Nutzungshinweise für den Sequencediagrammer

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Thema:	Nutzungshinweise für den Sequencediagrammer
Autoren:	Prof. Dr. Stephan Kleuker
Version / Datum:	0.1 / 16.6.2021
Empfänger:	Teilnehmer der Lehrveranstaltungen im Bereich Programmierung der Studiengänge Informatik

Der Sequencediagrammer ermöglicht es parallel zu einem laufenden Java-Programm die Methodenaufrufe und Ergebnisse in einem Sequenzdiagramm darzustellen. Er ist damit ein interessantes Hilfsinstrument existierende Programme zu analysieren und deren interne Zusammenhänge darzustellen. Dies kann z. B. beim Erlernen von Design-Pattern hilfreich sein. Die Nutzungsmöglichkeiten und Restriktionen werden in diesem Dokument beschrieben.

Die vorliegende Software ist als Prototyp das Ergebnis einer explorativen Technologiestudie zur Bearbeitung von Java-ByteCode und wird nicht weiterentwickelt. Die Nutzung ist frei, wenn man zustimmt, dass der Autor für keine der bei Nutzung entstehenden Schäden verantwortlich ist. Es wird die Javassist-Bibliothek (https://www.javassist.org/) genutzt, deren Lizenz zu beachten ist.

1 Einführendes Beispiel

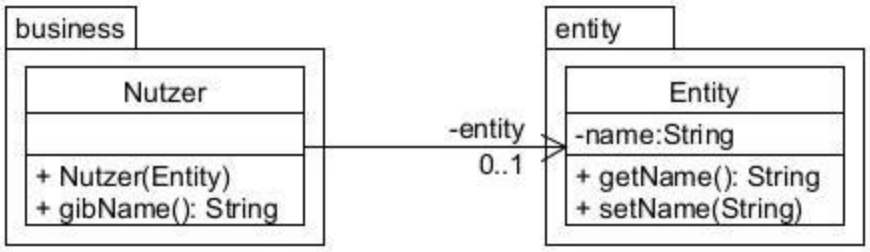

Abb. 1: benutzte Beispielklassen

In diesem Beispiel werden innerhalb einer Main-Methode zwei Objekte erzeugt und einige Methoden aufgerufen. Die fachlichen Klassen mit den angebotenen Methoden sind in Abb. 1 dargestellt. Der eigentliche Programmcode sieht wie folgt aus.

```
package entity;

public class Entity {
	private String name;

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}

package business;
import entity.Entity;

public class Nutzer {
	private Entity entity;

	public Nutzer(Entity entity) {
		this.entity = entity;
	}

	public String gibName() {
		return this.entity.getName();
	}
}
```

Die Klassen werden im folgenden Programm genutzt.

```
package main;

import business.Nutzer;
import entity.Entity;
```

```
public class MainSeq {

	public static void main(String[] s) {
		Entity e = new Entity();
		e.setName("Hai");
		Nutzer n = new Nutzer(e);
		System.out.println(n.gibName());
	}
}
```

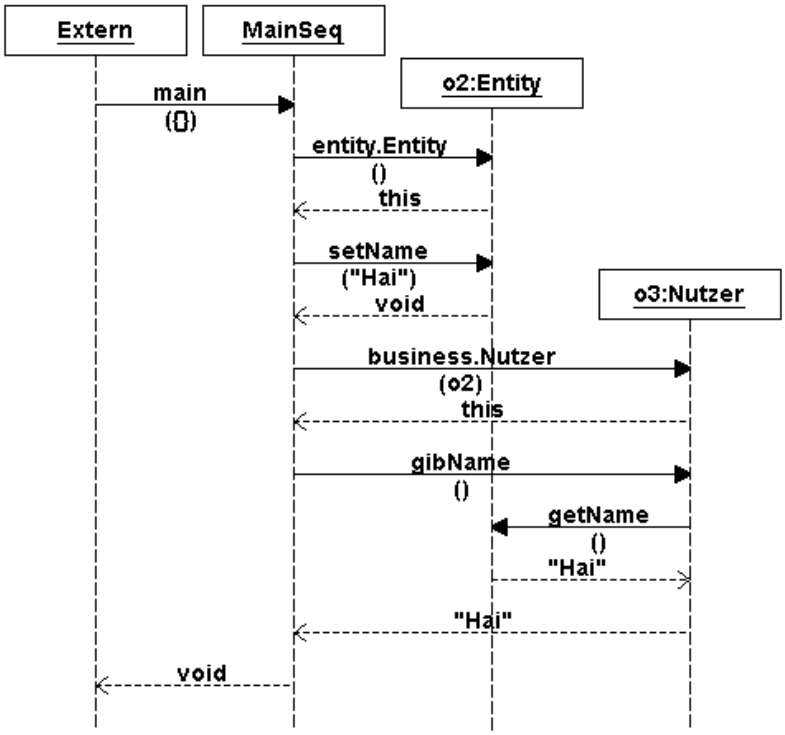

Abb. 2: generiertes Sequenzdiagramm

Läuft das Programm zusammen mit dem Sequencediagrammer wird das in Abb. 2 gezeigte Ergebnis ausgegeben. Die generelle Struktur eines Sequenzdiagramms ist erkennbar. In der obersten Zeile stehen die Objekte, die zum Zeitpunkt des Starts des Diagramms bereits vorhanden sind. Der Aufruf der main-Methode erfolgt vom System, dass durch ein „Extern"-Objekt symbolisiert wird. Der Methodenname steht immer oberhalb des aufrufenden Pfeils, die Parameter stehen in Klammern darunter. Da es sich bei main() um eine Klassenmethode handelt, erfolgt der Aufruf bei dem die Klasse MainSeq repräsentierenden Objekt. Eben ein solches Klassenobjekt gehört zu jeder in Java existierenden Klasse.

Der zeitliche Ablauf erfolgt in Sequenzdiagrammen von oben nach unten. Werden Objekte neu erzeugt, wird der zugehörige Kasten zum Zeitpunkt der Erstellung angezeigt. In der UML-Definition zeigt dabei der Konstruktoraufruf direkt auf diesen Kasten. Um die Darstellung kompakt zu halten, wird der Konstruktoraufruf hier immer unmittelbar unter dem neuen Objekt angegeben. Jedes benutzte oder erzeugte Objekt ist damit als eigener Kasten im Diagramm sichtbar, damit können beliebig viele Objekte einer Klasse getrennt voneinander im Sequenzdiagramm erscheinen.

Für jeden Aufruf steht das jeweilige Ergebnis auf einem gestrichelten Pfeil im Sequenzdiagramm, dies ist bei Konstruktoraufrufen das jeweilige Objekt selbst (this). Bei Methodenaufrufen ohne Rückgabe seht „void“ auf dem Rückgabepfeil. Beim Aufruf der Methode setName() zeigt das Diagramm, dass der Aufrufer das Klassenobjekt mit der main()-Methode ist und dass die Methode beim gerade erzeugten Objekt vom Typ Entity aufgerufen wird.

Die Objekte erhalten automatisch einen mit „o“ beginnenden Namen. Dies wird beim Aufruf des Nutzer-Konstruktors deutlich, der das vorher erzeugte Objekt übergeben bekommt. Der Aufruf der Methode gibName() des Nutzer-Objekts führt dazu, dass dieses zunächst die Methode getName() beim Entity-Objekt aufruft und dann das Ergebnis zurückliefert.

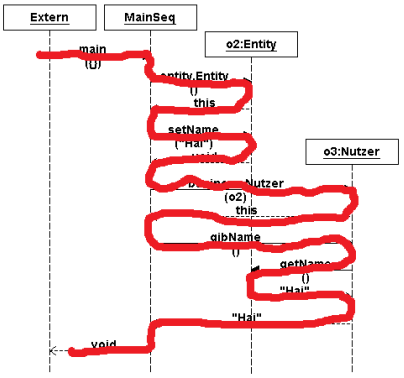

Abb. 3: roter Faden eines Sequenzdiagramms

Die UML erlaubt einige Varianten in der Darstellung von Sequenzdiagrammen. Hier wird eine detaillierte gewählt, bei der u. a. alle Rückgaben immer mit einem eigenen Pfeil eingezeichnet werden. Das hat den Vorteil, dass es einen „roten Pfaden“ durch das Diagramm gibt, der alle Pfeile in der angegebenen Richtung durchläuft, wie es auch in Abb. 3 skizziert ist. Der rote Faden entspricht genau dem Verhalten des untersuchten Programms.

2 Aufruf und grundlegende Nutzung

Die Zip-Datei des Werkzeugs kann von der Webseite http://home.edvsz.hs-osnabrueck.de/skleuker/gsdet/ geladen werden. Diese Zip-Datei wird in einem beliebigen sinnvollen Verzeichnis ausgepackt. Als Beispielpfad wird hier `C:\tmp\Jars` angenommen.

Der Sequenzdiagrammer muss als sogenannter Java-Agent beim Start des Programms als Parameter mit angegeben werden, er wird nicht als Bibliothek eingebunden. Statt eines Aufrufs

```
java main.MainProg
```

muss jetzt stehen

```
java -javaagent:C:\tmp\Jars\Sequencediagrammer.jar main.MainProg
```

Diese Ergänzung ist auch in allen gängigen Entwicklungswerkzeugen möglich. Eclipse bietet dazu mehrere Wege an, die Argumente zum Start der VM anzugeben. Wird z. B. ein Java-Programm zunächst über den Run-Button gestartet, erzeugt Eclipse automatisch eine Start-Konfiguration, die u. a. über den Launcher bearbeitbar ist. Dazu wird auf das gräuliche Zahnrad rechts in der Abb. 4 geklickt.

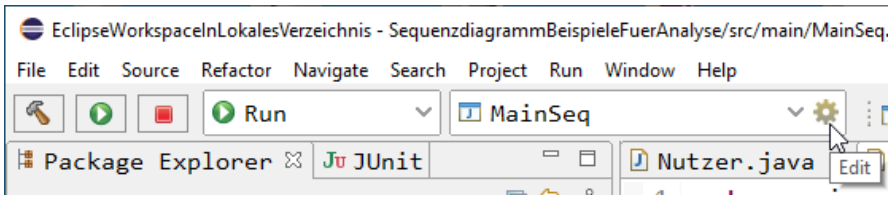

Abb. 4: Bearbeitung der Start-Konfiguration eines Java-Programms

Im sich öffnenden Fenster wird oben der Reiter „(x)= Arguments“ angeklickt und unter „VM arguments:“ der Agent-Eintrag ergänzt. Danach wird das Programm zusammen mit dem Agent gestartet, wenn der zugehörige Run-Knopf gedrückt wird. Meist ist es für die Übersichtlichkeit sinnvoll den Namen der Configuration, wie in Abb. 5 gezeigt, anzupassen.

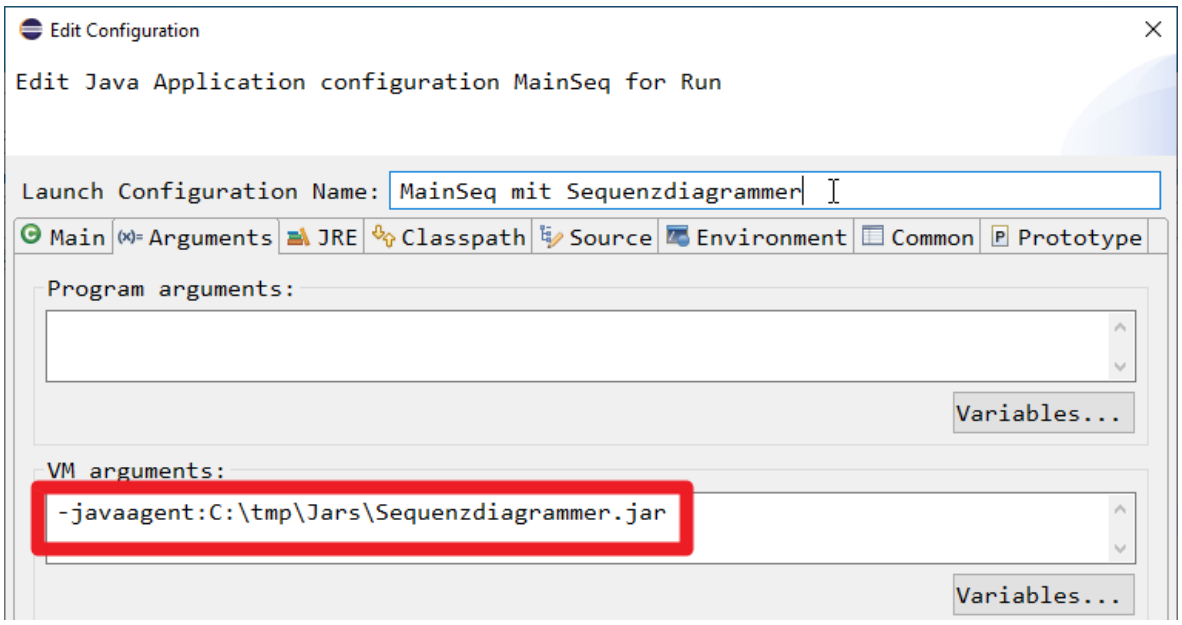

Abb. 5: Ergänzung des Agents zur Programmausführung

Das Fenster ist ebenfalls erreichbar, wenn nach einem Rechtsklick auf der ausführbaren Klasse „Run As“ und dann „Run Configurations“, wie in Abb. 6 gezeigt, angeklickt wird.

Abb. 6: Alternativer Weg zur Startkonfiguration

Nach dem Start des Programms öffnen sich die in Abb. 7 gezeigten zwei Fenster. Dabei dient das rechte Fenster ausschließlich zur Anzeige, es kann weder direkt bedient noch verschoben werden. Alle Aktionen werden vom linken Fenster aus gesteuert. In der Starteinstellung laufen alle Methodenaufrufe und Ergebnisse einzeln ab, der nächste Schritt wird durch einen Klick auf „einen Schritt weiter“ ausgeführt. Um ein Diagramm automatisch zu erzeugen, wird die Einzelschrittsteuerung über „Einzelschritte abschalten“ abgeschaltet und so das Steuerungsfenster ausgeblendet. Eine Ausnahme stellen Situationen dar, in denen das analysierte Programm auf Eingaben wartet. Hier hält die Sequenzdiagrammerzeugung natürlich auch an, um nach der Eingabe automatisch weiterzulaufen.

Die obere Zeile des Sequenzdiagramm-Fensters wiederholt die bereits genutzten Objekte und ist bei längeren Diagrammen hilfreich um die genutzten Objekte zu identifizieren.

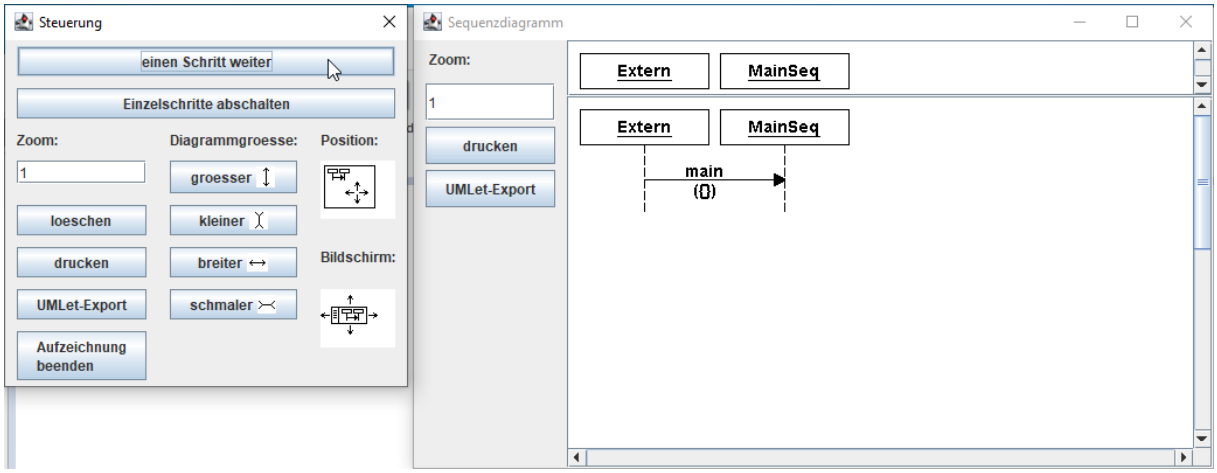

Abb. 7: Start der Sequenzdiagrammerzeugung

Das Steuerungsfenster bietet die weiteren folgenden Möglichkeiten an:

- Zoom: Die Größe des angezeigten Diagramms wird um den angegebenen Faktor geändert.
- loeschen: Das bisher erzeugte Diagramm wird gelöscht. Dies ist sinnvoll, wenn nur ein Ausschnitt eines Programms betrachtet werden soll. Es werden weiterhin nur vollständige Methodenaufrufe gezeigt, die Ergebnisse der zum Zeitpunkt des Löschens nicht beendeten Methoden werden nicht sichtbar.
- drucken: Das aktuelle Diagramm wird über den momentan eingestellten Systemdrucker ausgegeben. Dabei umfasst die vollständige Ausgabe nur eine Seite. Die Kombination kann zusammen mit einem PDF-Drucker sinnvoll sein, für genauere Betrachtungen und Bearbeitungen ist die nächste Möglichkeit meist sinnvoller.
- UMLet-Export: Das aktuell angezeigte Sequenzdiagramm wird in das uxf-Format exportiert, dass mit dem Programm UMLet (https://www.umlet.com/) gelesen und

bearbeitet werden kann. Die Datei wird im Ordner des Programmstarts erzeugt, der Name beginnt mit „Sequenzdiagramm“ und wird mit einem Zeitstempel ergänzt. Die Font-Größe in UMLet kann, wie in Abb. 8 angedeutet, geändert werden.

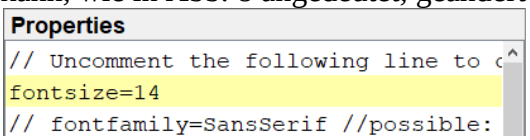

Abb. 8: Font-Größe für ein Diagramm in UMLet

- „Aufzeichnung beenden“: Das Steuerungsfenster verschwindet und nur noch das Sequenzdiagramm-Fenster kann mit seinen Scrollbalken und Knöpfen genutzt werden.
- Da das Sequenzdiagramm-Fenster nicht verändert werden kann, bieten die vier Knöpfe „groesser“, „kleiner“, „breiter“ und „schmaler“ die Möglichkeit die Fenstergröße schrittweise anzupassen.
- Position: Mit dem darunter angegebenen Drag-Board kann mit gedrückter linker Maustaste der angezeigte Ausschnitt aus dem Sequenzdiagramm verschoben werden. Es ist auch möglich die Maus über die Board-Grenzen hinaus zu verschieben. Durch die Nutzung des Scrollrades auf dem Drag-Board kann das Diagramm nach oben und nach unten verschoben werden.
- Bildschirm: Mit dem darunter angegebenen Drag-Board kann mit gedrückter linker Maustaste das Sequenzdiagramm-Fenster verschoben werden. Es ist auch möglich die Maus über die Board-Grenzen hinaus zu verschieben. Durch die Nutzung des Scrollrades auf dem Drag-Board ist der Zoom-Wert des Diagramms änderbar.

3 Möglichkeiten und Besonderheiten bei der Nutzung

3.1 Objekterzeugung

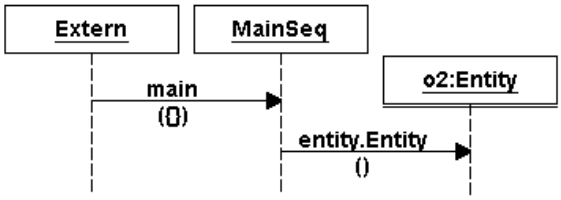

Abb. 9: Erstellung eines neuen Objekts

Wird im Sequencediagrammer ein neues Objekt erstellt, wird dieses zunächst mit einem in Abb. 9 erkennbaren Unterstrich bei o2 markiert, da das Werkzeug die Identität des Objekts noch nicht kennt. Das ist etwas irritierend, bedeutet aber nur, dass es am Anfang noch keinen this-Wert gibt. Der ist erst am Ende der Konstruktorausführung bekannt, dann wird auch der Strich entfernt.

Dies kann zwischenzeitlich zu irritierenden Ergebnissen führen, wenn das Objekt eine Referenz auf sich selbst weitergibt, die vom aufgerufenen Objekt genutzt wird. Das folgende Beispiel verdeutlicht dieses Szenario.

```
package entity;

public class Service {
	private Interessent interessent;

	public int anmelden(Interessent i) {
		this.interessent = i;
		return this.interessent.getVal();
	}
}


package entity;

public class Interessent {
	private int val = 42;

	public Interessent(Service s) {
		s.anmelden(this);
	}

	public int getVal() {
		return this.val;
	}
}
```

Das Hauptprogramm sieht wie folgt aus.

```
package main;
```

```
import entity.Interessent;
import entity.Service;

public class MainServiceInteressent {

	public static void main(String[] args) {
		Service s = new Service();
		new Interessent(s);
	}
}
```

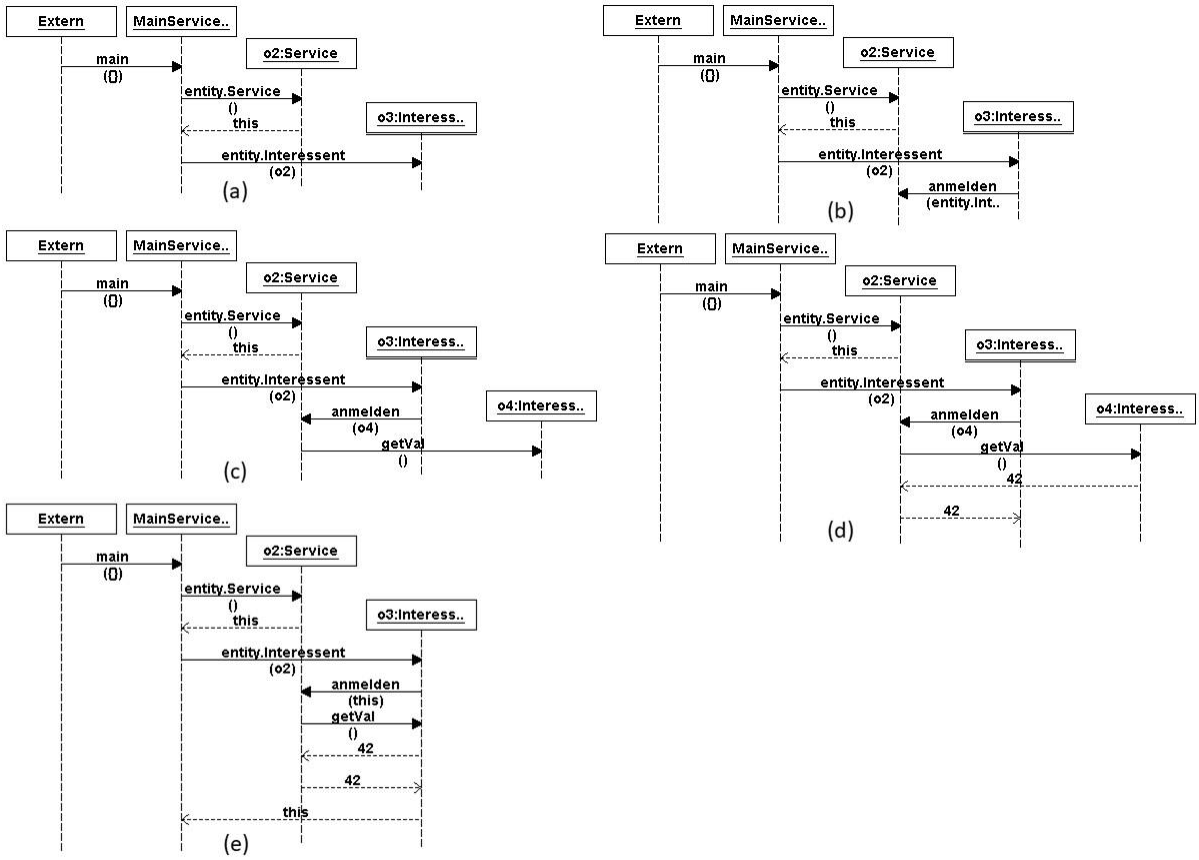

Abb. 10: Schrittweise Abarbeitung eines Konstruktoraufrufs

Abb. 10 zeigt die schrittweise Abarbeitung des Programms. In (a) wird ein neues Objekt unklarer Identität erzeugt, das in (b) Parameter der Methode anmelden() ist. Da das this-Objekt nicht bekannt ist, wird zur Darstellung nur die zugehörige toString()-Methode aufgerufen und nicht „this“ angegeben. In (c) wird dann bei dem übergebenen Objekt getVal() aufgerufen. Da dieses Objekt noch nicht bekannt ist, wird ein neues Objekt o4 angelegt und dabei erkannt, dass es sich um den Parameter von anmelden() handelt. In (d) werden die Ergebnisse der Methodenaufrufe zurückgegeben, wodurch der Konstruktor dann terminiert. Jetzt kann erkannt werden, dass o3 und o4 die gleichen Objekte sind, wodurch auch der Parameter von anmelden auf this gesetzt wird. Die folgende Ergebnisübermittlung von void an Extern ist im Diagramm in (e) verkürzend weggelassen.

3.2 Vererbung

Bei der Vererbung spielt wieder der Aufruf des Konstruktors eine Rolle. Zusätzlich zur im vorherigen Abschnitt beschriebenen Thematik ist zu beachten, dass beim Konstruktoraufruf einer erbenden Klasse zunächst immer ein Konstruktor der beerbten Klasse aufgerufen wird. Dies soll mit folgenden Klassen verdeutlicht werden.

```
package erb;

public class Ver1 {

	public int gib() {
		return 41;
	}
}
```

```
package erb;

public class Ver2 extends Ver1 {

	@Override
	public int gib() {
		return super.gib() + 1;
	}
}
```

```
package erb;

public class Ver3 extends Ver2 {

	@Override
	public int gib() {
		return super.gib() + 1;
	}
}
```

Das Hauptprogramm dazu sieht wie folgt aus.

```
package main;

import erb.Ver1;
import erb.Ver3;

public class Vererbung {

	public static void main(String[] args) {
		Ver1 v = new Ver3();
		v.gib();

	}

}
```

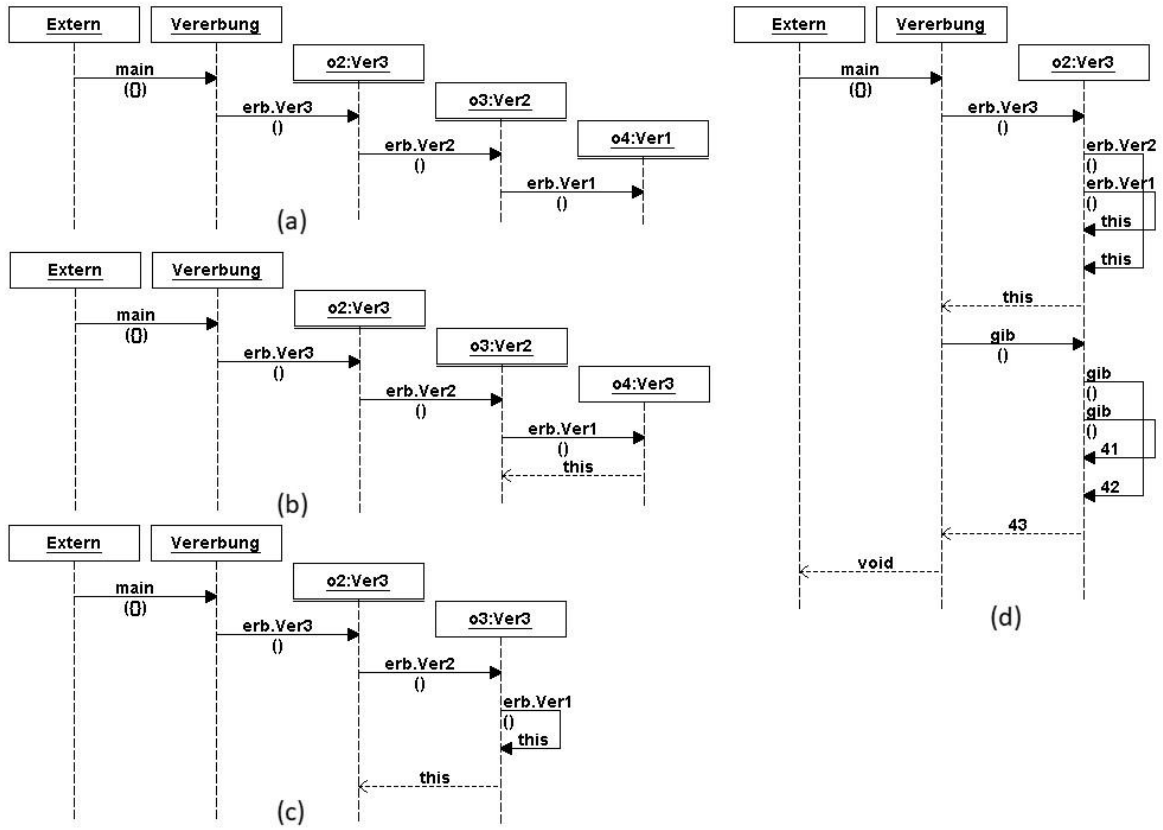

Abb. 11: Konstruktion eines erbenden Objekts

Abb. 11 zeigt links einen Ausschnitt aus dem Konstruktoraufruf und rechts in (d) das vollständige Diagramm. Da das zu konstruierende Objekt am Anfang unbekannt ist, gilt dies auch beim Konstruktoraufruf der beerbten Klasse, so dass temporär in (a) schrittweise drei Objekte angelegt werden. Nach Abschluss des Konstruktors in (b) wird jeweils erkannt, dass es sich um das gleiche Objekt handelt. Die Konstruktoren-Aufrufe sind deshalb im Ergebnis in (c) als ineinander geschachtelte lokale Aufrufe sichtbar. Die gleiche Darstellung wird für den Methodenaufruf genutzt, der jeweils mit „super“ an die beerbte Klasse weitergeleitet wird. Es ist allerdings nicht im Diagramm erkennbar, welche der gib()-Methoden genutzt wird. Eine vergleichbare Schachtelung objektlokaler Aufrufe ergibt sich auch bei der Darstellung von Rekursion.

In Sequenzdiagrammen werden nur „echte“ Objekte eingezeichnet. Dies bedeutet insbesondere, dass nur der Typ des Konstruktors und nicht einer Variablen eine Rolle spielt. Zwar können durch abstrakte Typen wie Interfaces die Anzahl zugreifbarer Methoden eingeschränkt werden, aber das dahinterliegende Objekt bleibt identisch. Dies ist in Abb. 11 an der Stelle zu sehen, an der das Objekt o2 der Klasse Ver3 erzeugt und einer Variablen vom Typ Ver1 zugewiesen wird.

3.3 Klassenmethoden

In den Beispielen zum UML-Standard wird bei Sequenzdiagrammen, anders als bei Klassendiagrammen, nicht explizit auf Klassenmethoden eingegangen. Wird programmiertechnisch ein Objekt zum Aufruf einer Klassenmethode genutzt, könnte der Aufruf in der üblichen Form als Aufruf beim Objekt eingezeichnet werden. Spätestens wenn so ein

Aufruf sauber programmiert direkt auf die Klasse zielt, gibt es dann aber ein Darstellungsproblem.

Die konsequente und im UML-Standard implizit enthaltene Lösung ist die Nutzung eines Klassenobjekts. Dabei hat jede Klasse genau ein Klassenobjekt, an das dann Klassenmethodenaufrufe gerichtet werden. Dieser Ansatz ist aus der Meta-Modellierung bekannt, die auch Grundlage der UML-Semantik ist und etwas erfahreneren Entwickler*innen aus der Nutzung von Reflection kennen. In Java ist das jeweilige Objekt zum Typen z. B. durch den Zugriff mit .class, also z. B. String.class oder int.class, zugänglich. Dieser Ansatz wird durch folgende Klassen veranschaulicht.

```
package entity;

public class MitKlassenmethode {
	private static int count = 1;
	private int val = 1;

	public static int nextCount() {
		return MitKlassenmethode.count++;
	}

	public int nextVal() {
		return this.val++;
	}
}
```

Das Hauptprogramm sieht wie folgt aus.

```
package main;

import entity.MitKlassenmethode;

public class MainKlassenmethode {

	public static void main(String[] args) {
		MitKlassenmethode.nextCount();
		MitKlassenmethode m1 = new MitKlassenmethode();
		MitKlassenmethode m2 = new MitKlassenmethode();
		m1.nextVal();
		m2.nextVal();
		m1.nextCount(); // schlechter Programmierstil
		m2.nextCount();
	}
}
```

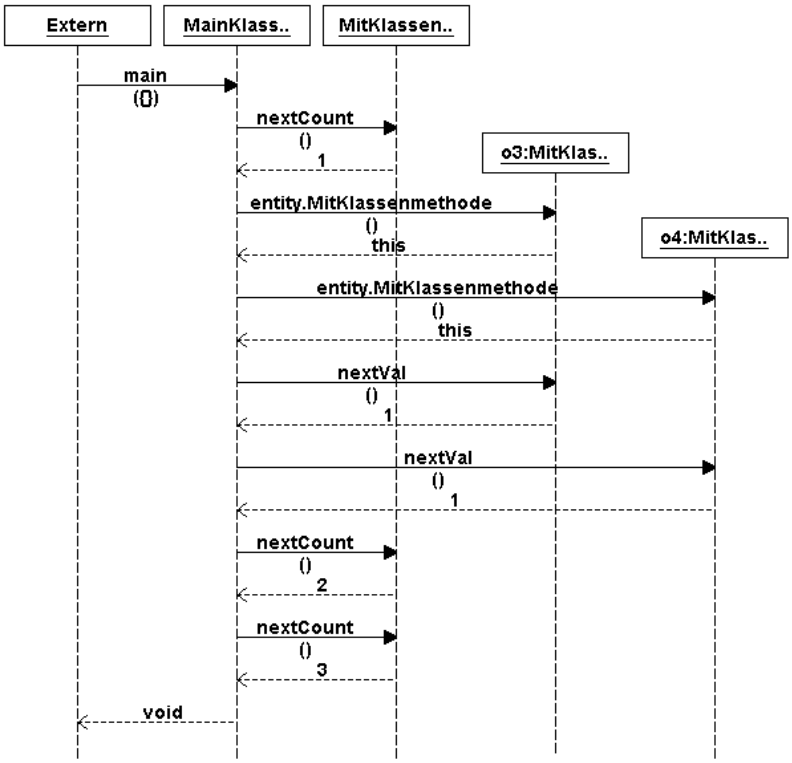

Abb. 12: Ausführung von Objekt- und Klassenmethoden

Die Abb. 12 zeigt das zugehörige Sequenzdiagramm. Klassenobjekte sind daran zu erkennen, dass im Kasten nur der Klassenname steht. Unabhängig davon, ob ein Objekt oder die Klasse zum Aufruf der Klassenmethode genutzt werden, verlaufen die Aufrufe im Sequenzdiagramm immer zum Klassenobjekt. Dies ist die sinnvolle Lösung, da die Aufrufe mit den Objekten nichts zu tun haben. Da Klassenobjekte nach dem Laden der Klasse existieren, existieren sie damit auch beim Start des Sequenzdiagramms und werden in der obersten Zeile eingetragen.

3.4 Ein- und Ausblenden von Klassen mit Effekten

Generell werden in der Programmierung sehr viele bereits existierende Klassen genutzt. Um die entstehenden Diagramme kompakter zu halten, wird eine große Menge von Klassen nicht im Sequencediagrammer angezeigt. Dies kann in einzelnen Situationen, gerade bei Programmieranfängern, die z. B. die Nutzung einer ArrayList beobachten wollen, unbefriedigend sein. Aus diesem Grund kann für das zu beobachtende Programm in dessen Startordner eine Konfigurationsdatei sdconfig.txt angelegt werden, mit denen die Auswahl der betrachten Klassen etwas anpassbar wird.

Klassen und Objekte von Klassen aus den Standard-Java-Modulen java.base, java.se, java.instrument, java.desktop, java.datatransfer und java.logging werden grundsätzlich unabhängig von Konfigurationsdatei nicht vom Sequencediagrammer angezeigt. Generell kann die Visualisierung anderer Klassen aus Java-Standardmodulen zu Problemen führen.

Weiterhin werden Klassen und Objekte von Klassen standardmäßig nicht angezeigt, wenn der vollqualifizierte Name, also der mit den Paketnamen davor, mit folgenden Buchstabenkombinationen beginnt: „java“, „bluej“, „nu“, „io“, „Difflib“, „nonapi“, „junit“, „sun“, „jdk“, „org“, „EDU“, „jersey“, „com“, „gsdet“, „ignore“. Sollen eigene Klassen nicht

angezeigt werden, kann man sie als ein möglicher Ansatz in einem Paket mit einem solchen Namen einordnen.

Mit der Konfigurationsdatei sdconfig.txt kann man Klassen hinzufügen oder unsichtbar schalten. Wichtig ist dabei, dass für nicht sichtbare Klassen alle eingehenden und alle ausgehenden Methodenaufrufe nicht eingezeichnet werden. In der Konfigurationsdatei werden vollqualifizierte Klassen- und Paketnamen eingetragen, dabei ist am Anfang und am Ende ein „*“ nutzbar, der für beliebig viele beliebige Zeichen steht. Ist das erste Zeichen ein „-“ (minus), sollen diese Klassen nicht sichtbar sein. Sichtbarkeitseinträge sind wichtiger als Nichtsichtbarkeitseinträge, gibt es einen passenden Sichtbarkeitseintrag, ist die Klasse sichtbar. Gleiches gilt, wenn kein Eintrag für einen Klassennamen passt, dann hängt die Sichtbarkeit ausschließlich von den genannten Standardeinstellungen ab. Durch ein „#“ als erstes Zeichen wird der Eintrag ignoriert, was auch zum Kommentieren genutzt werden kann. Als Trennzeichen zwischen Paket und Klassennamen wird „/“ genutzt.

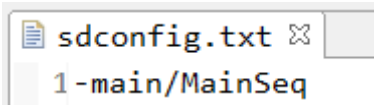

Abb. 13: Beispiel für eine Konfigurationsdatei

Wird im Beispiel aus Abb. 2 die Konfigurationsdatei aus Abb. 13 genutzt, wird alles, was mit der MainSeq-Klasse zusammenhängt ausgeblendet. Das Ergebnis ist in Abb. 14 dargestellt.

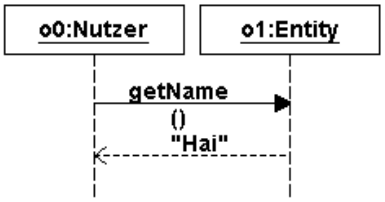

Abb. 14: Diagramm nach Ausblendung einer Klasse

Wird in der Konfigurationsdatei die Zeile

```
main/*
```

ergänzt, ist das Ergebnis wieder das ursprüngliche Diagramm. Durch das Ausblenden von Klassen kann es passieren, dass der Zusammenhang eines Sequenzdiagramms und damit der rote Faden aus der Abb. 3 verloren geht, wie es in Abb. 15 skizziert ist.

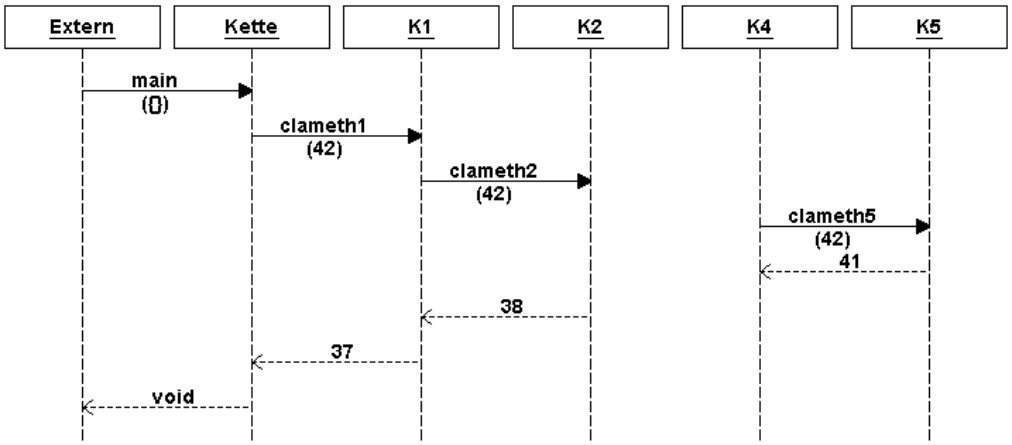

Abb. 15: nicht zusammenhängendes Sequenzdiagramm

Java bietet viele Möglichkeiten Klassen zu erzeugen. Die zugehörigen Klassen stehen im Sequenzdiagramm, für die vereinfachend zusammenfassend als Klassenname $<intern> genutzt wird, wobei es sich natürlich um unterschiedliche anonyme Klassen handelt.

Die Java-VM erzeugt zur Laufzeit weitere interne Methoden, um z. B. Lambda-Ausdrücke und Switch-Befehle über Enumerations auszuführen, diese werden ebenfalls angezeigt. Als Beispiel werden die folgende Enumeration und das folgende Interface genutzt.

```
package entity;

public enum Aufz {
	AA, BB, CC, DD
}
```

```
package business;

public interface Mini {
	public int mach();
}
```

Das zugehörige Hauptprogramm sieht wie folgt aus.

```
package main;

import business.Mini;
import entity.Aufz;

public class MainAnonym {

	public static void main(String[] args) {
		Mini m1 = new Mini() {
			@Override
			public int mach() {
				return 26;
			}
		};
		m1.mach();
		new Mini() {
			@Override
			public int mach() {
				return 2;
			}
		}.mach();
		Mini m3 = () -> 2000;
		m3.mach();
		Aufz az = Aufz.AA;
		switch(az) {
			case AA:
			case BB:
			case CC:
			case DD:
			default: m3.mach();
		}
	}
```

```
}
```

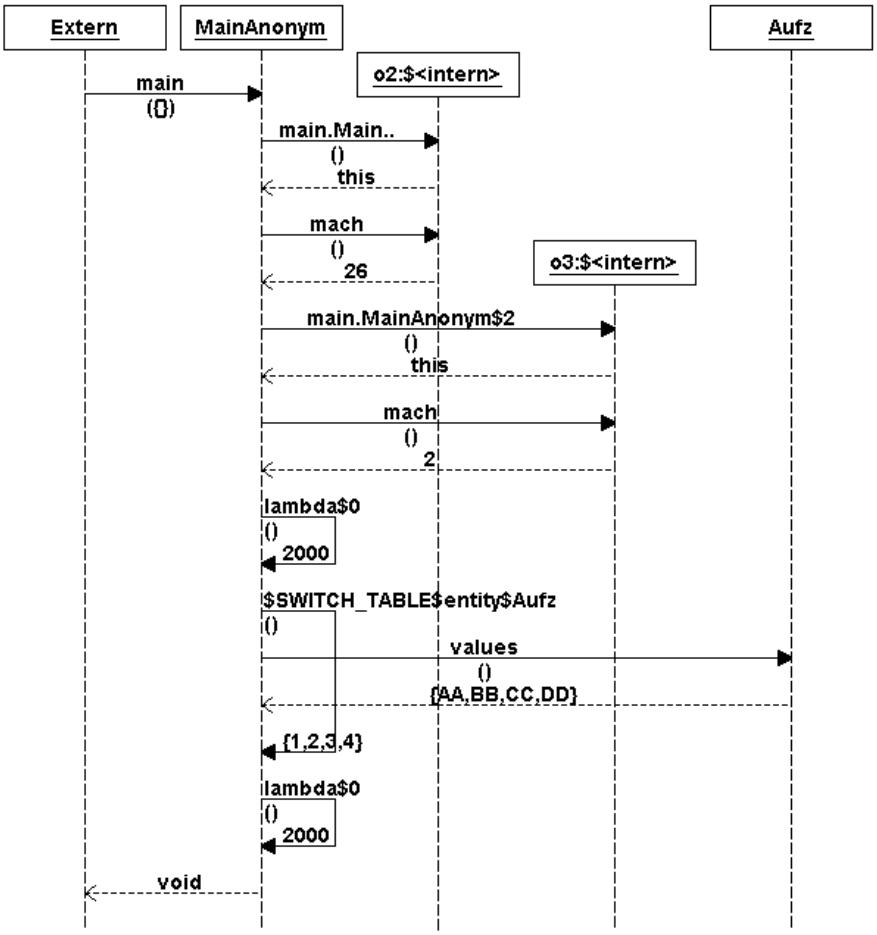

Abb. 16: Sequenzdiagramm mit anonymen Klassen und switch mit Enumeration

Abb. 16 zeigt das resultierende Sequenzdiagramm. Lambda-Ausdrücke, die ein Objekt erzeugen, werden zu einer internen Methode, bei Switch-Befehlen werden vor der Ausführung alle Werte der Enumeration geladen. Die eigentliche Ausführung des Befehls findet danach statt.

Ein Zugriff auf die Klasse java.util.ArrayList ist weiterhin nicht möglich, da sie im Modul java.base liegt. Soll trotzdem die Visualisierung einer solchen Klasse erfolgen, muss zu Workarounds gegriffen werden. Ein typischer Ansatz ist eine Klasse, die ein Objekt der fehlenden Klasse enthält und darum eine Fassade baut. Sie bietet damit jede Methode der fehlenden Klasse an, delegiert diese Aufrufe an die Methoden des Objekts und gibt dessen Ergebnisse zurück. Ergänzend kann eine Methode sinnvoll sein, die ein Objekt der fehlenden Klasse in ein Objekt der Workaround-Klasse verwandelt. Der Anfang einer solchen Klasse für die ArrayList-Klasse kann wie folgt aussehen, dabei wurden bewusst ein verwandter Paketname genutzt und die Implementierungen der Interfaces der Ursprungsklasse übernommen.

```
package temporary.java.util;
  public class ArrayList<E> implements java.util.List<E>
        ,java.util.RandomAccess
        ,java.lang.Cloneable
```

```
    ,java.io.Serializable{

  private java.util.ArrayList<E> var;

  public ArrayList(java.util.Collection<? extends E> v0) {
    this.var = new java.util.ArrayList<E>(v0);
  }

  public ArrayList() {
    this.var = new java.util.ArrayList<E>();
  }

  public int size() {
    return this.var.size();
  }

  …
```

Der Sequencediagrammer kann auch in der Kombination mit der Ausführung von JUnit-Tests genutzt werden. Der Agent ist wie gewohnt und in Abb. 5 gezeigt für die Ausführung als VM-argument zu ergänzen. Zur Visualisierung können dann Klassen von JUnit sichtbar geschaltet werden, wie es z. B. für die Klasse Assertions wie folgt möglich ist. Mit dem zweiten Eintrag können Exceptions sichtbar gemacht werden, die allerdings ohne Verbindung zum vorherigen Ablauf erzeugt werden.

```
org/junit/jupiter/api/Assertions

org/opentest4j/AssertionFailedError
```

Das Ergebnis wird mit folgendem Beispieltest gezeigt.

```
package main;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import business.Mini;

class MiniTest {

      @Test
      void test() {
            Mini m1 = new Mini() {
                  @Override
                  public int mach() {
                        return 25;
                  }
            };
            Assertions.assertEquals(26, m1.mach()); // scheitert
      }
}
```

Das resultierende Sequenzdiagramm ist in Abb. 17 dargestellt. Es ist erkennbar, dass von einer nicht sichtbaren Klasse des JUnit-Frameworks ein Objekt der Testklasse erstellt und am Ende die Überprüfung durchgeführt wird. Da JUnit die JVM am Ende terminiert, werden automatisch

alle Fenster geschlossen, zum Speichern des Diagramms muss so die vorletzte Aktion bekannt sein und dann gespeichert werden.

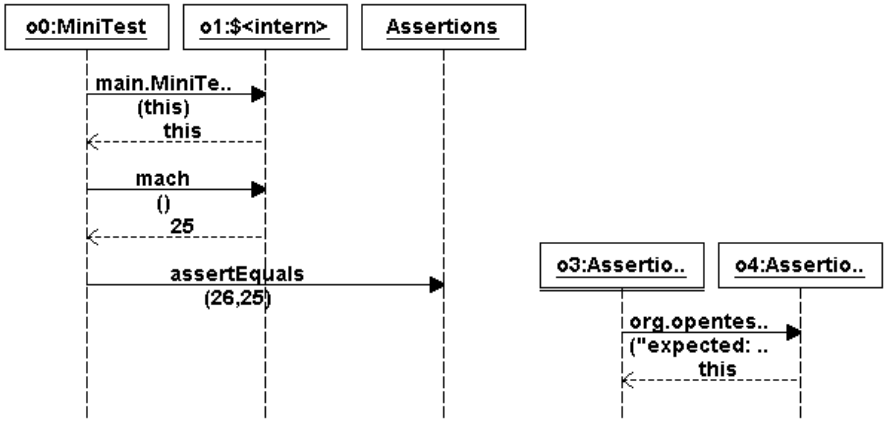

Abb. 17: Sequenzdiagramm zur Testausführung

Bei der Beobachtung von JUnit ist zu beachten, dass dieses Framework in jeder Entwicklungsumgebung individuell zur Darstellung der Ergebnisse eingebunden wird. Eclipse hat z. B. einen eigenen TestRunner.

3.5 Visualisierung von Exceptions

Mit Exceptions werden Methoden und Konstruktoren vor ihrem üblichen Ende terminiert. Der Sequencediagrammer stellt dieses als Methodenende dar, das mit "*Exception*" markiert ist. Dies wird mit folgendem Beispiel visualisiert, das verschiedene Varianten der Exception-Behandlung zeigt.

```
package entity;

public class Nicht13Exception extends IllegalArgumentException {

	public Nicht13Exception() {
		super("nich 13");
	}
}


package entity;

public class Adder {
	private int val;

	public Adder() {}

	public Adder(int x) {
		this.val = x;
		if (x < 0) {
			throw new IllegalArgumentException(
					"nicht so negativ");
		}
```

```
	}

	public int add(int w) {
		if (w == 7) {
			throw new IllegalArgumentException("nicht 7");
		}
		try {
			if (w == 13) {
				throw new Nicht13Exception();
			}
			if (w == 17) {
				throw new IllegalArgumentException(
						"nicht 17");
			}
			this.val = this.val + w;
		} catch (Exception e) {
			this.reset();
		}
		return this.val;
	}

	public void reset() {
		this.val = 0;
	}
}
```

Das Hauptprogramm sieht wie folgt aus.

```
package main;
import entity.Adder;

public class MainAdder {

	public static void main(String[] args) {
		Adder black = new Adder(42);
		try {
			black.add(42);
			black.add(17);
			black.add(13);
			black.add(7);
		} catch (Exception e) {
			black.reset();
		}
		try {
			black = new Adder(-42);
		} catch (Exception e) {
		}
		black.add(7); // wird nicht gefangen
	}

}
```

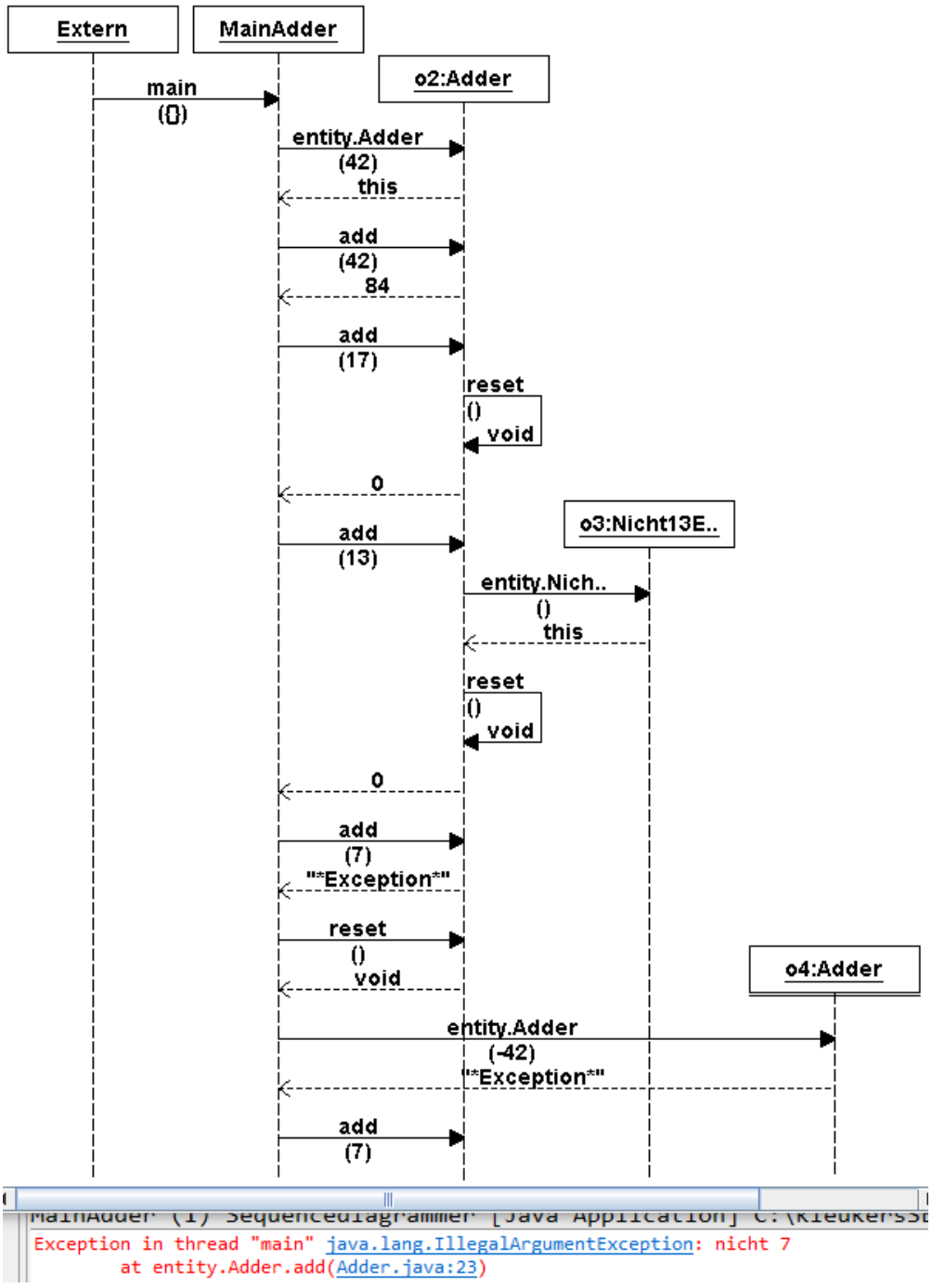

Abb. 18: Ausnahmebehandlung im Sequenzdiagramm

Das Diagramm in Abb. 18 zeigt das resultierende Ergebnis. Bei dem mit einer Exception beendeten Konstruktoraufruf wird eine Ungenauigkeit des Sequencediagrammer deutlich, dass diese Art von Objekten nicht sauber erkannt wird und mit dem zusätzlichen Unterstrich stehen bleibt. Sollten Vererbungen im Konstruktoraufruf stehen, würden die ebenfalls nicht zusammengeführt. Dies Problem sollte eher theoretischer Natur sein, da die weitere Verwendung eines Objekts, dessen Erzeugung mit einer Exception endet, ein fragwürdiger Programmierstil ist.

Beim letzten Methodenaufruf wird die Exception nicht gefangen und das Programm terminiert. Dies ist auch im Sequenzdiagramm erkennbar, da u. a. der finale mit „void“ beschriftete Pfeil zu Extern fehlt.

3.6 Kombination mit Debugger

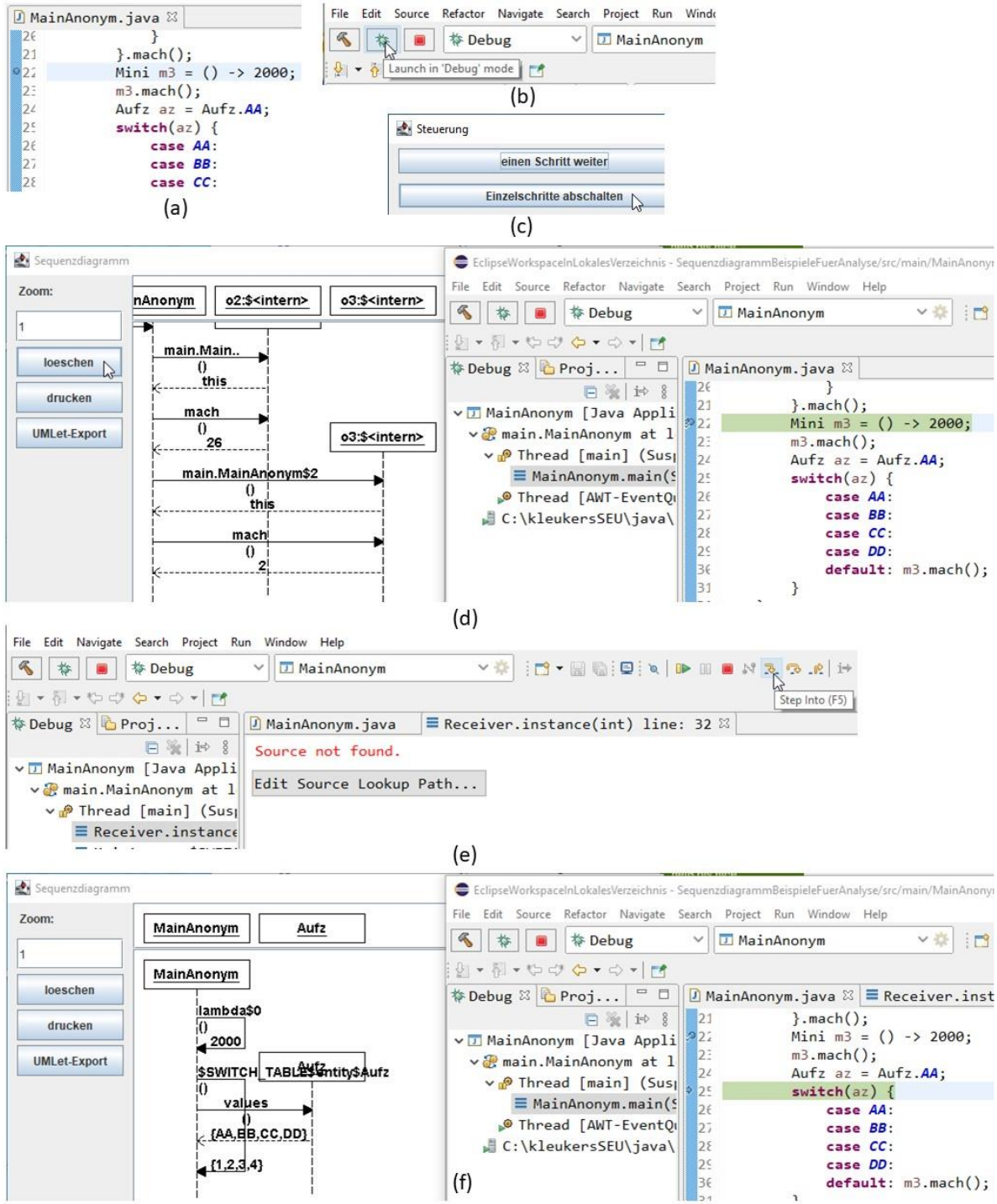

Abb. 19: Debugger mit Sequencediagrammer kombiniert

Der Sequencediagrammer an sich ist in seinem Verhalten verwandt mit einem Debugger. Er lässt sich recht einfach mit einem beliebigen Debugger kombinieren, wie es exemplarisch für Eclipse in Abb. 19 gezeigt wird. Im Beispiel besteht nur Interesse für ein Sequenzdiagramm ab der Zeile 22 des zur Abb. 16 gehörenden Programmcodes.

Zunächst wird in (a) der Break-Point, der Einstiegspunkt, gewählt, was in Eclipse mit einem Klick am Rand möglich ist. Danach wird das Programm über den Debugger in (b) gestartet. Da die gleiche Startkonfiguration wie beim normalen Ablauf genutzt wird, ist der Agent-Eintrag `-javaagent:C:\tmp\Jars\Sequencediagrammer.jar` hier automatisch einhalten. Generell können Debugger und Sequencediagrammer sofort parallel laufen, wobei dann häufiger neben dem nächsten Schritt im Debugger der nächste Einzelschritt im Sequencediagrammer ausgelöst werden muss. Um direkt zum Break-Point zu kommen ist es deshalb sinnvoll „Einzelschritte abschalten“ anzuklicken. Spätestens wenn der Break-Point erreicht wird, öffnet sich in (d) die Debugger-Ansicht. Da nur ein Interesse an einem Sequenzdiagramm ab diesem Punkt besteht, wird in (d) mit „loeschen“ das bisherige Diagramm gelöscht. Danach wird die übliche Debugger-Steuerung genutzt, um das Programm weiterlaufen zu lassen. Läuft das Programm ohne weitere Debug-Schritte, wird jetzt ein Diagramm ab diesen Punkt angelegt. Gibt es weitere Break-Points, kann so ein Sequenzdiagramm von einem Punkt bis zu einem weiteren Punkt angelegt werden. Wird die Einzelschritt-Steuerung des Debuggers genutzt, kann die schrittweise Erstellung des Sequenzdiagramms verfolgt werden. Wird die Step-Into-Möglichkeit genutzt kann es passieren, dass wie in (e) zu einem Receiver-Objekt gesprungen wird. Dies gehört zum Sequencediagrammer und ist zu ignorieren, also mit Step-Return oder Step-Out zu verlassen. In (f) wird ein Zwischenergebnis präsentiert, nachdem der erste Teil der Switch-Anweisung abgearbeitet ist.

3.7 Weitere Konfigurationsmöglichkeiten

In der Datei sdconfig.txt gibt es weitere Konfigurationsmöglichkeiten, die u. a. das Erscheinungsbild des Sequencediagrammer, aber auch sein Verhalten verändern können. Diese Einstellungen sind als experimentell anzusehen, da sie nicht in ihren Kombinationsmöglichkeiten einfach alle testbar sind. Da wesentliche Elemente des Erscheinungsbildes unverändert bleiben, wie die Kästchengröße, können die resultierenden Diagramme leicht unlesbar werden. Der plumpe Ratschlag ist, diese Einstellungen nicht zu nutzen und den Default-Einstellungen zu vertrauen.

Optionen werden typischerweise in der einfachen Form <Name_der_Option>=<Wert> angegeben, dabei stehen die Optionen aus Abb. 20 zur Verfügung.

Name_der_Option	Bedeutung	Standardwert
`XSIZE`	Startbreite des Fensters mit dem Sequenzdiagramm	800
`YSIZE`	Starthöhe des Fensters mit dem Sequenzdiagramm	605
`FONTNAME`	Name des Fonts, der zur Anzeige genutzt wird	SANS_SERIF
`FONTBOLD`	soll der Font fett angezeigt werden?	true

FONTSIZE	genutzte Font-Größe im Sequenzdiagramm	14
SHOWPRIVATE	sollen private-markierte Methoden eingezeichnet werden?	true
SHOWLAMBDA	sollen für Lambda-Ausdrücke erzeugte Objekte eingezeichnet werden (entspricht jeweils einer neuen Klassenmethode)?	true
SHOWSWITCH	soll ein switch über Enumerationswerte als eigene Methode (es werden mit values() alle möglichen Werte abgerufen) eingezeichnet werden?	true
SHOWEQUALSANDHASHCODE	sollen Aufrufe von equals() und hashCode, insofern die Methode von Object überschrieben wird, eingezeichnet werden? [hat eine gute Chance zum Absturz mit einer Endlosrekursion zu führen]	false
SHOWTOSTRING	sollen Aufrufe von Klassen, die toString() überschreiben, eingezeichnet werden? [hat eine gute Chance zum Absturz mit einer Endlosrekursion zu führen]	false
COMPLETETEXT	zeigt immer die vollständigen Klassen und Methodennamen sowie Parameter und Ergebnisse an	false
OBJEKTBREITE	die Breite mit der die Objekte in der oberen Zeile angezeigt werden	100
OBJEKTABSTAND	der Abstand zwischen den Objekten innerhalb der oberen Zeile	20
ZEILENABSTAND	der Abstand zwischen den einzelnen Pfeilen	32
(anderes Format:) -METHODE:<Methodenname>	alle Methoden mit genau dem Methodennamen <Methodenname> werden nicht angezeigt, d. h. auch Aufrufe die in der Abarbeitung dieser Methode stattfinden, sind nicht sichtbar.	

Abb. 20: Konfigurationsmöglichkeiten

Die Möglichkeiten werden mit folgenden Klassen gezeigt.

```
package entity;

public enum Aufzaehlung {
	BLUBB,BING,bumm,Bang
}
```

```java
package entity;

public interface Mini {
	public int mach();
}
```

```java
package entity;

import java.util.List;

public class Methodenvarianten {

	private int val;

	public Methodenvarianten(int val) {
		super();
		this.val = val;
	}

	public int eineMethode() {
	  return 42;
	}

	public int publicRuftPrivateAuf(int x) {
		return this.privateMethode(x) + this.val;
	}

	private int privateMethode(int x) {
		return x + 1;
	}

	public int mitLamdba() {
		var liste = List.of("a","b","c");
		liste.forEach(System.out::println);
		Mini m3 = () -> 2000;
		m3.mach();
		return liste.size();
	}

	public int mitSwitch(Aufzaehlung a) {
		switch(a) {
		case Bang: return 1;
		case BLUBB: return 2;
		case bumm: return 3;
		}
		return 0;
	}

	@Override
	public int hashCode() {
		return val;
```

```
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Methodenvarianten other = (Methodenvarianten) obj;
		if (val != other.val)
			return false;
		return true;
	}

  @Override
  public String toString() {
    return "Methodenvarianten [val=" + val + "]";
  }
}
```

Die Main-Methode sieht wie folgt aus.

```
package main;

import entity.Aufzaehlung;
import entity.Methodenvarianten;

public class MainOptionen {
	public static void main(String... s) {
		Methodenvarianten m1 = new Methodenvarianten(42);
		m1.eineMethode();
		m1.hashCode();
		m1.equals(m1);
		String tmp = m1.toString();
		System.out.println(m1);
		m1.publicRuftPrivateAuf(5);
		m1.mitLamdba();
		m1.mitSwitch(Aufzaehlung.BLUBB);
	}
}
```

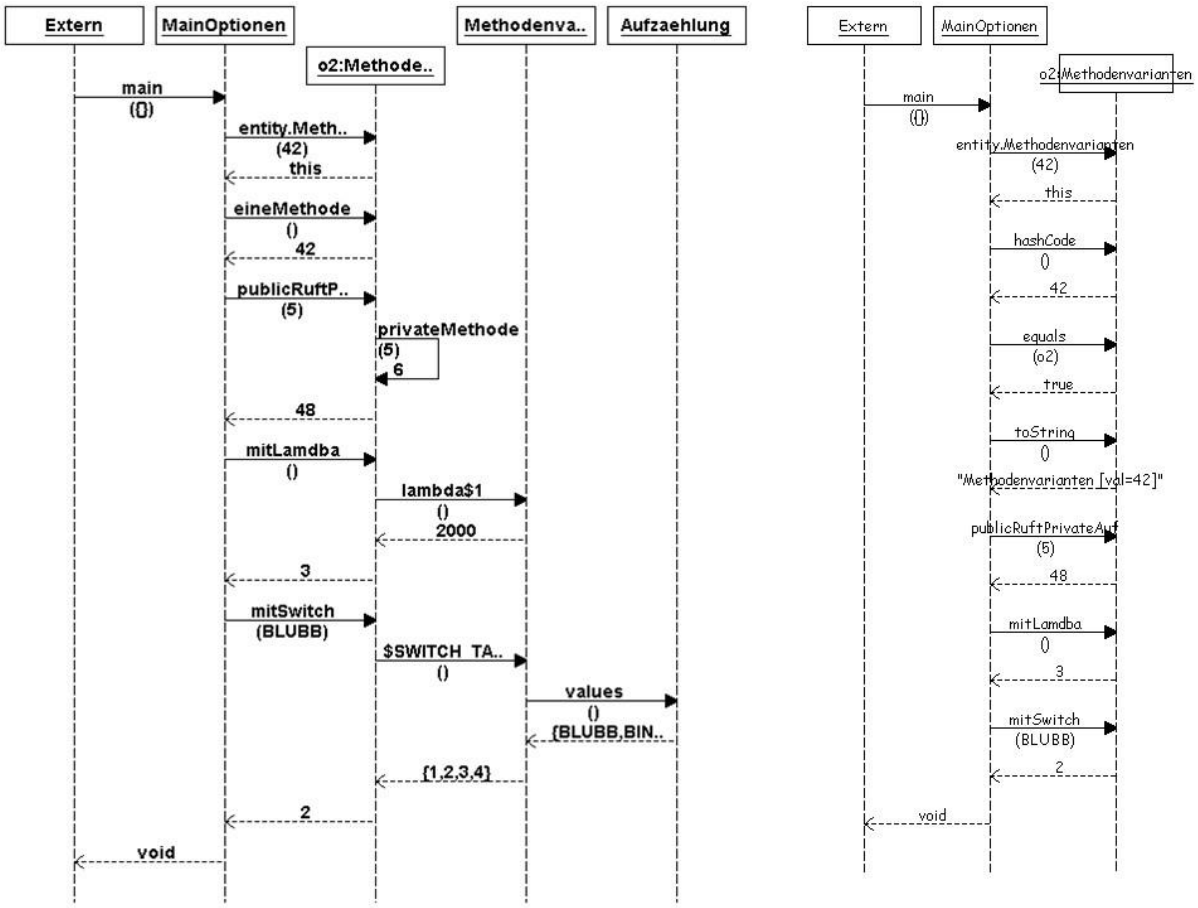

Abb. 21: Beispiel für Konfigurationsmöglichkeiten

Eine hier genutzte Beispielkonfigurationsdatei kann damit wie folgt aussehen:

```
#-main/MainOptionen
XSIZE=1200
YSIZE=700
FONTNAME=Comic Sans MS
FONTBOLD=false
FONTSIZE=12
SHOWPRIVATE=false
SHOWLAMBDA=false
SHOWTOSTRING=true
SHOWSWITCH=false
SHOWEQUALSANDHASHCODE=true
OBJEKTBREITE=80
OBJEKTABSTAND=5
ZEILENABSTAND=38
COMPLETETEXT=true
-METHODE:eineMethode
```

Abbildung Abb. 21 zeigt auf der linken Seite das Ergebnis mit den Standardeinstellungen, die rechte Seite das Ergebnis unter Nutzung der gezeigten Konfigurationsdatei, wobei die Größe angepasst wurde. Der implizite Aufruf der toString()-Methode im println() ist nicht sichtbar, da der Aufruf innerhalb einer Methode der nicht dargestellten Klasse PrintStream erfolgt.

4 Ansätze für die Neuentwicklung

Das vorgestellte Werkzeug ist ein Prototyp, der die Sprache Java bei Weitem nicht abdeckt und Fehler enthält. Es ist im Rahmen von Experimenten zur ByteCode-Bearbeitung erstellt und für exploratorische Prototypen einer Machbarkeitsstudie üblich mit dem Interesse „ob und wie etwas machbar ist“ und nicht mit einer vollständig systematischen Entwicklung entstanden. Dies bedeutet insbesondere, dass der Code nicht in Richtung Wart- und Erweiterbarkeit entstanden ist und so für weitere Funktionalitäten eine Neuimplementierung notwendig wird.

Wesentliche Erkenntnisse des Prototypen sind:

- für den gewählten Ansatz ist zu diskutieren, ob man mit den Einschränkungen leben kann, dass fast alle Klassen des JDK selbst nicht gemonitort werden
- vor einer Neuimplementierung ist eine intensive Auseinandersetzung mit dem Thema ClassLoader, wo werden welche Klassen verwaltet und welche Änderungen sind dann möglich, notwendig
- der genutzte Ansatz basiert auf einer ByteCode-Bearbeitung in einem Java-Agent, alternativ könnte die Debug-Schnittstelle der Java VM (Java Debugger Interface, JDI) in Betracht kommen, erste Schritte zeigten aber, dass u. a. die Dokumentationslage die Einarbeitung deutlich erschwert
- die Erkennung und Behandlung von Exceptions, u. a. in Konstruktoren und objektlokalen Methoden ist frühzeitig einzuplanen

Grober Ansatz des existierenden Systems:

Der Ansatz nutzt einen Java-Agent, der durch die Konfiguration vorgegebene Klassen beim ersten Laden anhand des Decorator-Patterns mit Hilfe einer ByteCode-Bearbeitung verändert. Jeweils am Anfang und Ende von Konstruktoren, Klassen- und Objektmethoden wird eine neue Beobachtungsklasse Receiver über diese aufgetretenen Ereignisse mit dem betroffenen Objekt, den Parametern und Ergebniswerten informiert. Ein Handicap ist, dass am Anfang des Konstruktors „this“ noch nicht bekannt ist. Diese Ereignisse werden auf einem Stapel verwaltet und können so einander zugeordnet werden. Exceptions werden nur indirekt daran erkennt, dass Methodenaufrufe oder -ergebnisse nicht den Erwartungen des Stapels entsprechen.

Verwandte Arbeiten:

Eine grobe Recherche hat gezeigt, dass einige Entwicklungsumgebungen Erweiterungen haben, mit denen aus dem Quellcode ein Sequenzdiagramm generiert wird. Dabei werden meist hierarchische Sequenzdiagramme genutzt, um die Ablaufalternativen zu zeigen. Weiterhin können Sequenzdiagramme auf Basis der existierenden Methoden manuell zusammengeklickt werden. Eine Generierung zur Laufzeit findet dabei nicht statt. Für die Generierung zur Laufzeit gibt es einen recht frühen Ansatz in JAVAVIS[1] (Achtung, es gibt mehrere Projekte mit diesem

[1] R. Oechsle, and T. Schmitt, “JAVAVIS: Automatic program visualization with object and sequence diagrams using the java debug interface (JDI)”, in Lecture Notes in Computer Science, vol. 2269: Software Visualization, pp. 176-190, Springer-Verlag, Berlin, Germany, 2002

Namen), der aber nicht mehr aktiv und über das Internet erhältlich ist. Die Eclipse-Erweiterung JIVE (https://cse.buffalo.edu/jive/) erlaubt die Generierung von Sequenzdiagrammen zur Laufzeit, sieht diese aber nur als ein Teil eines größeren Analysetools ohne schrittweise Interaktivität. Die generierten Sequenzdiagramme enthalten nicht alle Detailinformationen über Parameter, weiterhin werden alle erzeugten Objekte am Anfang des Diagramms gezeigt, was der UML wiederspricht. Mit MaintainJ (http://maintainj.com/) gibt es ein kommerzielles Analysewerkzeug, das Abläufe protokolliert und nachträglich ein Sequenzdiagramm, u. a. mit den Laufzeiten der Methoden generiert. Diese Ansatz zur Generierung nachträglich sichtbarer Sequenzdiagramme wird mit zusätzlichem Programmieraufwand mit AspectJ auch in https://github.com/MeenakshiParyani/SequenceDiagramGenerator genutzt, ebenfalls ein Einpersonenprojekt das seit 2017 nicht weiterentwickelt wird. Ein weiteres seit 2016 nicht erweitertes Projekt mit ähnlichem Ansatz ist https://sourceforge.net/projects/javacalltracer/. Der Sequencediagrammer erzeugt dagegen das Diagramm parallel zum laufenden Programm und ist wegen der u. a. für die Darstellung benötigten Rechenzeit nicht konzipiert Laufzeiten berechnen zu können. Im Bereich nebenläufiger Systeme gibt es Möglichkeiten Protokolle zur Laufzeit zu erstellen, wobei diese typischerweise erst danach betrachtet werden. Diese Protokollsysteme fokussieren dabei auf am Anfang vorgegebene Prozesse und berücksichtigen keine Objekterstellung und Vererbung.

Weitere interessante Aufgabenstellungen und Funktionalitäten:

- Die Verknüpfungsmöglichkeit mit einem Debugger wurde kurz bereits gezeigt. Das Werkzeug könnte selbst hier weitere Möglichkeiten, wie die Veränderung von Objekten, anbieten.
- Eine Möglichkeit zum flexiblen Ein- und Ausblenden von Objekten und Methodenaufrufen wäre sinnvoll. Eine Änderung der Sortierung und Umbenennung der Objekte könnte hilfreich sein.
- Ausgaben über System.out oder System.err werden im Sequenzdiagramm nicht angezeigt, was durch eine eigene kapselnde Klasse zwar ergänzt werden kann, aber optional direkt sichtbar werden sollte.
- Das aktuelle System unterstützt nur einen Thread, eine Erweiterung auf mehrere Threads sollte problemlos möglich sein, dabei müssen diese Threads visuell unterschieden werden.
- Die fehlenden Aktivitätsbalken, mit denen u. a. ineinander geschaltete Methodenaufrufe besser zu verfolgen sind, sind zu ergänzen.
- Eine Lösung mit der mehr, möglichst alle Methoden und Klassen visualisiert werden können, ist vorzuziehen.
- Die unvollständigen, genauer abgekürzten Informationen, sollten vollständig z. B. mit einem Mouse-Over-Effekt sichtbar gemacht werden.
- …