroux, Hugo; Mingins, Christine und Réquilé-Romanczuk, Annya (2003). *A Conceptual Model for Visualising Concurrent Java Programs Using UML*. Technischer Bericht, School of Computer Science and Software Engineering, Monash Universität, Victoria, Australien

[45] Leroux, Hugo; Mingins, Christine und Réquilé-Romanczuk, Annya (2003). *JACOT: A UML-Based Tool for the Run-Time Inspection of Concurrent Java Programs*. In *Proceedings of the First Workshop on Advancing the State-of-the-Art in Run-Time Inspection at the European Conference for Object-Oriented Programming*, ECOOP '03

[46] Leroux, Hugo; Réquilé-Romanczuk, Annya und Mingins, Christine (2003). *JACOT: A Tool to Dynamically Visualize the Execution of Concurrent Java Programs*. In

Proceedings of the 2nd international conference on Principles and practice of programming in Java, PPPJ '03, S. 201–206. Computer Science Press, Inc.

[47] Li, Xiaoshan; Liu, Zhiming und Jifeng, He (2004). *A Formal Semantics of UML Sequence Diagram.* In *Proceedings of the 2004 Australian Software Engineering Conference*, ASWEC '04, S. 168–190. IEEE Computer Society

[48] Malnati, Giovanni; Cuva, Caterina Maria und Barberis, Claudia (2007). *JThreadSpy: Teaching Multithreading Programming by Analyzing Execution Traces.* In *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '07, S. 3–13. ACM

[49] Malnati, Giovanni; Cuva, Caterina Maria und Barberis, Claudia (2008). *JThreadSpy: A Tool for Improving the Effectiveness of Concurrent System Teaching and Learning.* In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, Band 5 von *CSSE '08*, S. 549–552. IEEE Computer Society

[50] Marcus, Andrian; Feng, Louis und Maletic, Jonathan I. (2003). *3D Representations for Software Visualization.* In *Proceedings of the 2003 ACM Symposium on Software visualization*, SoftVis '03, S. 27–ff. ACM

[51] Mehner, Katharina (2002). *JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs.* In *Revised Lectures on Software Visualization, International Seminar*, S. 163–175. Springer-Verlag

[52] Mehner, Katharina und Wagner, Annika (2000). *Visualizing the Synchronization of Java-Threads with UML.* In *Proceedings of the 2000 IEEE International Symposium on Visual Languages*, VL '00, S. 199–. IEEE Computer Society

[53] Mehner, Katharina und Weymann, Bernd (2001). *Visualization and Debugging of Concurrent Java Programs with UML.* In W. de Pauw, J. Stasko, S. Reiss (Hg.), *Proceedings of the Workshop on Software Visualization*

[54] Object Management Group, *OMG Unified Modelling Language (OMG UML) Superstructure Specification (Version 2.4.1).* http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/. Zuletzt aufgerufen: 7. Oktober 2012

[55] Panas, Thomas; Berrigan, Rebecca und Grundy, John (2003). *A 3D Metaphor for Software Production Visualization.* In *Proceedings of the Seventh International Conference on Information Visualization*, IV '03, S. 314–319. IEEE Computer Society

[56] Panas, Thomas; Lincke, Rüdiger und Löwe, Welf (2005). *Online-configuration of software visualizations with Vizz3D.* In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, S. 173–182. ACM

[57] Pérez-Castillo, Ricardo; de Guzmán, Ignacio García-Rodríguez und Piattini, Mario (2011). *Knowledge Discovery Metamodel-ISO/IEC 19506: A Standard to Modernize Legacy Systems. Computer Standards & Interfaces*, 33(6):519–532

[58] Shneiderman, Ben (1996). *The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations.* In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, VL '96, S. 336–343. IEEE Computer Society

[59] Silberschatz, Abraham; Gagne, Greg und Galvin, Peter Baer (2005). *Operating System Concepts.* John Wiley & Sons, Inc, 7 Auflage

[60] The Eclipse Foundation, *The AspectJ Project.* `http://www.eclipse.org/aspectj/`. Zuletzt aufgerufen: 7. Oktober 2012

[61] Tilley, Scott und Huang, Shihong (2003). *A Qualitative Assessment of the Efficacy of UML Diagrams as a Form of Graphical Documentation in Aiding Program Understanding.* In *Proceedings of the 21st annual international conference on Documentation*, SIGDOC '03, S. 184–191. ACM

[62] Wettel, Richard und Lanza, Michele (2007). *Visualizing Software Systems as Cities.* In *4th IEEE international workshop on visualizing software for understanding and analysis*, S. 92–99. Society Press

[63] Wettel, Richard und Lanza, Michele (2008). *CodeCity: 3D visualization of large-scale software.* In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, S. 921–922. ACM

[64] Wettel, Richard; Lanza, Michele und Robbes, Romain (2011). *Software Systems as Cities: A Controlled Experiment.* In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, S. 551–560. ACM

[65] Wulf, Christian (2010). *Runtime Visualization of Static and Dynamic Architectural Views of a Software System to identify Performance Problems.* Bachelorarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik, Arbeitsgruppe Software Engineering

[66] Zernick, Dror; Snir, Marc und Malki, Dalia (1992). *Using Visualization Tools to Understand Concurrency. IEEE Softw.*, 9(3):87–92