Diplom-Arbeit

Ein Visualisierungssystem zur objektorientierten Animation von Java-Programmen

Entwurf und Implementierung

Autor: André Vratislavsky

Betreuer: Prof. Dr.-Ing. Klaus-Peter Löhr

Animation

- Abfolge von Einzelbildern
- Messwerte werden als Bild dargestellt
- Für fließende Bewegung Interpolation notwendig
- Interpolation von Messwerten aber nur bedingt sinnvoll
  - Funktion über die Zeit i.A. unbekannt
  - Bei diskreten Werten gibt es keine Zwischenwerte
- Gestaltung mit „fliegenden“ Variablen fragwürdig
- Animation hier als Visualisierung realer Zustände

Objektbeziehungen

- Referenz
  - Entspricht Java-Semantik
  - Darstellung als Pfeil im Objektdiagramm
- Komponente
  - Aggregation, Teil-von-Beziehung
  - Existenzabhängiges Wert-Attribut
  - Unterobjekt, welches semantisch eng zu seinem umschließenden Objekt gehört
  - Hier Darstellung durch verschachtelte Rechtecke (in UML Pfeil mit Raute)

Besitzverhältnisse

- Es gibt gewisse Enthaltensein-Beziehung durch Objektschachtelung in Java
- Komponenten sind nicht in Java vorgesehen
  - Aufspannender Baum kein Komponentenbaum
- [Clarke, Noble, Potter]: *Ownership Types*
  - Konzept selbst definierbarer Enthaltensein-Beziehungen
  - Abstrakte Beschreibung der Eigenschaften
  - Besitzverhältnisse bestimmen Sichtbarkeit
    - Unterobjekt soll Zugriffschutz genießen
    - Bsp.: Unterkomponenten nicht öffentlich schreibbar

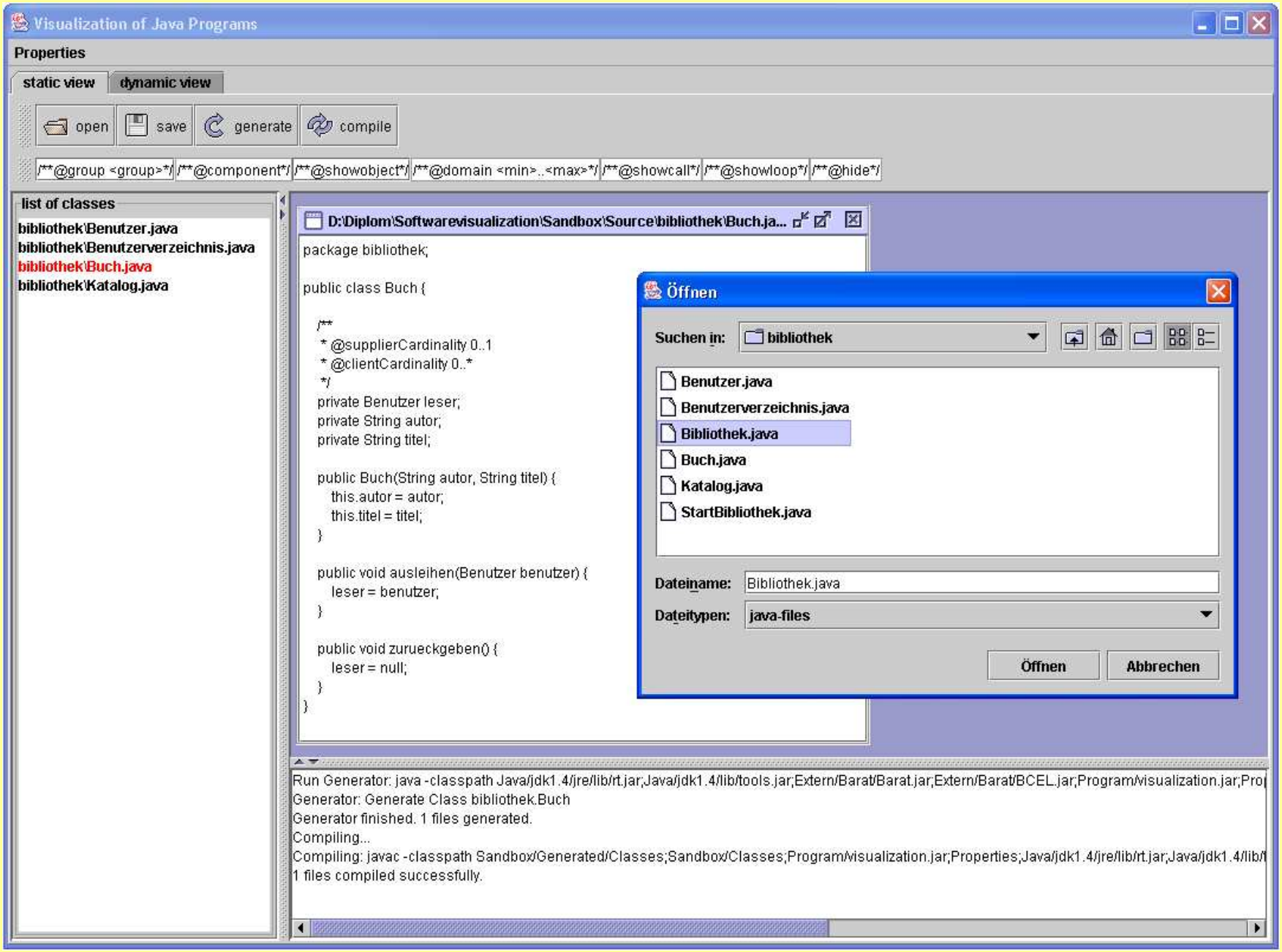
Sichtbarkeit

- Sichtbarkeit von Unterobjekten in Java
  - Geregelt durch `public`, `protected`, `private` (Betrifft Paketstruktur und Vererbung)
  - Wünschenswert wären: *lesbar*, *schreibbar*
  - Feinkörnigere Sichtbarkeit mit Zugriffsmethoden („Handarbeit“, nicht in Java-Semantik enthalten)
- Von Sichtbarkeit kann auf Besitzverhältnisse geschlossen werden
  - Beschreibung der Besitzverhältnisse anwenden
    - Bsp.: Unterkomponenten nicht öffentlich schreibbar
    - Impl.: Öffentliche Variable enthält keine Unterkomponente

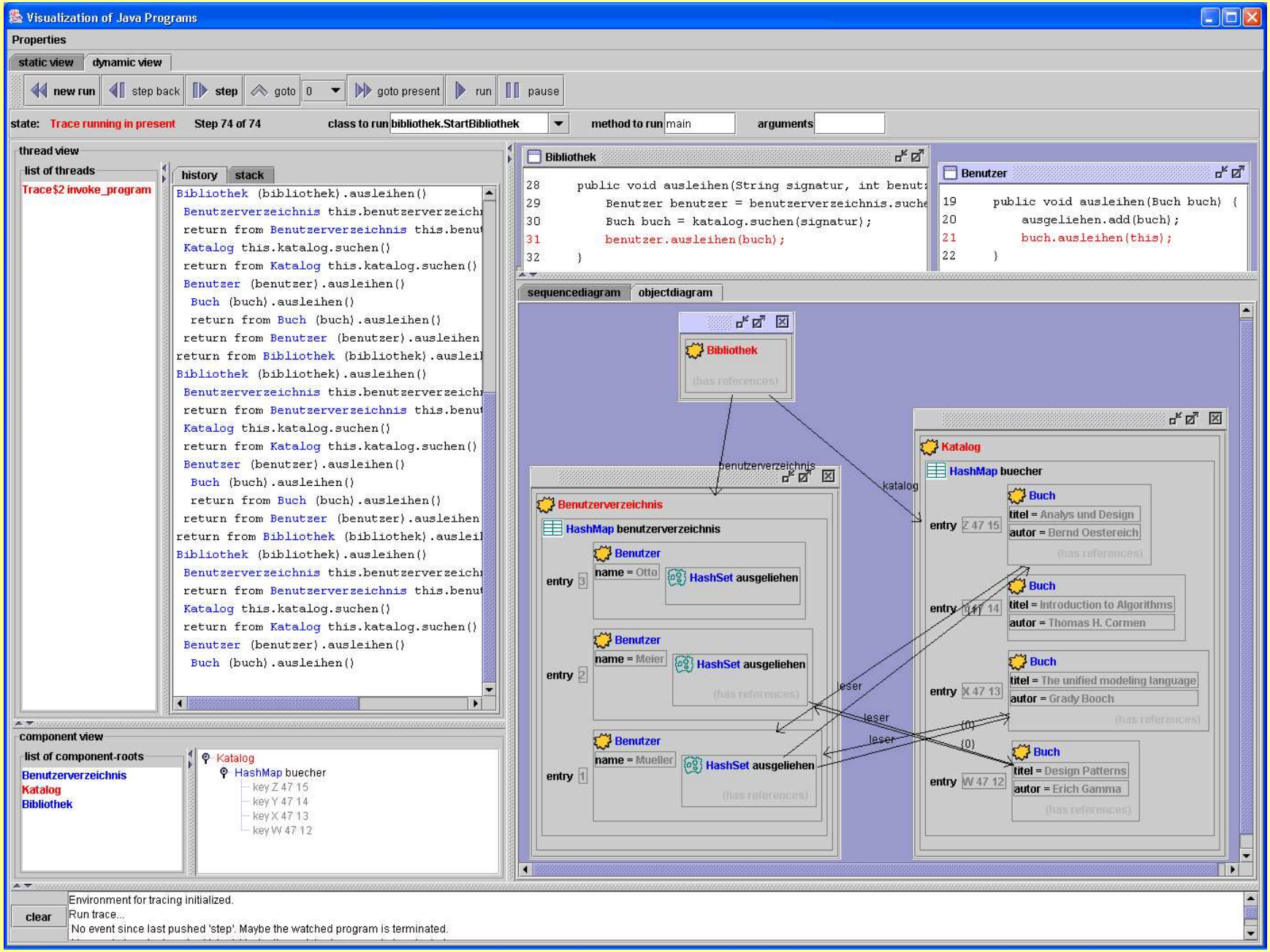
Automatische Erkennung von Komponenten

- Analyse der Sichtbarkeiten im Programm
  - Schwierig, da teilweise semantisch in Methoden versteckt
- Interpretation der Besitzverhältnisse
  - Bsp.: Wenn keine öffentliche Methode existiert, die schreibenden Zugriff auf eine private Variable hat, so kann das referenzierte Objekt als Komponente betrachtet werden, da Komponenten nicht von fremden Objekten modifiziert werden sollen
  - Notwendig zu definieren, was schreibender Zugriff ist
    - Darf nicht zu restriktiv definiert sein

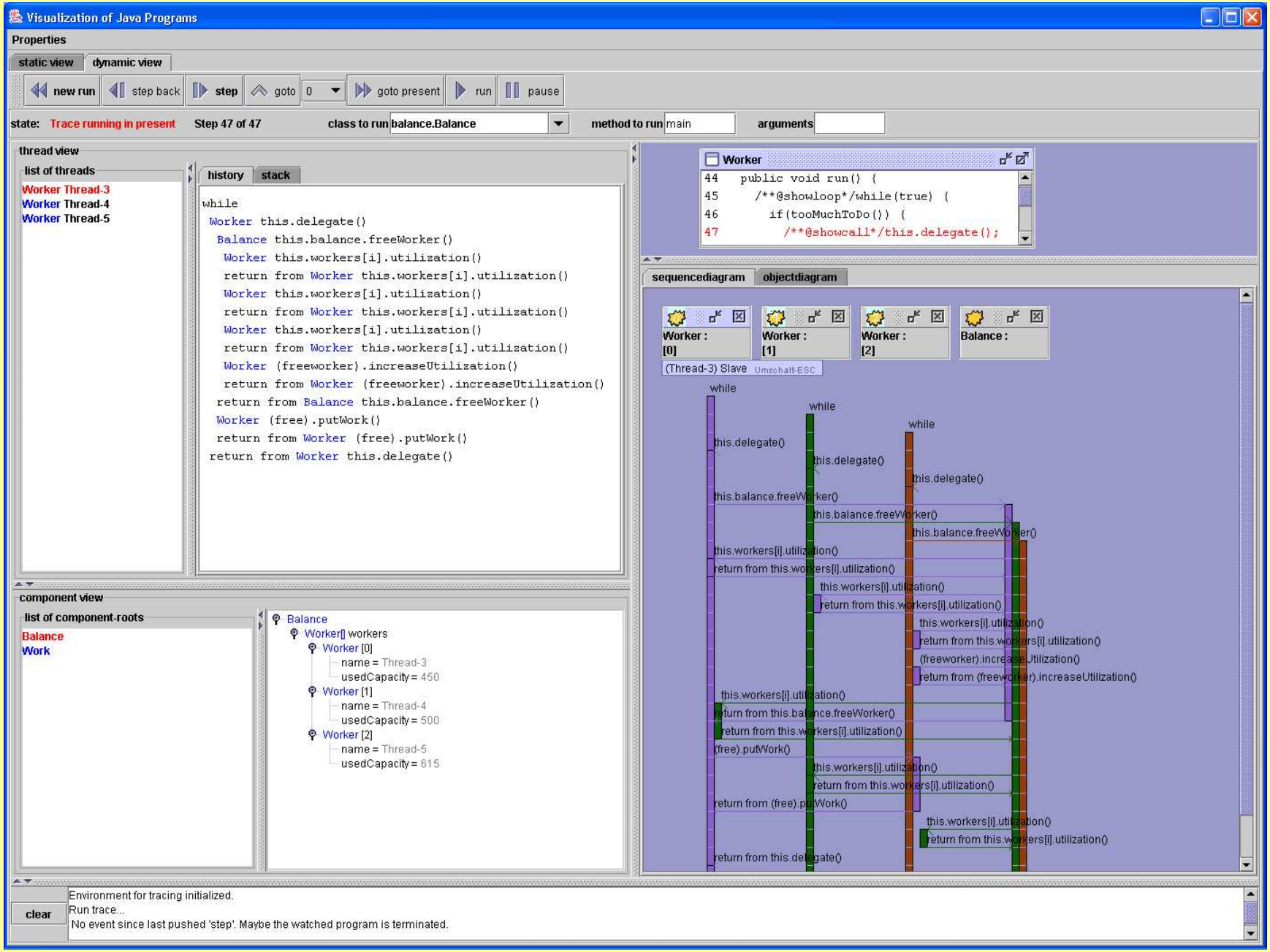
Statische Ansicht



Dynamische Ansicht: Objektdiagramm



Dynamische Ansicht: Ablaufdiagramm



Funktionsumfang I

- Zeitabhängige Visualisierung von
  - Datenstrukturen durch Objektdiagramme
  - Kommunikation zwischen den Objekten (Methodenaufrufen) durch Ablaufdiagramme
- Schrittweise Ausführung vorwärts und „rückwärts“
- Darstellung mehrerer Threads bei nebenläufigen Programmen
- Annotationen im Quelltext eines inspizierten Programms
  - Geben Detailtiefe der Darstellung vor
  - Erweitern Variablen um Semantik: *Komponenten (Aggregationen)*

Funktionsumfang II

- Methodenaufrufstack und Historie der Aufrufe für jeden Thread
- Threads im Ablaufdiagramm farblich gekennzeichnet
- Inhalt der Diagramme kann interaktiv beeinflusst werden
- Darstellung von Quelltext (aktueller Aufrufe)
- Entkopplung von inspiziertem Programm und Visualisierungstool (getrennte JVM's)
- Neben der Animation von Software auch Animation von Algorithmen möglich

Tags: Annotationen im Quelltext

- Im Quelltext können spezielle Kommentare, sogenannte Tags, angebracht werden, die vom Visualisierungstool automatisch ausgewertet werden
- Tags können gewünschte bzw. ungewünschte Methodenaufrufe, Schleifendurchläufe, Unterobjekte und Referenzen markieren

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

- Es werden zwei zueinander orthogonale Gruppen von Tags unterschieden...

1) Tags zur Ergänzung von Semantik

- Erweiterung um Semantik, die nicht mit Java ausgedrückt werden kann
- Definieren **wie** eine Variable dargestellt wird
  - **Definitionsbereich** von Zahlenattributen
  - **Komponentenbegriff**
  - Beispiele:

```
/**
 * @component
 */
Object someObj;
```

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

Tag an Felddeklaration	Typ des aufgerufenen Feldes
/**@range <minimum>..<maximum>*/	Primitiver Zahlentyp
/**@component*/	komplexer Typ

2) Tags zur Auswahl der darzustellenden Objekte und Abläufe

- Definieren, **was** dargestellt wird
  - Inhalt von Variablen
  - Schleifen und Methodenaufrufe
- Grundeinstellung wird von `hide` / `show` überschrieben
- Beispiel: `/** @showloop */ if (someObj.method()) {...}`

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

Beispiele

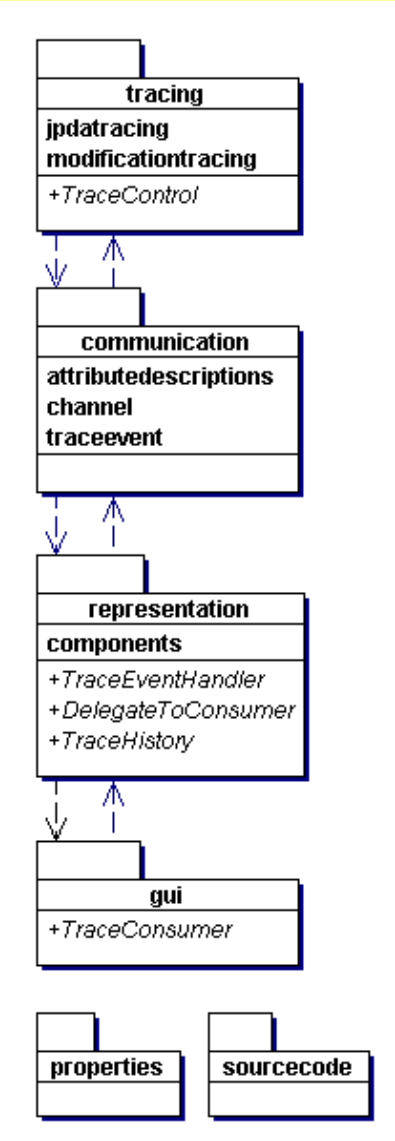
- /**
  * @showloop
  * @showcall
  */
  if (object.method()) {...}
- /**
  * @showcall
  * @showobject
  */
  obj = object.method();
- /**
  * @show
  * @component
  * @group somegroup
  */
  Object someObj;
- int max=10;
  /**
  * @show
  * @range 0..max
  */
  int someInt;
- /**
  * @group somegroup
  */
  class MyClass {...}

Verfahren der Quelltextmodifikation

- *Wie kommt man zur Laufzeit an die Informationen über den Zustand der Objekte und den Ablauf des darzustellenden Programms?*
- Gewinnung von Informationen aus dem inspizierten Programm durch vorhergehende Instrumentierung des Quellcodes
  - Analyse des originalen Quellcodes
  - Einfügen spezieller Anweisungen, die die gewünschten Informationen erfassen
  - Informationsverarbeitung findet **nicht** im inspizierten Programm statt
- Entwurf eines Generators notwendig

Schichtenstruktur des Entwurfs

- Ablaufverfolgung (`tracing`)
  - Generator
  - Schnittstelle für inspiziertes Programm
- Kommunikation (`communication`)
  - Beschreibungen der Daten
  - Kommunikation über RMI
- Repräsentation (`representation`)
  - Speicherung der Abläufe
  - Aufbau von Komponentenbäumen
- Graphische Benutzerschnittstelle (`gui`)
- Sonstige Hilfspakete (`properties`, `sourcecode`)
- Kommunikation zwischen den Schichten über Interfaces

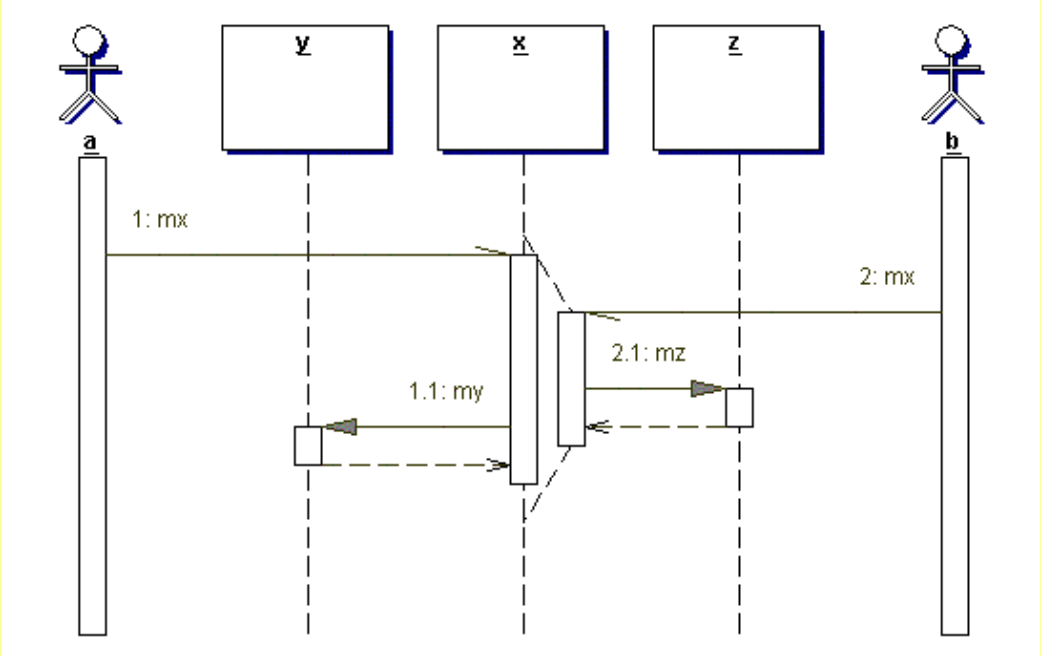


Notwendige Schritte für Animation

- 1. Schritt: Generierung (statische Ansicht in GUI)
  - Gegebenenfalls Quelltext mit Tags versehen
  - Analyse des Programms (mit *Barat*)
  - Dabei Auswertung der Annotationen
  - Bereitstellung lauffähigen Codes mit speziellen Anweisungen, die die Informationen für die Visualisierung an die Ablaufverfolgung übermitteln
  - Übersetzen
- 2. Schritt: Visualisierung (dyn. Ansicht in GUI)
  - Repräsentation des inspizierten Programms während der Laufzeit aufbauen und aktualisieren
  - Graphische Darstellung

Ablaufverfolgung von Methodenaufrufen

- Sowohl Aufrufender (Caller) als auch Aufgerufener (Callee) werden zur Verfolgung gebraucht
  - Reihenfolge der Aufrufe auf Objekt x reicht nicht
  - Bsp.: (a ruft auf `x.mx`) → (x ruft auf `y.my`)



- Ereignisse werden beim Aufruf einer Methode vom **Aufrufenden** gemeldet, da nur dieser sowohl den **Aufrufenden** als auch den **Aufgerufenen** kennt
  - Aufruf von Bibliothekmethoden kann visualisiert werden

Ablaufverfolgung

- Anmerkung: Der Stack von gemeldeten geschachtelten Methodenaufrufen kann Lücken enthalten, wenn nicht alle Aufrufe gemeldet werden sollen (callee eines gemeldeten Aufrufs ungleich caller des enthaltenen Aufrufs)
- Die Schnittstelle zwischen dem inspizierten Programm und der Ablaufverfolgung sollte möglichst einfach sein
  - Keine Verwaltung von Objekten oder Zuständen der Ablaufverfolgung im inspizierten Programm
  - Inspiziertes Programm meldet Ereignisse durch Aufruf statischer Methoden der Ablaufverfolgung

Statische Methoden der Ablaufverfolgung zum Melden von Änderungen an der Objektstruktur

- Werden hinter dem schreibenden Zugriff eingefügt
  - `referenceChanged` für Referenzen
  - `componentChanged` für Komponenten
  - `componentCreated` für neue Objekte
    - Objekte von Bibliothekklassen können visualisiert werden

```
public class Benutzer {
  ...
  public Benutzer(String name) {
    ...
    /**@show*/this.name = name;
    /*generated*/Trace.componentChanged(Thread.currentThread(), this,
                 "name", name, "bibliothek.Benutzer", "<init>",
                 new String[]{"java.lang.String"}, 15, 17, 16, 0);
    /* Thread inThread, Object inObject,
       String varname, Object comp, String inClass, String inMethod,
       String[] inMethodParameter, int codebegin, int codeend, int line)/*
  }
}
```

Statische Methoden der Ablaufverfolgung zum Melden von Methodenaufrufen und Schleifen

- Methodenaufrufe werden von `beginOfCall` und `endOfCall` eingeschlossen (Schleifen analog)
- Geschachtelte Aufrufe ergeben entsprechende Historie
- `endOfCall` muss auch aufgerufen werden, falls
  - aufgerufene Methode mit einer Exception abbricht
  - aufgerufene Methode in einem return-Statement steht
- Realisierung durch `try / finally` Klausel
  - `endOfCall` in `finally` Klausel wird in jedem Falle aufgerufen

Beispiel für Meldung eines Methodenaufrufs

```
public class Worker extends java.lang.Thread {
  ...
  void delegate() {
    /*generated*/Worker free;
    /*generated*/try {
    /*generated*/Trace.beginOfCall(Thread.currentThread(), this, this.balance,
                "this.balance", 2, "freeWorker", "balance.Worker",
                "delegate", new String[]{}, 49, 53, 50, 0);
    /* Thread inThread, Object caller, Object callee,
       String calleeCode, int kindOfRelation, String method, String inClass,
       String inMethod, String[] inMethodParameter,
       int codebegin, int codeend, int line, int presentation)*/
    /**
     * @showcall
     */
    /*Worker */free = this.balance.freeWorker();
    /*generated*/}
    /*generated*/finally {
    /*generated*/Trace.endOfCall(Thread.currentThread(), this, this.balance,
                "balance", 0, "freeWorker", "balance.Worker",
                "delegate", new String[]{}, 49, 53, 50, 0);
    /*generated*/}
  }
}
```

Entkopplung von inspiziertem Programm und Visualisierung

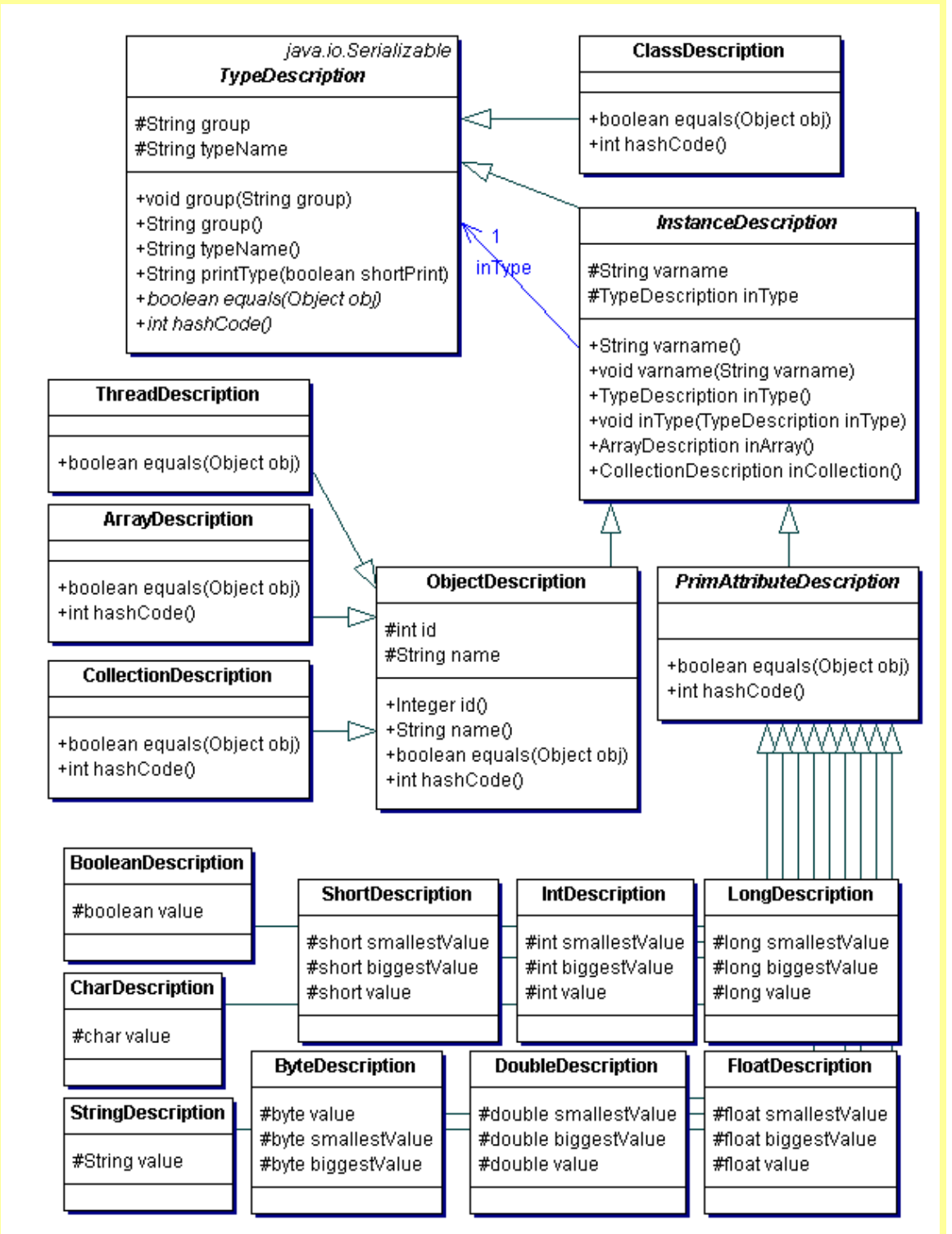
- Forderung: Inspiziertes Programm soll bei laufender GUI beendet und neu gestartet werden können
- Daraus resultiert: Unbekannte Threads des inspizierten Programms müssen vernichtet werden
  - 1. Möglichkeit: Threads synchronisieren
  - 2. Möglichkeit: VM mit inspiziertem Programm beenden
- Weitere Forderung: Keine Rückwirkungen von der Visualisierung auf das inspizierte Programm
- Lösung: Ablaufverfolgung und Repräsentationsschicht auf zwei getrennten JVM's

Beschreibung von Objekten in Nachrichten und Repräsentation

- Zu einem Objekt im inspizierten Programm gibt es i.A. viele Nachrichten / Beschreibungen
  - Beschreibung der Identität des originalen Objekts
    - Bei Attributen primitiven Typs (und Behältern) durch den **Variablennamen** und das **umschließende Objekt** (bzw. die umschließende Klasse im statischen Fall)
    - Bei komplexen Objekten durch **Identifikationsnummer** mit dem Hashcode des originalen Objekts
- Zur Darstellung in der Repräsentationsschicht darf es für ein Objekt nur einen Repräsentanten geben
  - Abbildung der Beschreibungen auf Repräsentanten (`equals` und `hashCode` entsprechend implementiert)

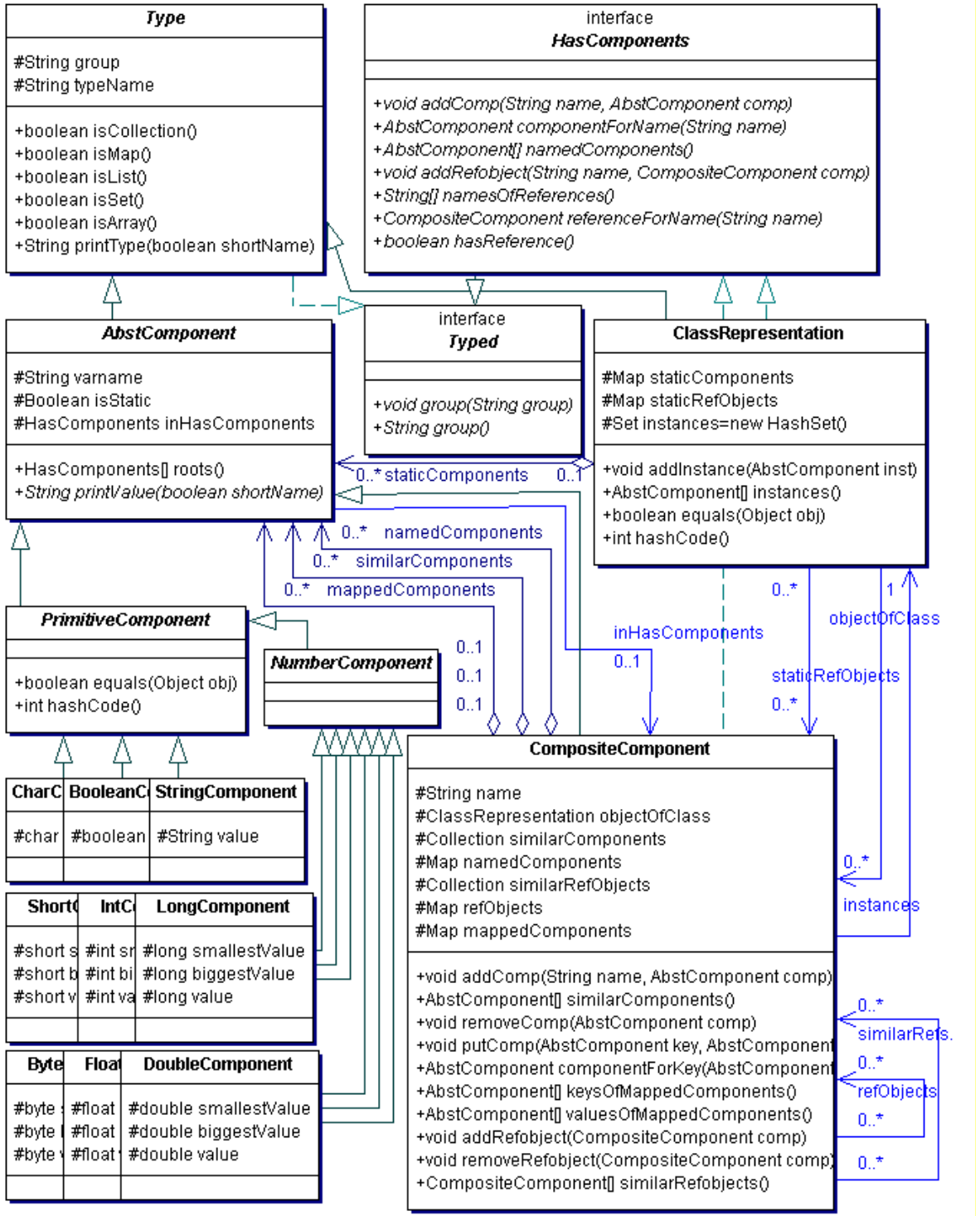
Beschreibung von Objekten komplexen und primitiven Typs

- Beschreibung ist Ausschnitt aus Objektstruktur
- Unterklassen von `TypeDescription`
- Gleichheitssemantik für Abbildung von Beschreibungen zu Repräsentanten
  - `equals`, `hashCode`



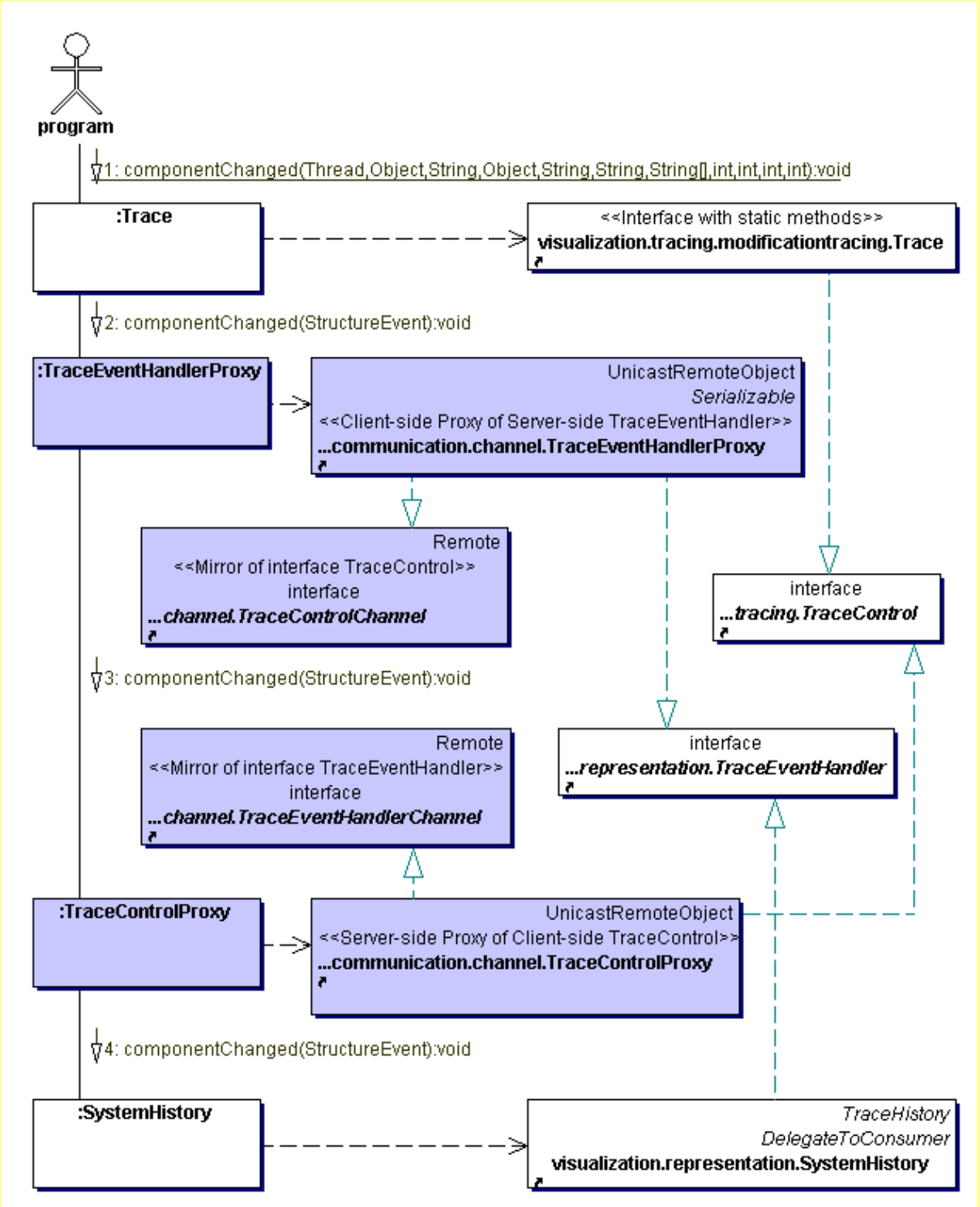
Repräsentation von Objekten primitiven und komplexen Typs und ihren Beziehungen

- Unterklassen von `AbstComponent`
- Zusammengesetzte / Primitive Komponenten
- Unterkomponenten und Referenzen
- Benannte Komp's, Ref's Elemente von Mengen
- Objektwertige / Statische Komp's, Ref's



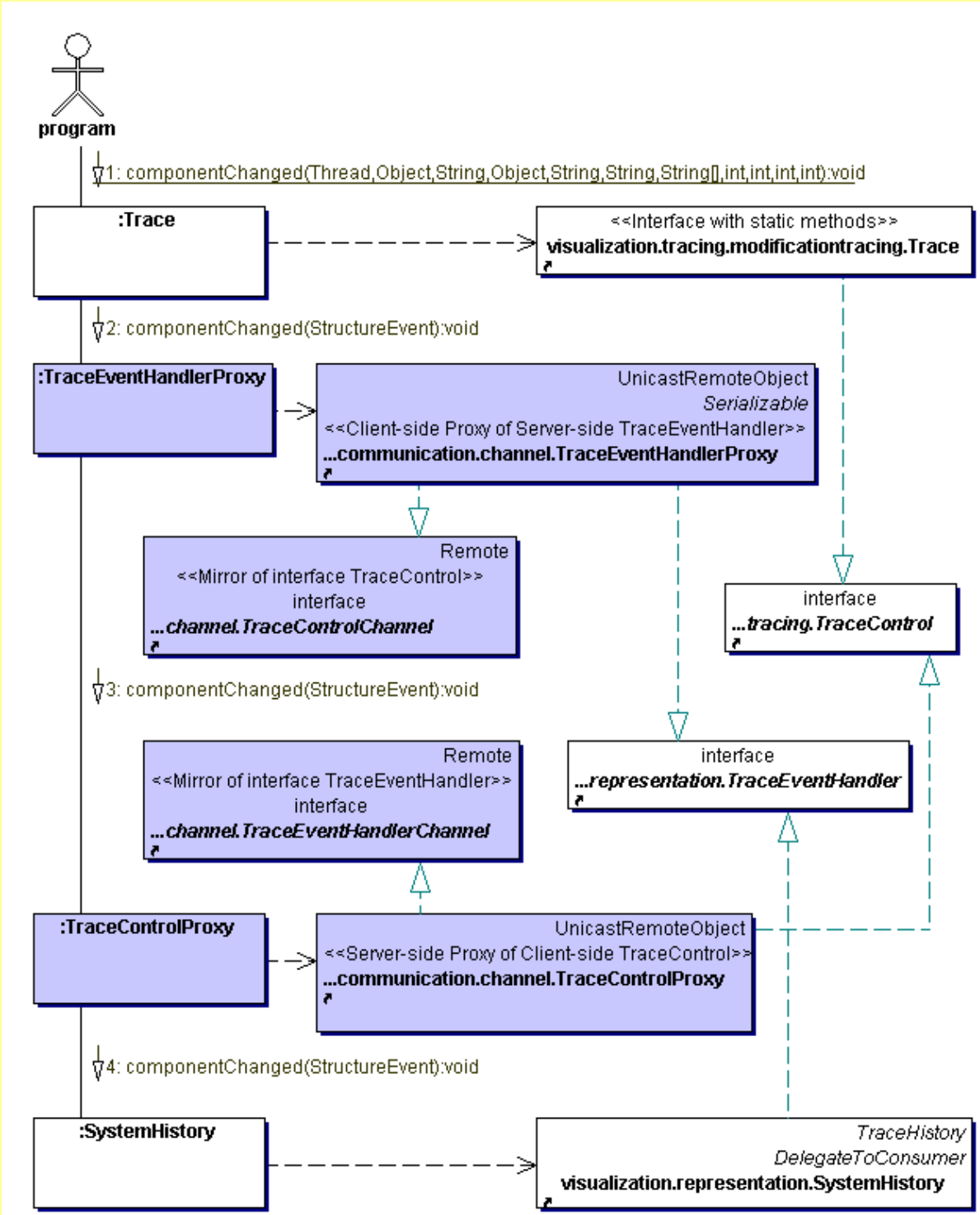
Kommunikation über RMI

- Visualisierungstool (Server) startet eigenen Prozess (JVM) für inspiziertes Programm (Client)
- Client-Seite wird initialisiert
  - Vertreter (Proxy) der Repräsentationsschicht
  - Ablaufverfolgung Trace
  - Inspiziertes Programm
- Trace nimmt Verbindung zum Server auf (über RMI-Registry)



Kommunikation über RMI

- `TraceEventHandler` wird von `SystemHistory` und dem Proxy `TraceEventHandlerProxy` implementiert
- `TraceEventHandlerProxy` und `TraceControlProxy` sind zwischen `Trace` und `SystemHistory` eingeschleust
- Aufruf von `TraceControl` analog über `TraceControlProxy`



Graphische Oberfläche (GUI)

- Implementierung mit Swing
- Trennung von Darstellung und Repräsentation der dargestellten Daten
  - Speichern und Verarbeiten von Daten nicht in der GUI sondern in der Repräsentationsschicht
  - Swing-Modelle vermitteln Repräsentation-Darstellung
- Objektdiagramme bestehen aus geschachtelten `JPanel` in inneren Fenstern (`JInternalFrame`)
- Relationen und Aufrufe auf `JPanel` zeichnen, transparent über Objektdiagramme gelegt
  - Expansionslinien: Jarvis's March

Quellen

- www.inf.fu-berlin.de/~vratisla/Diplom/Diplom.html
- Barat by Boris Bokowski
  - sourceforge.net/projects/barat
  - www.boris.bokowski.de/boris/barat/index.html
- [Clarke, Noble, Potter]
  - David G. Clarke, James Noble, John M. Potter, Simple Ownership Types for Object Containment, 2001, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, clad@cs.uu.nl, School of Mathematical and Computing Sciences, Victoria University, Wellington, New Zealand, kjx@mcs.vuw.ac.nz, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, potter@cse.unsw.edu.au
  - David Clarke, Object Ownership and Containment, An Object Calculus with Ownership and Containment, www.cs.uu.nl/~dave/abstracts.html
  - David G. Clarke, John M. Potter, James Noble, Ownership Types for Flexible Alias Protection, 1998
  - James Noble, Jan Vitek, John Potter, Flexible Alias Protection, 1998