Eine textuelle Sprache zum automatischen Generieren von Sequenzdiagrammen

Daniel Jahn

Bachelorarbeit
eingereicht im Jahr 2015

Christian-Albrechts-Universität zu Kiel
Real-Time and Embedded Systems

Betreut durch: Christoph Daniel Schulze

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Sequenzdiagramme sind Diagramme aus der *Unified Modeling Language (UML)*, welche den Austausch von Nachrichten zwischen Objekten in einem System modellieren. Es gibt zwei Möglichkeiten, diese Diagramme zu erstellen: textuell oder mit einem *Drag & Drop*-Editor. Im Rahmen dieser Arbeit wird eine textuelle Sprache namens *KIELER Sequencediagram Language (KIESL)* entwickelt, die mit Hilfe eines Layoutalgorithmusses und den Technologien des *Kiel Integrated Environment for Layout Eclipse RichClient (KIELER)*-Projekts in Echtzeit visuell dargestellt werden kann. In einer kleinen Studie wird KIESL mit zwei anderen textuellen Sprachen und zwei Drag & Drop-Editoren verglichen, mit dem Ergebnis, dass sich KIESL gegen die anderen Editoren und Sprachen durchsetzen kann.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Problemstellung . . . 2
- 1.2 Verwandte Arbeiten . . . 3
  - 1.2.1 Papyrus . . . 3
  - 1.2.2 WebSequenceDiagrams . . . 4
  - 1.2.3 Quick Sequence Diagram Editor . . . 4
- 1.3 Aufbau der Arbeit . . . 5

2 Grundlagen **7**
- 2.1 Sequenzdiagramme . . . 7
- 2.2 Verwendete Technologien . . . 12
  - 2.2.1 KIELER . . . 12
  - 2.2.2 KGraph . . . 12
  - 2.2.3 KIML . . . 13
  - 2.2.4 Xtext . . . 14
  - 2.2.5 KLighD . . . 14

3 Textuelle Sprache **17**
- 3.1 Analyse bestehender Sprachen . . . 18
  - 3.1.1 WebSequenceDiagrams . . . 18
  - 3.1.2 Quick Sequence Diagram Editor . . . 19
- 3.2 Leitgedanken bei der Entwicklung von KIESL . . . 20
- 3.3 Realisierung von KIESL . . . 22
- 3.4 Editor . . . 23

4 Synthese **25**
- 4.1 Aufbau des KGraph . . . 25
- 4.2 KRendering . . . 28
- 4.3 Diagrammoptionen . . . 29

5 Evaluation **31**
- 5.1 Aufbau der Studie . . . 31
- 5.2 Ergebnisse der Studie . . . 32

6 Schlussfolgerung **37**
6.1 Zusammenfassung 37
6.2 Ausblick 37

A Anhang **39**

Bibliography **41**

Abbildungsverzeichnis

1.1 Ein einfaches Sequenzdiagramm 2
1.2 Der Papyrus Editor in Eclipse 3
1.3 Die Webseite von WebSequenceDiagrams 4
1.4 Der Quick Sequence Diagram Editor 5

2.1 Sequenzdiagramm mit zugehöriger KIESL Syntax 8
2.2 Aufbau eines KGraph 13
2.3 Schritte für die Visualisierung von Sequenzdiagrammen. 15

3.1 Beispiel eines Sequenzdiagramms 17
3.2 Falsche Platzierung der Antwortnachricht im Quick Sequence Diagram Editor 21

4.1 Der Aufbau des KGraphen in einem Sequenzdiagramm 26
4.2 Die Struktur des KGraphen 27
4.3 KRendering für eine Lebenslinie 28

5.1 Durchschnittliche Gesamtzeit für die Absolvierung der Studie 32
5.2 Durchschnittliche Bearbeitungszeit jeder Aufgabe aus der Studie 34
5.3 Durchschnittliche Bewertung der Editoren 34

A.1 Ein weiteres Sequenzdiagramm mit zugehöriger KIESL Syntax 40

Tabellenverzeichnis

2.1 Die verschiedenen Nachrichtentypen 10
2.2 Die verschiedenen Fragmenttypen 11

5.1 Die Fehler der Teilnehmer in den Editoren 33
5.2 Vor- und Nachteile der Editoren 35

A.1 Rohdaten der Studie 39

Abkürzungen

KIESL KIELER Sequencediagram Language

UML Unified Modeling Language

KIELER Kiel Integrated Environment for Layout Eclipse RichClient

KLighD KIELER Lightweight Diagrams

KIML KIELER Infrastructure for Meta Layout

KLay KIELER Layout Algorithms

Kapitel 1

Einleitung

Die *Unified Modeling Language (UML)* ist eine Modellierungssprache, welche hauptsächlich verschiedene grafische Darstellungen von Systemen spezifiziert. In der UML sind die Diagramme in Struktur- und Verhaltensdiagramme unterteilt. Die aktuelle UML-Version 2.5 vom Juni 2015 beinhaltet jeweils sieben Diagrammtypen in den beiden Gruppen. Je nachdem, welcher Aspekt eines System dargestellt werden soll, wählt man einen Diagrammtypen aus diesen Gruppen. Die hier behandelten Sequenzdiagramm gehören in die Gruppe der Verhaltensdiagramme.

Sequenzdiagramme beschreiben den Austausch von Nachrichten zwischen Objekten. In Abbildung 1.1 sieht man ein einfaches Sequenzdiagramm. Jedes Sequenzdiagramm hat einen Namen, der in der oberen linken Ecke des Diagramms zu sehen ist. Ebenfalls sind immer *Lebenslinien* enthalten, welche Klassen oder Instanzen von Klassen darstellen. An jeder Lebenslinie führt eine gestrichelte Linie vertikal nach unten und gibt den zeitlichen Ablauf der Nachrichten wieder. Die Nachrichten verlaufen in der Regel von Lebenslinie zu Lebenslinie.

Gregor Hoops hat 2013 in seiner Diplomarbeit [Hoo13] einen Algorithmus zum Layouten von Sequenzdiagrammen entwickelt[1] und in den UML-Editor *Papyrus* integriert. Papyrus ist ein Editor, der per *Drag & Drop* UML-Diagramme erstellen kann. Wenn man in Papyrus ein Sequenzdiagramm erstellt hat, kann man danach das Diagramm per Knopfdruck layouten lassen.

Das Erstellen von Diagrammen in Papyrus kann zum Teil sehr lange dauern, da oft nach den gewünschten Elementen, die in dem Diagramm platziert werden sollen, in der Auswahlleiste gesucht und dann für die Elemente manuell ein Platz gefunden werden muss. Danach findet meistens ein Wechsel von der Maus zur Tastatur statt, um dem neuen Element eine Beschriftung zu geben. Eine Alternative zu einem Drag & Drop-Editor wäre zum Beispiel eine textuelle Sprache. Mit einer solchen Sprache hat man einen entscheidenden Vorteil gegenüber einem graphischen Editor: Bemerkt man beispielsweise bei einem graphischen Editor im späteren Verlauf, dass ein Element hinzugefügt werden muss, dann muss in vielen Fällen erst Platz für das neue Element geschaffen werden. Bei einem textuellen Editor ist dieser Vorgang deutlich einfacher: Der Text wird an der gewünschten Stelle eingefügt und der nachfolgende Text verschiebt sich automatisch mit.

[1] Der Layoutalgorithmus ist aktuell Teil des KIELER Projekts, wird aber in Zukunft voraussichtlich Teil des Eclipse Layout Kernel-Projekts werden.

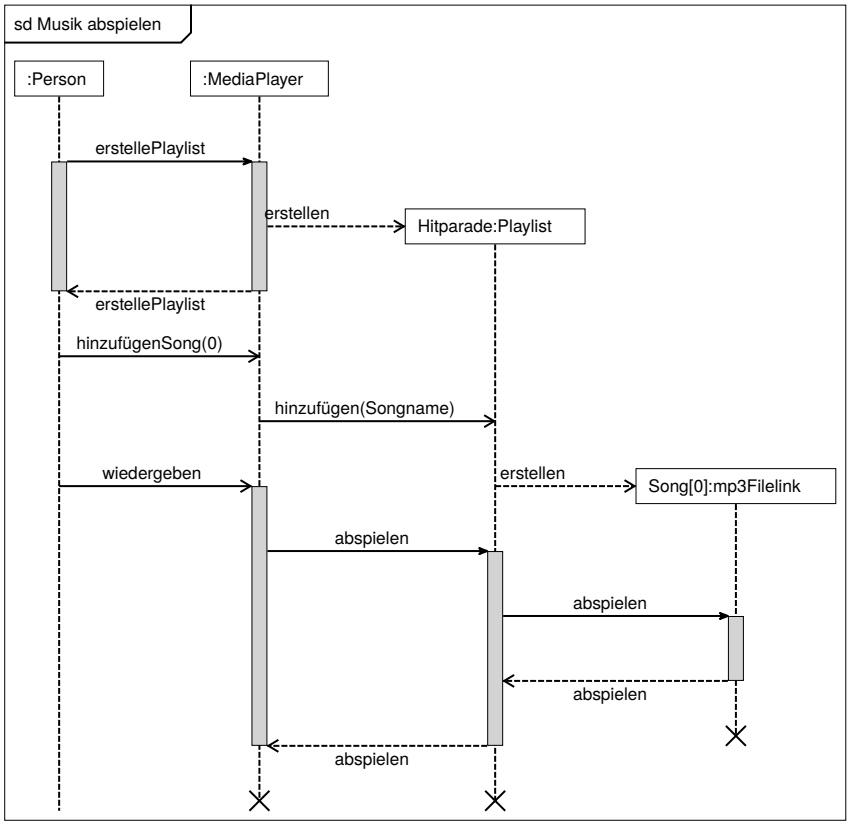


Abbildung 1.1. Ein einfaches Sequenzdiagramm in Anlehnung an [JRH+04]

Auch das Bearbeiten eines schon vorhandenen Objekts lässt sich direkt am Text vornehmen. Es gibt Editoren, die den neuen Graphen in Echtzeit anzeigen oder den Graphen per Knopfdruck erstellen. Bei einer textuellen Sprache hat man die Möglichkeit nicht ständig zwischen Maus und Tastatur zu wechseln. Das kommt dann auf die Fähigkeiten des Benutzers und die Funktionen des Editors drauf an. Bei der im Rahmen dieser Arbeit entwickelten Sprache, namens *KIELER Sequencediagram Language (KIESL)*, wird das Diagramm in Echtzeit erstellt und es gibt zusätzlich eine Autovervollständigung, die das Schreiben beschleunigt.

1.1 Problemstellung

Im Rahmen dieser Arbeit habe ich die textuelle Sprache KIESL entwickelt und Hoops' Algorithmus [Hoo13] in Kombination mit den in Abschnitt 2.2 vorgestellten Technologien zum Erstellen eines Diagramms genutzt. Meine Arbeit teilt sich dafür in zwei Abschnitte auf. Als Erstes muss die Grammatik von KIESL entworfen werden. Anhand von KIESL

wird dann ein Objektmodell generiert, um Diagramme im Speicher zu repräsentieren. Als Zweites muss eine Synthese geschrieben werden, welche Instanzen des Objektmodells in eine Form bringt, mit der der Layoutalgorithmus arbeiten kann. Das ist insofern ein Problem, da dieser Algorithmus nur speziell aufgebaute Graphen akzeptiert, die entsprechend erstellt werden müssen.

1.2 Verwandte Arbeiten

Als Teil des UML-Standards sind Sequenzdiagramme sehr bekannt und daher gibt es bereits einige Editoren, mit denen man Sequenzdiagramme erstellen kann. Im Folgenden stelle ich drei Editoren vor, welche bereits das Layouten von Sequenzdiagrammen beinhalten. Zwei dieser Editoren nutzen eine textuelle Sprache, welche ich in Abschnitt 3.1 umfangreicher mit KIESL vergleichen werde.

1.2.1 Papyrus

Papyrus[2] ist ein auf dem Anwendungs-Framework *Eclipse* basierender und quelloffener Editor. Mit Papyrus kann man dreizehn der vierzehn Diagrammtypen aus der UML darstellen. Für Sequenzdiagramme benötigt man den Layoutalgorithmus von Hoops [Hoo13], damit das Diagramm am Ende gelayoutet werden kann. Der Editor selbst wird per Drag & Drop bedient. Das heißt, dass die Elemente aus einer Auswahlleiste in das Diagramm mit Hilfe der Maus eingefügt und bearbeitet werden. Das Aussehen des Editors ist in Abbildung 1.2 abgebildet.

[2] `https://eclipse.org/papyrus/`

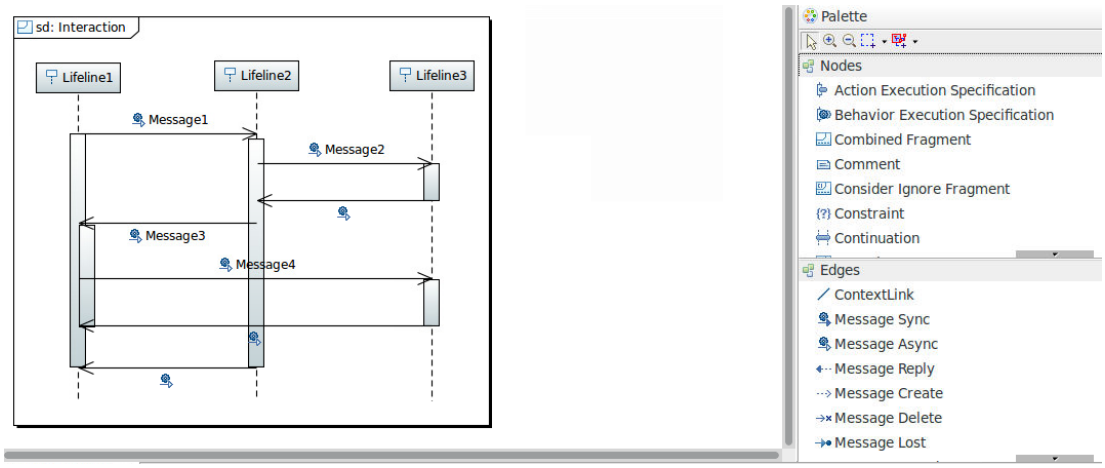


Abbildung 1.2. Der Papyrus Editor in Eclipse

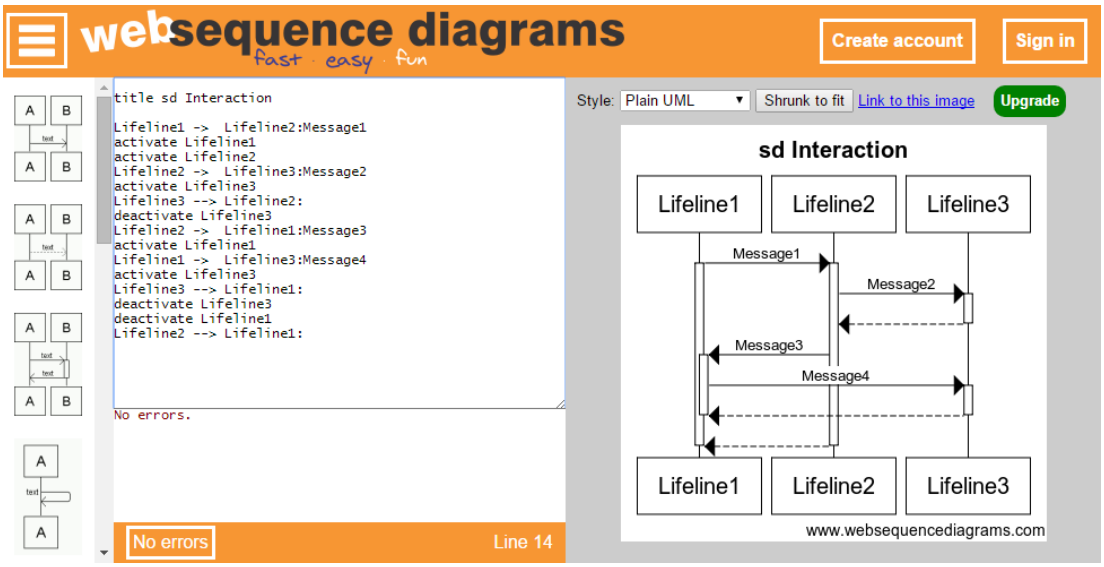


Abbildung 1.3. Die Webseite von WebSequenceDiagrams

1.2.2 WebSequenceDiagrams

Wie der Name bereits ausdrückt, ist *WebSequenceDiagrams*[3] eine Web-basierte Anwendung, in der man Sequenzdiagramme mit einer textuellen Sprache erstellen kann. Im linken Feld gibt man dafür den Text ein, woraufhin das Diagramm automatisch im rechten Feld der Seite angezeigt wird. Ein Vorteil dieser Seite ist also, dass man nur einen Browser braucht und dann direkt über die Seite Diagramme erstellen kann. Weiterhin werden an der linken Seite vorgefertigte Beispiele angezeigt, die man anklicken kann und die daraufhin als Text eingefügt werden und somit auch im generierten Diagramm auftauchen. In den Beispielen sind allerdings nicht alle Funktionen der Sprache enthalten. Die restlichen Funktionen findet man im Menü unter *getting started*. Das führt dazu, dass man manche Funktionen erst später kennenlernt, da man vermuten könnte, dass in den Beispielen auf der Startseite alle Funktionen vorgestellt werden. Ein weiterer Nachteil ist, dass manche Funktionen einen Premiumaccount benötigen und andere Funktionen aus der UML gar nicht unterstützt werden. Die Sprache selbst ist sehr leicht zu verstehen, bietet aber im Gegensatz zu KIESL keine Autovervollständigung. Das Aussehen der Webseite ist in Abbildung 1.3 abgebildet.

1.2.3 Quick Sequence Diagram Editor

Der *Quick Sequence Diagram Editor*[4] ist eine Anwendung zum Herunterladen. Nach dem Ausführen startet sich ein Editor, bei dem man im unteren Eingabefeld die textuelle

[3] `https://www.websequencediagrams.com/`

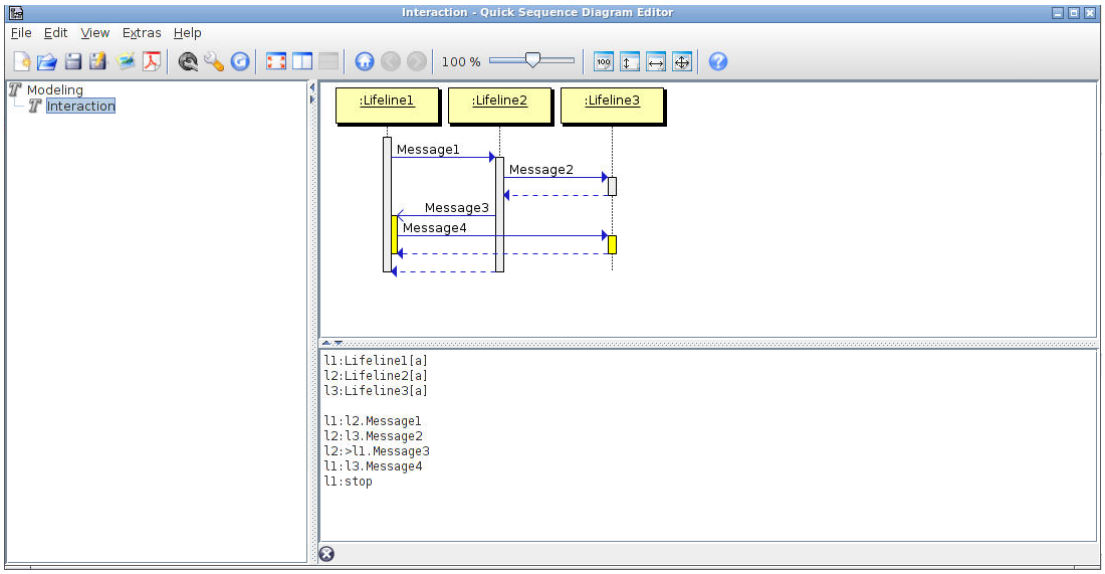


Abbildung 1.4. Der Quick Sequence Diagram Editor

Beschreibung eingibt und im oberen Feld das zugehörige Diagramm erscheint. In diese Sprache findet man keinen sehr guten Einstieg. Es gibt zwar drei Beispiele auf der Webseite, aber ohne genaueres Einlesen in die Dokumentation ist die Syntax nicht sofort verständlich. Weiterhin gibt es zwar eine simple Autovervollständigung beim Schreiben, aber diese kann nur bereits erstellte Objekte vervollständigen und leitet einen nicht mit an der aktuellen Stelle möglichen Vorschlägen durch die Sprache weiter. Das Aussehen des Editors ist in Abbildung 1.4 abgebildet.

1.3 Aufbau der Arbeit

Zunächst erläutere ich in Kapitel 2, was ein Sequenzdiagramm ist und aus welchen Bestandteilen es sich zusammensetzen kann. Weiterhin stelle ich die verwendeten Technologien dieser Arbeit vor. In Kapitel 3 stelle ich KIESL näher vor und vergleiche die Sprache mit der textuellen Sprache von WebSequenceDiagrams und von dem Quick Sequence Diagram Editor. Ebenfalls erkläre ich meine Herangehensweise, die mich zu meiner Sprache geführt hat. Als Nächstes erkläre ich in Kapitel 4, welche Schritte notwendig sind, um das Modell, welches durch KIESL erstellt wird, layouten zu können. Danach zeige ich in Kapitel 5 die Ergebnisse einer kleinen Studie, in der ich KIESL mit vier anderen Editoren verglichen habe. Abschließend gebe ich in Kapitel 6 eine Zusammenfassung über meine Arbeit und stelle Themen vor, die man anknüpfend an diese Arbeit weiter verfolgen kann.

[4] `http://sdedit.sourceforge.net/`

Kapitel 2

Grundlagen

In diesem Kapitel gebe ich einen Überblick über die zum Verständnis nötigen Grundlagen meiner Arbeit. Ich erkläre zunächst den Aufbau von Sequenzdiagrammen und im Anschluss welche Technologien ich verwendet habe.

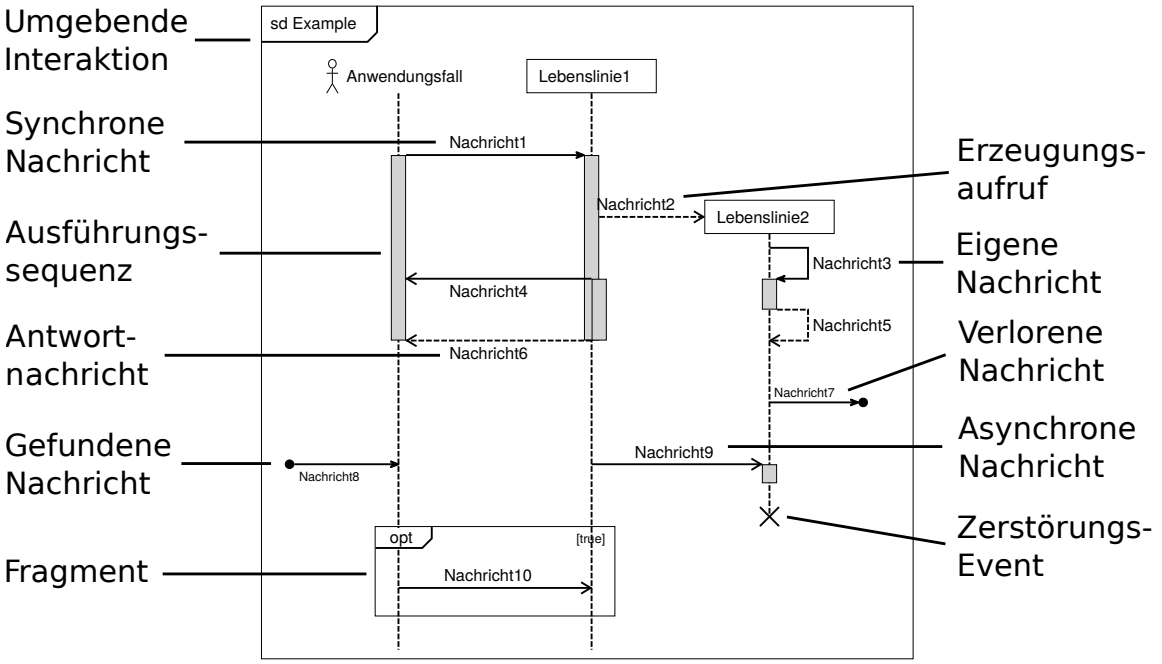
2.1 Sequenzdiagramme

Ein Sequenzdiagramm besteht aus verschiedenen Arten von Elementen und es visualisiert im Wesentlichen den Austausch von Nachrichten zwischen Objekten eines Systems. Was genau das bedeutet und aus welchen Elementen ein Sequenzdiagramm bestehen kann [JRH+04], wird im Folgenden genauer erklärt und in Abbildung 2.1 mit zugehöriger KIESL-Syntax dargestellt.

1. Die *Umgebende Interaktion (engl. Surrounding Interaction)* ist ein Kasten, welcher um jedes Sequenzdiagramm gezeichnet ist und den Namen des Diagramms in der oberen linken Ecke beinhaltet.

2. Eine *Lebenslinie (engl. Lifeline)* besteht aus einem Namen, welcher von einem Kasten umschlossen ist und einer gestrichelten Linie darunter, welche als Zeitachse fungiert. Eine Lebenslinie erstreckt sich typischerweise über die gesamte Höhe der umgebenden Interaktion und die Namen der Lebenslinien sind in der Regel oben im Diagramm nebeneinander angeordnet. Bei der objektorientierten Programmierung bestehen die Programme aus Klassen und Instanzen, welche Objekte einer solchen Klasse sind. Eine Lebenslinie repräsentiert eine Klasse oder eine Instanz einer Klasse. Wenn der Name der Lebenslinie mit einem Doppelpunkt beginnt, ist die Lebenslinie als Instanz der Klasse zu verstehen, wohingegen sie ohne einen Doppelpunkt als die Klasse selbst zu verstehen ist.

3. Eine *Nachricht (engl. Message)* tauscht Informationen zwischen Lebenslinien oder auf einer einzelnen Lebenslinie aus. Programme bestehen aus Methoden, welche das Verhalten von Objekten beschreiben und Nachrichten repräsentieren Methodenaufrufe im Diagramm. Einer Nachricht wird einer von mehreren Typen zugewiesen, die in Tabelle 2.1 näher erläutert werden. Die einfachste Nachricht beinhaltet dabei nur den Namen der aufgerufenen Methode. Man beachte dabei, dass der Name im Gegensatz



(a) Veranschaulichung der möglichen Elemente in einem Sequenzdiagramm. Das Diagramm wurde mit KIESL erstellt.

```
sequenceDiagram "Example"

usecase "Anwendungsfall" as u
lifeline "Lebenslinie1" as l1
lifeline "Lebenslinie2" as l2

u sync "Nachricht1" to l1
        sourceStartExec
        targetStartExec
l1 create "Nachricht2" to l2
l1 async "Nachricht4" to u
        sourceStartExec
l2 sync "Nachricht3" to l2
        targetStartExec
l1 response "Nachricht6" to u
        sourceEndExec all
        targetEndExec
l2 response "Nachricht5" to l2
        sourceEndExec
l2 async lost "Nachricht7"
u sync found "Nachricht8"
l1 async "Nachricht9" to l2
        targetStartExec
fragment "opt" {
        label "true"
        u async "Nachricht10" to l1
}
l2 destroy
```

(b) Textuelle Repräsentation des Sequenzdiagramms

Abbildung 2.1. Sequenzdiagramm mit zugehöriger KIESL Syntax

zu einem Methodenaufruf ohne Klammern geschrieben wird. Der Nachricht kann aber auch beliebig viele Argumente mitgegeben werden, welche in Klammern notiert und jeweils mit einem Komma voneinander abgetrennt werden.

4. Eine *Ausführungssequenz (engl. Sequence Execution)* wird beim Senden oder Empfangen von Nachrichten gestartet. Sie stellt die Ausführung einer Methode bis zu ihrem Abschluss dar und kann auch geschachtelt sein. Das Nutzen von Ausführungssequenzen ist optional. Eine Ausführungssequenz wird bei einer synchronen Nachricht durch den Erhalt einer Antwortnachricht und bei einer asynchronen Nachricht durch das Abschließen der Methode beendet. Sie kann also vor dem Verschicken oder Erhalten einer neuen Nachricht beendet sein.

5. Das *Zerstörungs-Event (engl. Destruction-Event)* signalisiert, dass die Lebenslinie an dieser Stelle beendet wird. Es können dann keine Nachrichten, die mit dieser Lebenslinie in Verbindung stehen, mehr gesendet oder empfangen werden.

6. Ein *Kombiniertes Fragment (engl. Combined Fragment)* ist ein Kasten, der eine oder mehrere Nachrichten umschließt. Ein solches Fragment erfüllt je nach Gebrauch verschiedene Aufgaben, welche in Tabelle 2.2 erläutert werden. Manche Fragmente können auch aus mehreren Regionen bestehen, welche durch eine gestrichelte Linie voneinander abgetrennt sind und zum Beispiel mehrere alternative Auswahlmöglichkeiten repräsentieren.

7. Ein *Anwendungsfall (engl. Use-Case)* stellt das Verhalten eines Systems aus der Sicht eines Nutzers dar. In der UML ist es möglich, typenübergreifende Diagramme zu erstellen. Ein Anwendungsfall ist ein anderer Diagrammtyp als ein Sequenzdiagramm. Es kann also alternativ zu einer Lebenslinie auch ein Anwendungsfall erstellt werden.

8. Ein *Kommentar (engl. Note)* kann an den Start- oder Endpunkt einer Nachricht oder an eine Lebenslinie angefügt werden und stellt Notizen innerhalb des Diagramms dar.

9. Eine *Zustandsinvariante (engl. State Invariant)* stellt eine Bedingung dar. Der Ablauf des Sequenzdiagramms kann nur weitergeführt werden, wenn diese Bedingung erfüllt ist. Die Bedingung kann an die Lebenslinie oder eine Nachricht angeheftet sein.

10. Ein *Verknüpfungspunkt (engl. Gate)* ist eine Nachricht, welche die Kommunikation zwischen referenzierten Interaktionen oder kombinierten Fragmenten ermöglicht. Die Nachricht zeigt zum Beispiel auf das Referenzfragment und im referenzierten Sequenzdiagramm beginnt die Nachricht auf dem Rand der umgebenden Interaktion.

11. *Lokale Variablen (engl. Local Variables)* können in der umgebenden Interaktion oder in kombinierten Fragmenten angegeben werden und sind nur im entsprechenden Bereich und deren Unterbereichen gültig.

Tabelle 2.1. Die verschiedenen Nachrichtentypen

Nachrichtentyp	Beschreibung
Asynchron	Der Sender der Nachricht kann nach dem Verschicken der Nachricht bereits weiterarbeiten und muss nicht auf eine Antwort des Empfängers warten.
Synchron	Der Sender der Nachricht muss auf die Fertigstellung des Empfängers warten. Dieser sendet eine Antwortnachricht und erst dann kann der Sender seine Ausführung fortsetzen. Der Sender ist also nach Abschicken einer synchronen Nachricht blockiert.
Antwort (*engl. response*)	Wie eben beschrieben wird bei einer synchronen Nachricht auf eine Antwort gewartet. Diese enthält typischerweise das Ergebnis des Methodenaufrufs.
Erzeugungsaufruf (*engl. create*)	Erstellt eine neue Lebenslinie. Die Nachricht zeigt auf den Namen der Lebenslinie und nicht auf ihre Zeitachse. Das bedeutet insbesondere, dass der Name der Lebenslinie nicht oben im Diagramm angezeigt wird, sondern dass die Lebenslinie vertikal nach unten verschoben ist.
Eigene Nachricht (*engl. self*)	Bei dieser Nachricht ist der Empfänger derselbe wie der Sender.
Verlorene Nachricht (*engl. lost*)	Eine Nachricht bei der der Empfänger nicht bekannt ist oder als nicht wichtig empfunden wird. Die Nachricht zeigt nicht auf eine zweite Lebenslinie, sondern auf einen Punkt. Zum Beispiel kann man sich hier ein Framework vorstellen. Während der Entwicklung ist dabei möglicherweise noch nicht spezifiziert, wer dieses Framework überhaupt nutzen wird.
Gefundene Nachricht (*engl. found*)	Bei dieser Nachricht ist der Sender nicht bekannt. Er wird ebenfalls durch einen Punkt im Diagramm repräsentiert. Hierbei kann man einen Anwendungsfall aus dem Alltag als Beispiel nennen: Wenn man zu Hause draußen Lärm hört, schließt man das Fenster. Dabei ist der Verursacher des Lärms unwichtig.

12. Mit einer *Ordnungsbeziehung (engl. General Ordering)* wird eine Reihenfolge für Nachrichten festgelegt. Damit ist sichergestellt, welche Nachricht zuerst ausgeführt werden muss, falls dies zum Beispiel durch Fragmente schwierig oder gar nicht darstellbar ist.

13. Mit *Zeitangaben (engl. Time Designations)* kann einer Nachricht explizit gesagt werden, wann sie beginnen und enden soll. Das führt dazu, dass sich asynchrone Nachrichten auch überkreuzen können und somit nicht mehr horizontal gezeichnet werden.

14. Eine *zeitliche Beschränkung (engl. Duration Constraint)* kann einerseits an einer Nachricht alleine angegeben werden und andererseits zwischen zwei Nachrichten, um anzugeben, wie viel Zeit die Nachrichten in Anspruch nehmen.

15. Eine *Koregion (engl. Coregion)* ist eine alternative Darstellung zu einem parallelen Fragment. Auf der Lebenslinie wird ein Bereich in eckigen Klammern abgetrennt. Alle Nachrichten innerhalb dieses Bereichs können in beliebiger Reihenfolge abgearbeitet werden.

Eine kurze Umfrage unter Nutzern von Sequenzdiagrammen ergab, dass die Elemente 1 bis 11 oft genutzte Elemente sind. Von diesen Elementen unterstützt der Layoutalgorithmus von Hoops [Hoo13] bisher lediglich die Elemente 1 bis 7.

Tabelle 2.2. Die verschiedenen Fragmenttypen

Fragmenttyp	Abkürzung	Wird genutzt für…
Alternativ	alt	alternative Ablaufmöglichkeiten
Optional	opt	optionale Interaktionsteile
Abbruchfragment	break	Interaktionen in Ausnahmefällen
Negation	neg	ungültige Interaktionen
Schleife	loop	mehrfach ausgeführte Interaktionen
Parallel	par	nebenläufige Interaktionen
Lose Ordnung	seq	abhängige chronologische Abläufe
Strenge Ordnung	strict	unabhängige chronologische Abläufe
Kritischer Bereich	critical	atomare Interaktionen
Irrelevante Nachrichten	ignore	Filter unwichtiger Nachrichten
Relevante Nachrichten	consider	Filter wichtiger Nachrichten
Sicherstellung	assert	unabdingbare Interaktionen
Interaktionsreferenz	ref	andere oder ausgelagerte Interaktionen

2.2 Verwendete Technologien

Um die weiteren Kapitel verstehen zu können, benötigt man zunächst ein gewisses Grundverständnis über die in meiner Arbeit genutzten Technologien, die ich nun vorstellen werde.

2.2.1 KIELER

Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) ist ein Projekt, welches als eine akademische Experimentierplattform genutzt wird. Das komplette Projekt basiert dabei auf dem Anwendungs-Framework Eclipse und wird dort über mehrere Plugins integriert. Ein Plugin erweitert dabei den Umfang von Eclipse.

KIELER besteht aus zwei großen Themengebieten. Das eine Gebiet ist die *Pragmatik*, welche die Produktivität von Entwicklern beim graphische Modellieren verbessern soll. Das Ziel hierbei ist, ein dynamisches und schnelles Arbeiten mit Diagrammen zu ermöglichen. Das andere Gebiet ist die *Semantik*, welches sich vor Allem um synchrone Sprachen und den dazu entworfenen *Sequentially Constructive Charts (SCCharts)* [HDM+13] kümmert. Nebenbei produziert die Semantik auch Anwendungsbeispiele für die Pragmatik.

Das Thema dieser Arbeit ist dem Gebiet der Pragmatik zuzuordnen.

2.2.2 KGraph

Der *KGraph* ist eine Datenstruktur zur Repräsentation von Graphen. In Abbildung 2.2 sieht man das Objektmodell dieser Datenstruktur. Ein KGraph besteht aus einer oder mehreren Hierarchieebenen. In der ersten Hierarchieebene befindet sich der Wurzelknoten (*engl. root*). Dieser kann beliebig viele Kinder, also Knoten in der nächsten Hierarchieebene, haben, welche wiederum beliebig viele Kinder in der nachfolgenden Hierarchieebene haben können. All diese Knoten sind sogenannte *KNodes*. Die Knoten können eingehende und ausgehende Kanten (*KEdges*) haben, bei denen der Start- und Zielknoten gespeichert ist. Weiterhin gibt es noch *KPorts*, welche aber im Umfeld von Sequenzdiagrammen nicht genutzt werden. Knoten, Kanten und Ports können mit Beschriftungen versehen sein. An den Graphen können verschiedene Eigenschaften angeheftet werden. Informationen über das Layout sind *KLayoutData*-Instanzen. Diese sind in *KShapeLayout* für Knoten, Ports und Beschriftungen und in *KEdgeLayout* für Kanten, aufgeteilt. Weiterhin können Optionen zur Konfiguration des Layouts an KLayoutData angeheftet werden. Informationen über die Darstellung sind *KRendering*-Instanzen.

Ein Sequenzdiagramm kann nicht so einfach auf diese Graphenstruktur abgebildet werden. Kapitel 4 erklärt genauer, warum das so ist und wie der KGraph aufgebaut sein muss, damit der Layoutalgorithmus damit arbeiten kann.

[1] `http://rtsys.informatik.uni-kiel.de/confluence`

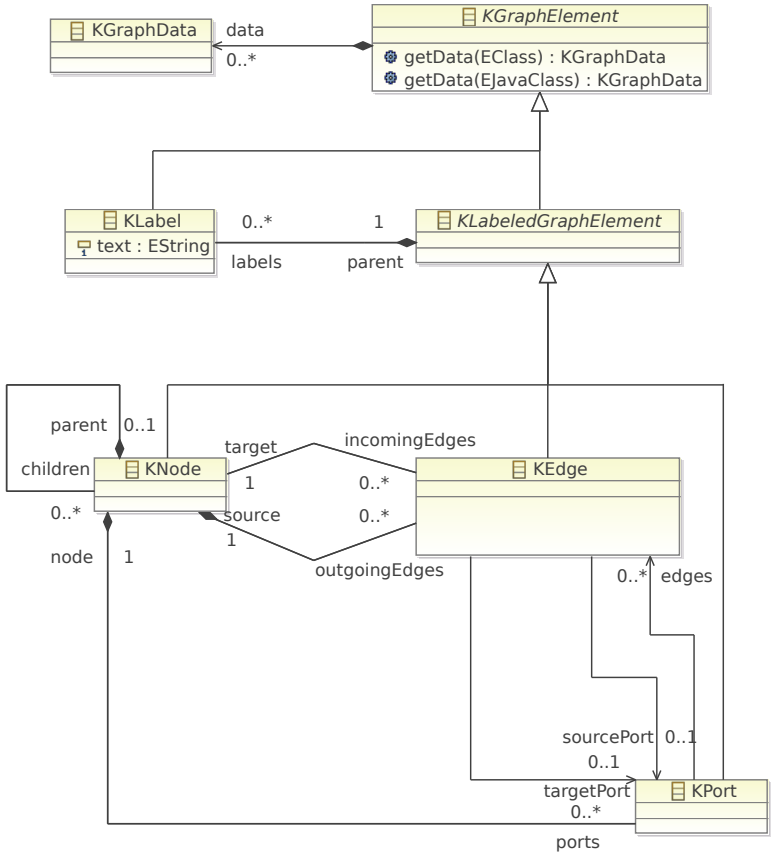


Abbildung 2.2. Aufbau eines KGraph. Quelle: Real-Time and Embedded Systems Kiel[1]

2.2.3 KIML

KIELER Infrastructure for Meta Layout (KIML) [SFH09] bietet Schnittstellen zwischen Layoutalgorithmen und Editoren für das automatische Layout von graphischen Modellen. Für das konkrete Layout eines Graphen ist der Layoutalgorithmus selbst zuständig. KIML kann mit vielen quelloffenen Layoutbibliotheken umgehen, wie zum Beispiel GraphViz,[2] oder aber mit den selbst entwickelten, die durch *KIELER Layout Algorithms (KLay)* zur Verfügung gestellt werden. Der Layouter für Sequenzdiagramme von Hoops [Hoo13] ist auch an KIML angebunden und kann daher verwendet werden.

[2] http://www.graphviz.org/

2.2.4 Xtext

Xtext[3] ist ein quelloffenes *Framework*, was das Erstellen von Programmiersprachen und Domänenspezifische textuelle Sprachen ermöglicht. Es wird als Plugin in Eclipse eingebunden. Alle Aspekte einer Sprachinfrastruktur können dabei angepasst werden. Man kann also zum Beispiel das Formatieren des Textes oder die Validation der Sprache nach den eigenen Wünschen verändern. Durch die Integration in Eclipse werden auch alle Vorteile des Anwendungs-Frameworks genutzt. Der Editor unterstützt durch Xtext eine intelligente Autovervollständigung, welche also auch in der erstellten Sprache genutzt werden kann. Weiterhin kann Xtext ein Objektmodell der Sprache erstellen. Beim Nutzen der Sprache wird dann eine Instanz des Objektmodells, also ein konkretes Modell mit konkreten Informationen, die man im Editor beschrieben hat, erstellt.

2.2.5 KLighD

KIELER Lightweight Diagrams (KLighD) vereinfacht das Visualisieren von Datenstrukturen [SSH13]. Es kombiniert die eben vorgestellten Technologien. Wie man in Abbildung 2.3 sieht, müssen erst mehrere Schritte durchlaufen werden, damit am Ende eine Visualisierung auf dem Bildschirm angezeigt werden kann. Zunächst wird die Instanz des Modells, welches wir durch Xtext erhalten, an KLighD weitergereicht. Mit der Synthese wird das Modell in einen KGraph umgewandelt, mit dem KLighD umzugehen weiß. Durch die KRendering-Informationen aus dem KGraph weiß KLighD, wie es die einzelnen Elemente des Graphen darstellen soll, aber noch nicht, wo diese platziert werden sollen. Der Graph wird daher an KIML weitergereicht, wodurch konkrete Positionen der Elemente des Graphen berechnet werden. Der Layoutalgorithmus berechnet genaue Koordinaten und speichert diese im sogenannte Makrolayout. KLighD berechnet das Mikrolayout und sorgt damit dafür, dass zum Beispiel dynamisch die Größe eines Rechtecks, wenn dieses Text beinhaltet, verändert wird. Das Ergebnis von KIML wird danach an KLighD zurückgeliefert und kann dann visuell dargestellt werden.

[3] http://www.eclipse.org/Xtext/

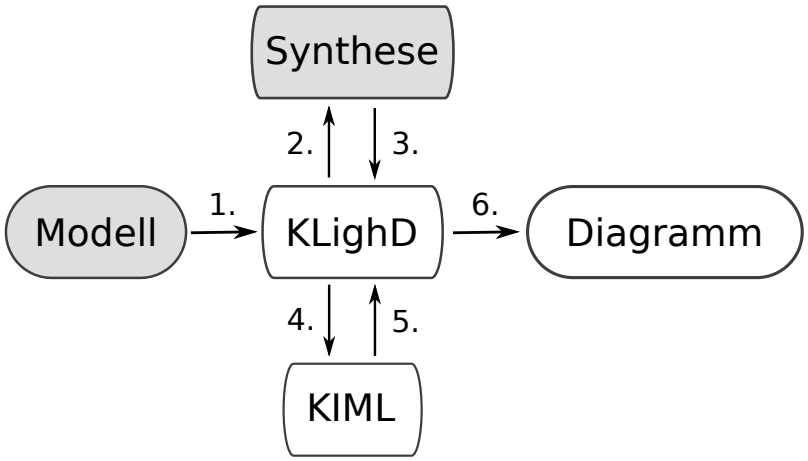


Abbildung 2.3. Schritte für die Visualisierung von Sequenzdiagrammen.

Kapitel 3

Textuelle Sprache

In diesem Kapitel analysiere ich zunächst die beiden textuellen Sprachen der gezeigten Editoren, die ich in Abschnitt 1.2 vorgestellt habe. Für jede Sprache gebe ich ein kurzes Codebeispiel des Sequenzdiagramms, welches in Abbildung 3.1 visualisiert ist. Danach erkläre ich meine Gedanken zu der Entwicklung von KIESL. Weiterhin gehe ich auf die Umsetzung von KIESL mit Hilfe von Xtext ein und zeige, wie das Codebeispiel vom Diagramm aus Abbildung 3.1 in KIESL aussieht. Als Letztes stelle ich die Besonderheiten des textuellen Editors vor.

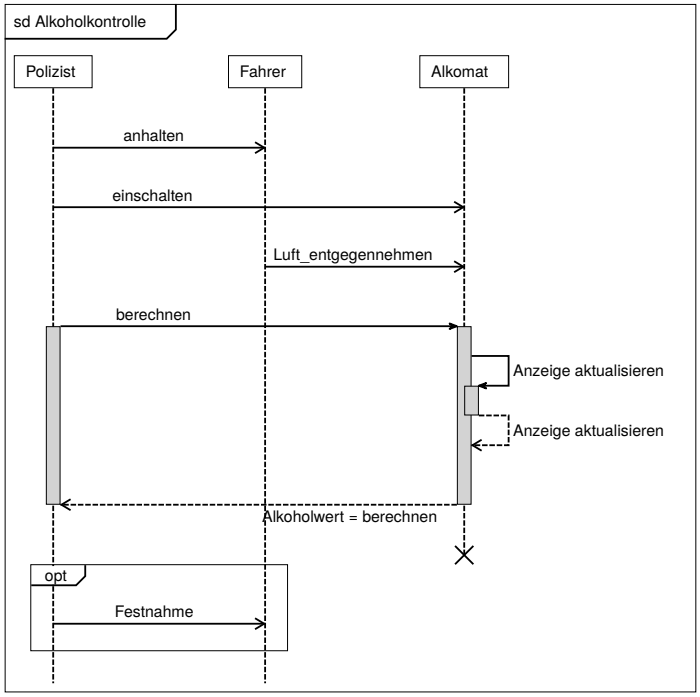


Abbildung 3.1. Sequenzdiagramm in Anlehnung an [JRH+04]

3.1 Analyse bestehender Sprachen

3.1.1 WebSequenceDiagrams

Der Grundgedanke der Sprache ist, dass man direkt anfängt Nachrichten zu modellieren. Eine Nachricht besteht aus zwei Lebenslinien (Sender und Empfänger), einem Nachrichtentyp und einem Nachrichtentitel. Falls eine in der Nachricht erwähnte Lebenslinie noch nicht existiert, wird sie automatisch erstellt. Die Syntax einer Nachricht setzt sich wie folgt zusammen:

```
<Sender> <Nachrichtentyp> <Empfänger> : <Nachrichtentitel>
```

Die unterschiedlichen Nachrichtentypen werden durch verschiedene Pfeile dargestellt. Eine asynchrone Nachricht zum Beispiel wird mit `->` identifiziert, wohingegen eine Antwortnachricht mit `-->>` identifiziert wird. Einen Erzeugungsaufruf unterstützt die Sprache nicht. Der Nachrichtentitel wird mit einem Doppelpunkt von der Lebenslinie abgetrennt. Wenn man eine andere Reihenfolge der Lebenslinien haben möchte, kann man die Lebenslinien am Anfang des Dokuments mit dem Schlüsselwort `participant` in eine Reihenfolge bringen. Weiterhin können den Lebenslinien kürzere Identifizierer gegeben werden, indem man mit dem Schlüsselwort `as` den neuen Namen anfügt:

```
participant <Lebenslinie> as <Identifizierer>
```

Die Angabe eines Titels für das Diagramm ist optional, aber der Titel kann mit Hilfe des Schlüsselworts `title` angegeben werden.

Mit `activate <Lebenslinie>` oder `deactivate <Lebenslinie>` werden Ausführungssequenzen, in auf die entsprechenden Nachrichten folgenden Zeilen gestartet, beziehungsweise beendet. Alternativ kann vor dem Empfänger zum Starten einer Ausführungssequenz auch ein `+` oder zum Beenden ein `-` verwendet werden, was aber oft unerwünschte Effekte hervorruft, da man diese Symbole nicht vor dem Sender nutzen kann und dort auf `activate` und `deactivate` zurückgreifen muss.

Drei Fragmenttypen (opt, loop, alt) werden in der Standardversion unterstützt und zwei weitere (par, ref) sind zu bezahlende Premiumfunktionen. Ein Fragment wird zum Beispiel mit dem Schlüsselwort `opt` gestartet und mit dem Schlüsselwort `end` beendet. Alle Nachrichten innerhalb der beiden Schlüsselwörter sind dann im Fragment enthalten.

Ein Zerstörungs-Event wird mit dem Schlüsselwort `destroy <Lebenslinie>` platziert. Im Folgenden ist ein Codebeispiel für das Sequenzdiagramm aus Abbildung 3.1:

```
1  title sd Alkoholkontrolle
2  participant Polizist as pol
3
4  pol ->> Fahrer:anhalten
5  pol ->> Alkomat:einschalten
6  Fahrer ->> Alkomat:Luft_entgegennehmen
7  pol -> Alkomat:berechnen
8  activate Alkomat
9  activate pol
10 Alkomat -> Alkomat:Anzeige aktualisieren
11 activate Alkomat
12 Alkomat -->> Alkomat:Anzeige aktualisieren
13 deactivate Alkomat
14 Alkomat -->> pol:
15 deactivate pol
16 destroy Alkomat
17 opt
18   pol ->> Fahrer:Festnahme
19 end
```

3.1.2 Quick Sequence Diagram Editor

Bei dieser Sprache fängt man zunächst damit an, alle Lebenslinien explizit zu deklarieren. Danach erst modelliert man die Nachrichten. Eine Deklaration der Lebenslinie sieht wie folgt aus:

```
<Identifizierer> : <Lebenslinie> [<Flag>] "<Alternative Bezeichnung>"
```

Den Identifizierer nutzt man bei den Nachrichten als Abkürzung für die entsprechende Lebenslinie. Das Flag ist optional und gibt mit dem Schlüsselwort `a` an, dass der Identifizierer nicht mit im Titel der Lebenslinie angezeigt wird. Ein weiteres Flag ist `x`, welches für die Lebenslinie automatisch am Ende ein Zerstörungs-Event platziert. Es können auch mehrere Flags gesetzt werden, die per Komma voneinander abgetrennt werden. Es gibt darüber hinaus noch weitere Flags, die ich hier nicht aufzählen werde. Eine alternative Bezeichnung kann angegeben werden, wenn man für die Lebenslinie eine komplett andere Beschriftung angezeigt haben möchte.

Die Basissyntax von Nachrichten sieht wie folgt aus:

```
<Sender> : <Titel der Antwortnachricht> = <Empfänger> . <Nachrichtentitel>
```

Dabei wird eine synchrone Nachricht erstellt, welche automatisch eine Antwortnachricht zurückgibt. Der Antwortnachricht kann mit `Titel der Antwortnachricht` ein Name gege-

ben werden. Um eine asynchrone Nachricht zu erstellen, nutzt man statt des Doppelpunkts `:>`. Ausführungssequenzen werden automatisch erstellt und beendet. Das automatische Erstellen von Antwortnachrichten und Ausführungssequenzen führt allerdings oft zu unerwünschten Ergebnissen und schränkt den Nutzer eher ein anstatt ihm zu helfen.

Ein Titel des Diagramms ist hier ebenfalls optional, kann aber mit `#![<Titel>]` angegeben werden.

Ein Fragment wird mit `[c:<Fragmenttyp> <Bedingung>]` begonnen und endet mit `[/c]`. Die Bedingung ist dabei optional. An sich unterstützt dieser Editor also alle Funktionen von Sequenzdiagrammen aus der UML. Im Folgenden wird ebenfalls das Sequenzdiagramm aus Abbildung 3.1 beschrieben:

```
1  #![Alkoholkontrolle]
2
3  pol:Polizist[a]
4  fahrer:Fahrer[a]
5  alk:Alkomat[a, x]
6
7  pol:>fahrer.anhalten
8  pol:>alk.einschalten
9  fahrer:>alk.Luft_entgegennehmen
10 pol:alk.berechnen
11 alk:alk.Anzeige aktualisieren
12 [c:opt]
13   pol:fahrer.Festnahme
14 [/c]
```

3.2 Leitgedanken bei der Entwicklung von KIESL

Bei der Überlegung, wie ich KIESL gestalten möchte, habe ich versucht, die Vorteile beider eben vorgestellter Sprachen zu kombinieren.

Zunächst ist es bei KIESL erforderlich, dem Diagramm einen Namen zu geben. Bei den beiden Editoren ist der Name nur optional. Mir war es aber wichtig, dass man sich bei den Diagrammen an die UML-Spezifikation[1] hält. Dort sind Sequenzdiagramme mit einer umgebenden Interaktion umschlossen und haben somit auch einen Titel.

Beim Quick Sequence Diagram Editor müssen Lebenslinien als Erstes deklariert werden; erst danach beginnen die Nachrichten. WebSequenceDiagrams erstellt die Lebenslinien implizit und das Angeben der Lebenslinien am Anfang ist optional. Für einen besseren Überblick über die Lebenslinien, welche im aktuellen Diagramm vorkommen, ist es sinnvoll

[1] `http://www.uml.org/`

die Lebenslinien am Anfang deklarieren zu müssen. Bei KIESL muss man jeder Lebenslinie einen Identifizierer geben, da dies zu deutlich kürzerem und übersichtlicherem Code führt.

Bei WebSequenceDiagrams kann man Ausführungssequenzen manuell steuern und somit selbst bestimmen, mit welcher Nachricht die Ausführung beendet wird. Beim Quick Sequence Diagram Editor wird bei jeder Nachricht automatisch eine Ausführungssequenz gestartet, was oft zu unerwünschten oder visuell überladenen Resultaten führt. Deshalb habe ich mich hierbei für den Ansatz von WebSequenceDiagrams entschieden.

Beim Quick Sequence Diagram Editor wird bei einer synchronen Nachricht die Antwortnachricht automatisch erstellt. Diese Option finde ich insofern gut, als dass sie den Code und somit auch die Zeit, die zum Schreiben benötigt wird, verkürzt. Die Antwortnachrichten werden allerdings nicht immer an der gewünschten Stelle platziert, weshalb ich diese Option bei mir nicht eingebaut habe. Wir betrachten folgendes Codebeispiel für den Quick Sequence Diagram Editor:

```
1 l1:Lebenslinie1[a]
2 l2:Lebenslinie2[a]
3
4 [c loop]
5   l1:l2.Nachricht1
6   l1:Antwort=l2.Nachricht2
7 [/c]
8 l2:l1.Nachricht3
```

Es fällt auf, dass `l1:Antwort=l2.Nachricht2` innerhalb des Fragments steht. Wenn man sich allerdings Abbildung 3.2 ansieht, dann ist die Antwortnachricht nicht im Fragment enthalten.

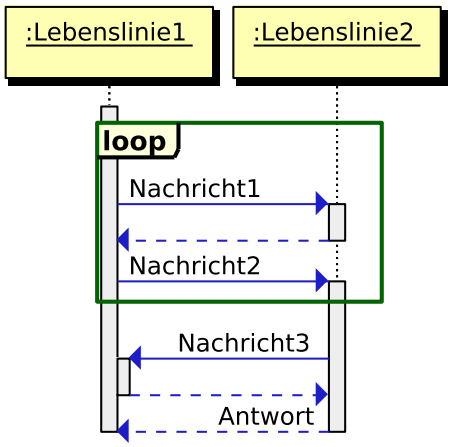


Abbildung 3.2. Falsche Platzierung der Antwortnachricht im Quick Sequence Diagram Editor

3.3 Realisierung von KIESL

Als Erstes gibt man bei KIESL mit dem Schlüsselwort `sequenceDiagram "Titel"` den Titel des Diagramms an. Daraufhin erstellt man die Lebenslinien mit folgender Syntax:

```
lifeline "<Lebenslinie>" as <Identifizierer>
```

Alternativ kann man statt dem Schlüsselwort `lifeline` auch das Schlüsselwort `usecase` verwenden, um einen Anwendungsfall anstatt einer Lebenslinie zu erstellen.

Danach werden alle Nachrichten definiert:

```
<Sender> <Nachrichtentyp> "<Nachrichtentitel>" to <Empfänger> <Optionen>
```

Eine Nachricht ist dabei anhand des Leseflusses aufgebaut. Wenn man ein Diagramm liest, sieht man zuerst den Sender der Nachricht. Daraufhin folgt man dem Pfeil der Nachricht, also erkennt man den Nachrichtentyp mit dem beinhalteten Nachrichtentitel. Als Letztes sieht man den Empfänger der Nachricht. Die Nachrichtentypen werden bei mir durch verschiedene Schlüsselwörter (`async, sync, create, response, lost, found`) identifiziert. Die Idee, Nachrichten mit Pfeilen zu identifizieren, ist zwar gut, allerdings kann man bei zu vielen Nachrichtentypen, diesen nicht immer einfach anhand des Pfeiles erkennen.

Im Wesentlichen können drei Optionen angegeben werden. Zuerst kann eine Ausführungssequenz beim Sender und/oder Empfänger mit dem Schlüsselwort `sourceStartExec`, beziehungsweise `targetStartExec`, gestartet werden. Daraufhin kann man eine Ausführungssequenz beim Sender und/oder Empfänger mit dem Schlüsselwort `sourceEndExec`, beziehungsweise `targetEndExec`, beenden und als Letztes kann eine Notiz beim Sender und/oder Empfänger mit dem Schlüsselwort `sourceNote "<Notiz>"`, beziehungsweise `targetNote "<Notiz>"` angeheftet werden. Für die Optionen, welche `endExec` enthalten, kann hinter dem Schlüsselwort noch eine Zahl oder `all` geschrieben werden, um eine bestimmte Anzahl oder alle Sequenzen auf der zugehörigen Lebenslinie zu beenden. Alternativ zu einer `startExec` kann man auch das Schlüsselwort `sourceStartEndExec`, beziehungsweise `targetStartEndExec` benutzen, um eine Sequenz zu starten, die vor dem Beginn der nächsten Nachricht auf der entsprechenden Lebenslinie automatisch beendet wird.

Ein Zerstörungs-Event wird durch `<Identifizierer> destroy` platziert.

Ein Fragment hat in KIESL folgende Syntax:

```
fragment "<Fragmenttyp>" {
  label "<Bedingung>"
  <Interaktionen>
} {
  label "<Bedingung>"
  <Interaktionen>
}
```

Die Zeile mit dem `label` ist optional und gibt die Bedingung im Fragment an. Mit Interaktionen sind Nachrichten und weitere Fragmente gemeint. Alle Interaktionen, die in den geschweiften Klammern aufgeführt sind, werden innerhalb des Fragments angezeigt. Wenn ohne das Schlüsselwort `fragment` ein weiteres Paar geschweifter Klammern angefügt wird, dann wird eine neue Region im Fragment erstellt, was beliebig oft wiederholt werden kann.

Im Folgenden ist das Codebeispiel von Abbildung 3.1 nun in KIESL angegeben:

```
1  sequenceDiagram "Alkoholkontrolle"
2
3  lifeline "Polizist" as po
4  lifeline "Fahrer" as fa
5  lifeline "Alkomat" as al
6
7  po async "anhalten" to fa
8  po async "einschalten" to al
9  fa async "Luft_entgegennehmen" to al
10 po sync "berechnen" to al
11         sourceStartExec
12         targetStartExec
13 al sync "Anzeige aktualisieren" to al
14         sourceStartExec
15 al response "Anzeige aktualisieren" to al
16         sourceEndExec
17 al response "" to po
18         sourceEndExec
19         targetEndExec
20 al destroy
21 fragment "opt" {
22         po async "Festnahme" to fa
23 }
```

3.4 Editor

Der Editor in Eclipse, in dem KIESL letztendlich genutzt wird, kann durch Xtext noch einige Besonderheiten erhalten. Als Erstes habe ich einen Formatierer entwickelt. In Eclipse kann man den geschriebenen Text mit der Tastenkombination *Strg* + *Shift* + *F* automatisch formatieren lassen. Dem Formatierer müssen also Informationen gegeben werden, damit dieser weiß, wie er den Text formatieren soll. Die Formatierung von KIESL sieht man in dem Codebeispiel aus Abschnitt 3.3. Hinter dem Diagrammtitel wird eine

leere Zeile eingefügt. Die Deklarationen der Lebenslinien stehen jeweils in einer Zeile, gefolgt von einer weiteren leeren Zeile. Nachrichten werden ebenfalls in jeweils eine Zeile geschrieben und falls diese mit Optionen versehen sind, werden diese in die nächste Zeile und um eine Einrückungsebene verschoben. Jede Option steht dabei alleine in einer Zeile. Bei Fragmenten wird die öffnende geschweifte Klammer in dieselbe Zeile wie das Schlüsselwort für Fragmente geschrieben und die schließende Klammer steht alleine in einer Zeile. Alle Interaktionen, die von den Klammern umschlossen sind, werden dabei auch um eine Einrückungsebene verschoben.

Die zweite Besonderheit ist, dass ich eine eigene Validation erstellt habe. Der normale Editor erkennt nur Syntaxfehler, aber keine logischen Fehler. Zum Beispiel, wenn man eine Ausführungssequenz beenden möchte, aber noch keine gestartet hat, würde der normale Editor diesen Fehler nicht erkennen. Ebenfalls überprüfe ich, ob der Identifizierer einer Lebenslinie bereits gewählt wurde, damit jede Lebenslinie eindeutig zuzuordnen ist. Weiterhin überprüfe ich, dass nach dem Zerstörungs-Event keine Nachrichten mehr auf der zerstörten Lebenslinie gesendet oder empfangen werden können.

Xtext erstellt automatisch einen globalen Geltungsbereich (*engl. global scope*) der Lebenslinien, welcher auf einen lokalen Geltungsbereich verringert werden musste, damit die Lebenslinien nur in der aktuellen Datei und nicht in den anderen Dateien aus demselben Projekt bekannt sind. „Bekannt“ bedeutet hierbei, dass die Lebenslinien in einem kleinen Auswahlfenster vorgeschlagen werden, wenn man Nachrichten mit Hilfe der Autovervollständigung erstellen möchte.

Kapitel 4

Synthese

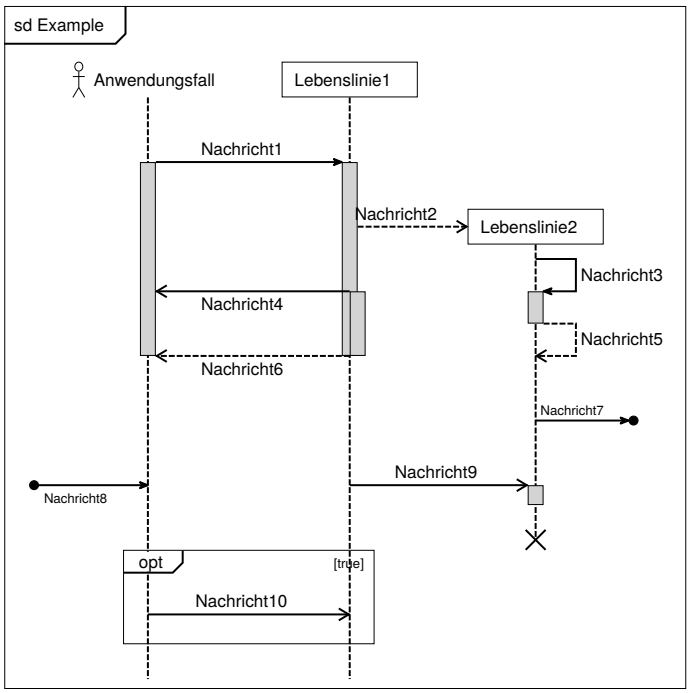
Ein Sequenzdiagramm kann nicht wie ein gewöhnlicher Graph in einen KGraph umgewandelt werden. Das liegt daran, dass ein Sequenzdiagramm eine komplett andere Struktur als andere Graphen hat. Bei einem gewöhnlichen Graphen werden Knoten als Kreis oder Rechteck dargestellt. Falls die Knoten Ports enthalten, verbinden die Kanten die Ports, ansonsten verbinden sie die Knoten. Die Kanten können schräg, rund oder gerade und mit beliebig vielen Beugungspunkten gezeichnet werden. Bei einem Sequenzdiagramm hingegen werden die Nachrichten zwischen den Zeitachsen der Lebenslinien ausgetauscht. Des Weiteren müssen die Ausführungssequenzen und Zerstörungs-Events auf diesen Lebenslinien platziert werden. Als Letztes gibt es noch Fragmente, die innerhalb der umgebenden Interaktion um Nachrichten gezeichnet werden können. Die Nachrichten müssen sowohl Lebenslinien, als auch Ausführungssequenzen und Fragmenten gleichzeitig zugeordnet werden können. Wie man das ermöglichen kann, werde ich im Folgenden erläutern. Danach erkläre ich, welche KRendering-Informationen an den einzelnen Elementen angebracht sind und welche Optionen man für die Darstellung des Diagramms auswählen kann.

4.1 Aufbau des KGraph

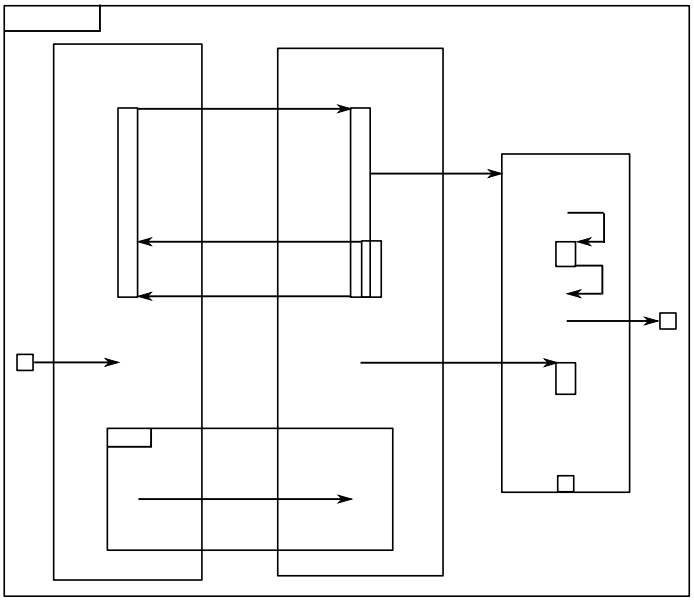
Bis auf die Kanten sind alle Elemente vom Sequenzdiagramm als Knoten repräsentiert. Zunächst zeige ich in Abbildung 4.1 einen Vergleich eines Sequenzdiagramms mit den zugehörigen Knoten.

Damit der Layoutalgorithmus von Hoops [Hoo13] arbeiten kann, muss eine spezielle Struktur, welche in Abbildung 4.2 abgebildet ist, eingehalten werden. Die Objekte, die durch die textuelle Sprache erstellt werden, müssen in diese Struktur transformiert werden. Die Quelle und das Ziel einer Nachricht muss immer eine Lebenslinie sein. Damit die Kanten weiterhin sowohl Ausführungssequenzen als auch Fragmenten zugeordnet werden können, nutzt man verschiedene Eigenschaften, welche auf den Kanten gesetzt werden müssen.

Jeder Graph hat einen Wurzelknoten, der das Diagramm repräsentiert, selbst aber nicht gezeichnet wird. Die umgebende Interaktion wird vom Algorithmus auch als Wurzelknoten erwartet, soll aber im Diagramm dargestellt werden. Um das zu realisieren, wird an den



(a) Sequenzdiagramm in der fertigen Darstellung



(b) Aufbau des zugehörigen KGraph

Abbildung 4.1. Der Aufbau des KGraphen in einem Sequenzdiagramm

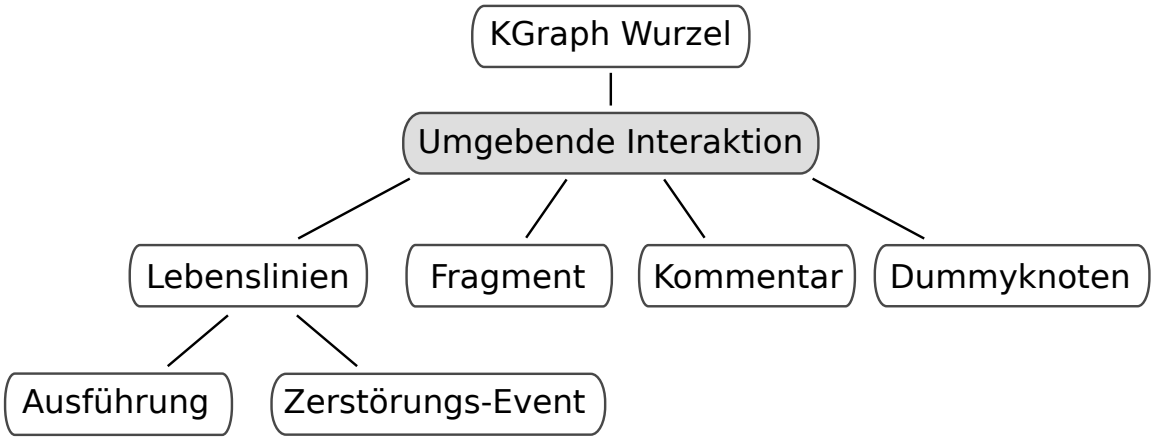


Abbildung 4.2. Die Struktur des KGraphen

wirklichen Wurzelknoten der *Fixed Layout Algorithmus* als Layoutoption angehängt. Dieser berechnet keine Koordinaten und verändert das Layout des Diagramms nicht. Außerdem wird der Knoten der umgebenden Interaktion dem Wurzelknoten als Kind hinzugefügt. An der umgebenden Interaktion kann nun der Layoutalgorithmus für Sequenzdiagramme als Layoutoption gesetzt werden; somit ist die umgebende Interaktion der Wurzelknoten dieses Layoutalgorithmus. Für diesen Knoten und alle folgenden Knoten wird ein Knotentyp als Eigenschaft gesetzt, damit die verschiedenen Knoten vom Layoutalgorithmus als die entsprechenden Elemente des Sequenzdiagramms erkannt werden können. Die umgebende Interaktion hat folgende Kinder:

Lebenslinien: Die Lebenslinien können Ausführungssequenzen und ein Zerstörungs-Event als Kinder enthalten, da diese Elemente auf der entsprechenden Lebenslinie angezeigt werden. Um Ausführungssequenzen Nachrichten zuordnen zu können, erhält jede Ausführungssequenz eine eindeutige Identifikationsnummer. Für die Quelle und das Ziel einer Nachricht wird jeweils eine Liste aller Ausführungssequenzen, in denen die Nachricht vorkommen soll, als Eigenschaft an die Nachricht angehängt.

Fragmente: Jedem Fragment wird ebenfalls eine eindeutige Identifikationsnummer zugewiesen. Wenn eine Nachricht innerhalb von Fragmenten dargestellt werden soll, wird eine Liste von den Identifikationsnummern der Fragmente als Eigenschaft an die entsprechende Nachricht angeheftet.

Kommentare: Nachrichten haben auch eine eindeutige Identifikationsnummer, damit die Nummern an diesen Knoten angeheftet werden können und der Kommentar somit der entsprechenden Nachricht zugeordnet werden kann.

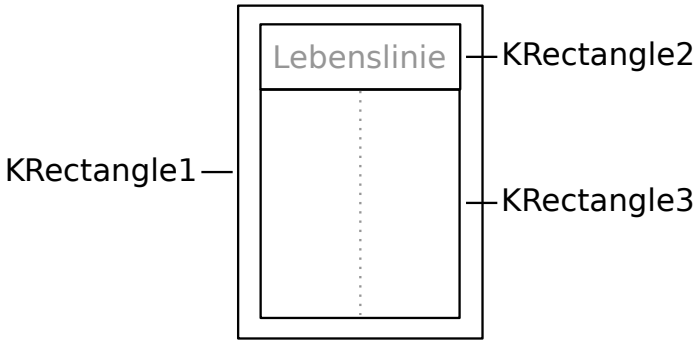


Abbildung 4.3. KRendering für eine Lebenslinie

Dummyknoten: Zu diesen Knoten gehören die Punkte, welche für verlorene beziehungsweise gefundene Nachrichten erstellt werden müssen. Ein solcher Knoten wird dann entsprechend als Sender oder Empfänger der Nachricht angegeben.

Nachrichten werden als Kanten an den Algorithmus übergeben. Es wird ähnlich wie bei den Knoten eine Eigenschaft für den entsprechenden Nachrichtentyp gesetzt. Für jede Nachricht wird die Quelle und das Ziel der Nachricht auf die jeweiligen Lebenslinien gesetzt und die Nachrichten erhalten eine Nummerierung, damit sie in der richtigen Reihenfolge im Diagramm angezeigt werden können.

4.2 KRendering

Die im vorherigen Abschnitt genannten Elemente sind mit KRendering-Informationen gefüllt, sodass sie im Diagramm sinnvoll angezeigt werden.

Umgebende Interaktion: Im Diagramm wird der Knoten der umgebenden Interaktion als Rechteck (KRectangle) repräsentiert und beinhaltet ein weiteres KRectangle, welches mit Platzierungsdaten in die obere linke Ecke des vorherigen KRectangle verschoben wird und den Namen des Diagramms beinhaltet.

Lebenslinien: In Abbildung 4.3 wird der Aufbau der KRectangles von Lebenslinien repräsentiert. Das KRectangle1 wird in der Synthese als nicht sichtbar markiert. Es enthält zwei weitere KRectangles (KRectangle2 und KRectangle3), welche untereinander angeordnet sind. KRectangle2 beinhaltet den Namen der Lebenslinie. KRectangle3 wird ebenfalls nicht im Diagramm dargestellt, aber enthält die gestrichelte Linie in der Mitte.

Ausführungssequenz: Eine Ausführungssequenz wird als KRectangle dargestellt.

Zerstörungs-Event: Für das Zerstörungs-Event wird ein Kreuz in einem nicht sichtbaren KRectangle gezeichnet. Dieses Kreuz setzt sich dabei aus zwei Linien (Polyline) zusammen.

Fragment: Ein Fragment ist genauso wie die umgebende Interaktion aufgebaut. Es besteht also aus einem KRectangle und einem zweiten KRectangle in der oberen linken Ecke, welches den Titel des Fragments enthält.

Kommentar: Für einen Kommentar enthält ein KRectangle den Text, der dargestellt werden soll.

Dummyknoten: Ein nicht sichtbares KRectangle enthält einen schwarz ausgefüllten Kreis für die Dummyknoten.

Nachricht Bei einer Nachricht wird die Pfeilspitze und Repräsentation der Kante je nach Nachrichtentyp verändert. Für eine synchrone Nachricht ist die Kante zum Beispiel durchgezogen und die Pfeilspitze ausgefüllt. Für eine Antwortnachricht hingegen ist die Kante gestrichelt und die Pfeilspitze offen.

4.3 Diagrammoptionen

Für das fertige Diagramm hat man die Möglichkeit, zwei Optionen per Mausklick zu verändern. Zunächst kann man zwischen drei verschiedenen Designs des Diagramms wählen, die sich vor Allem auf im Diagramm verwendete Farben sowie einige Formen auswirken.

Weiterhin hat man die Möglichkeit, die Anordnung der Lebenslinien zu verändern. Dafür unterstützt der Layoutalgorithmus drei verschiedene Vorgehensweisen: *Interactive*, *Layer Based* und *Short Messages*. *Interactive* zeigt die Lebenslinien in der Reihenfolge, in der sie definiert werden, an. *Layer Based* versucht, eine Treppenstruktur der Nachrichten zu erzeugen. *Short Messages* versucht schließlich, bei allen Nachrichten insgesamt so wenig Kreuzungen mit anderen Lebenslinien wie möglich zu erhalten.

Kapitel 5

Evaluation

Beim Erstellen von Software, mit der Anwender interagieren müssen, ist es sinnvoll, die Software mit ähnlichen Produkten zu vergleichen. Dafür habe ich eine kleine Studie durchgeführt, um zu testen, inwiefern sich die Bedienbarkeit von KIESL mit anderen textuellen Sprachen vergleichen lässt und ob sich eine textuelle Sprache im Gegensatz zu einem Drag & Drop-Editor durchsetzen kann.

5.1 Aufbau der Studie

Die Studie wurde mit 20 Teilnehmern aus dem Bereich der Informatik durchgeführt. Jeder Teilnehmer hat einen von fünf Editoren zugewiesen bekommen, wussten aber nicht, welcher KIESL ist. Als Editoren wurde KIESL, die drei Editoren aus Abschnitt 1.2 und *Visual Paradigm* genutzt. Visual Paradigm ist, wie Papyrus, ein Drag & Drop-Editor. Es wurde eine kurze Einführung in den jeweiligen Editor gegeben; danach sollten vier vorgegebene Sequenzdiagramme mit dem vorgegebenen Editor nachgebaut werden. Dabei wurde für jedes Diagramm die Zeit gemessen und den Teilnehmern wurde vorher gesagt, dass sie maximal zehn Minuten pro Diagramm zur Verfügung haben. Da die Zeit aber selten überschritten wurde, habe ich nur einen kurzen Hinweis gegeben, dass die zehn Minuten erreicht wurden, aber das Diagramm durfte noch beendet werden. Am Ende sollte ein kurzer Fragebogen über den Editor beantwortet werden. Dabei sollten Vor- und Nachteile des Editors beschrieben und eine Bewertung auf einer Skala von 1 bis 10 gegeben werden. Die Studie hat pro Teilnehmer maximal 45 Minuten Zeit in Anspruch genommen.

Bei dem Aufbau der Studie habe ich mich an einer Methode von Purchase [Pur12] orientiert. Ich nutze das *between-participants experiment*, in dem Teilnehmer wie eben beschrieben nach Editoren gruppiert werden. Alternativ zu der Durchführung, die ich gewählt habe, hätte ich auch das *within-participants experiment* wählen können, in dem jeder Teilnehmer alle Editoren nutzen müsste. Dadurch könnte er die Editoren untereinander vergleichen, allerdings hätte die Studie dann zu viel Zeit in Anspruch genommen. Für jeden Editor benötigt der Teilnehmer Einarbeitungszeit und die Teilnehmer werden bei zu langer Studiendauer unkonzentriert. Weiterhin wäre es nicht sinnvoll, für jeden Editor das gleiche Diagramm erstellen zu lassen, da man dadurch schon Vorkenntnisse hat und die späteren Editoren dann wohlmöglich besser abschneiden. Es ist allerdings nicht einfach

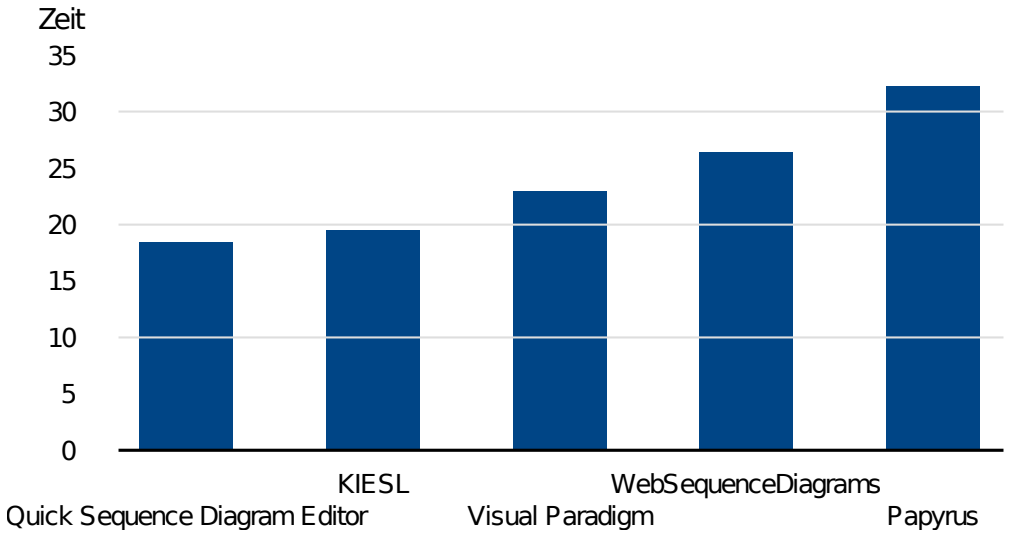


Abbildung 5.1. Durchschnittliche Gesamtzeit für die Absolvierung der Studie

verschiedene Diagramme des gleichen Umfangs und der gleichen Schwierigkeit für jeden Editoren zu finden.

Bei der Methode, die ich gewählt habe, gibt es weitere Nachteile. Einerseits müssen die Teilnehmer den Editor subjektiv bewerten, da sie keinen Vergleichswert haben. Andererseits hängen die Ergebnisse der Studie von der individuellen Leistung der Teilnehmer ab. Manche Teilnehmer, die einen textuellen Editor bekommen haben, schreiben eventuell etwas langsamer. Andere arbeiten sich sehr langsam in den jeweiligen Editor ein und benötigen deshalb mehr Zeit. Bei einer anderen zufälligen Verteilung der Teilnehmer auf die Editoren hätten die Ergebnisse somit anders ausfallen können. Weil die Teilnehmerzahl dieser Studie mit vier Teilnehmern pro Editor eher klein ist, hätte dieses Phänomen bei mehr Teilnehmern geringere Auswirkungen haben können. Die Ergebnisse der Studie müssen also mit Vorsicht betrachtet werden, können aber durchaus einen Eindruck vermitteln.

5.2 Ergebnisse der Studie

In Abbildung 5.1 wird die durchschnittliche Gesamtzeit aller Aufgaben dargestellt. KIESL ist dabei knapp hinter dem Quick Sequence Diagram Editor auf Platz zwei gelandet. Um die Daten weiter auswerten zu können, habe ich mir die fertigen Diagramme noch einmal angesehen und in Tabelle 5.1 die Anzahl der Fehler und die häufigsten Fehlerquellen notiert. Dabei fällt auf, dass mit dem Quick Sequence Diagram Editor vier mal so viele Fehler in den Diagrammen waren als mit KIESL. Die Antwortnachrichten wurden durch den Automatismus vom Quick Sequence Diagram Editor oft an der falschen Stelle platziert,

Tabelle 5.1. Die Fehler der Teilnehmer in den Editoren (nach Fehleranzahl sortiert)

Editor	Fehleranzahl	Fehlerquelle(n)
KIESL	4	Falscher Nachrichtentyp
WebSequence-Diagrams	10	Falscher Nachrichtentyp
Visual Paradigm	12	Falscher Nachrichtentyp; Antwortnachricht an der falschen Stelle
Papyrus	14	Falscher Nachrichtentyp; Nachricht nicht immer horizontal platziert; Antwortnachricht fehlt; Sequenz in Verbindung mit asynchronen Nachrichten, obwohl sie dort nicht notwendig ist
Quick Sequence Diagram Editor	16	Falscher Nachrichtentyp; Antwortnachricht an der falschen Stelle, wodurch die Sequenz zu lang oder zu kurz ist

wodurch die Ausführungssequenzen zu lang oder zu kurz geraten sind.

Als Nächstes habe ich in Abbildung 5.2 die durchschnittliche Zeit pro Aufgabe von jedem Editor angegeben. Bei der Aufgabe A3 sollte Abbildung 1.1 erstellt werden, mit dem einzigen Unterschied, dass die Erzeugungsaufrufe zu asynchronen Nachrichten geändert wurden, da WebSequenceDiagrams keine Erzeugungsaufrufe unterstützt. Im Vergleich mit den anderen Aufgaben war dieses Diagramm umfangreicher. Das Erstellen hat in Visual Paradigm im Durchschnitt weniger Zeit beansprucht als mit KIESL. Das könnte daran liegen, dass bei Visual Paradigm Antwortnachrichten schnell mit einem Klick erstellt und für Nachrichtentitel aus einem Auswahlmenü bereits genutzte Titel ausgewählt werden können. Da bei dieser Aufgabe die Titel der Antwortnachrichten mit dem Titel der zugehörigen synchronen Nachricht übereinstimmen, spart man hier viel Zeit.

In Tabelle 5.2 werden die im Fragebogen genannten Vor- und Nachteile der verschiedenen Editoren beschrieben. Bei KIESL wurde als Nachteil bei großen Diagrammen erwähnt, dass man keine Markierung im Diagramm erhält, auf welches Element sich die Zeile, in der sich der Cursor aktuell befindet, bezieht. Dadurch würde man schneller die Position finden, an der man ein Element einfügen oder bearbeiten muss, falls ein Fehler im Diagramm aufgetreten ist. Obwohl sich KIESL und WebSequenceDiagrams nicht sehr groß unterscheiden, schnitt KIESL in der Studie trotzdem besser ab. Das könnte zum Beispiel daran liegen, dass WebSequenceDiagrams sehr viel Zeit benötigt hat, um die Änderungen aus dem Editor im Diagramm anzuzeigen.

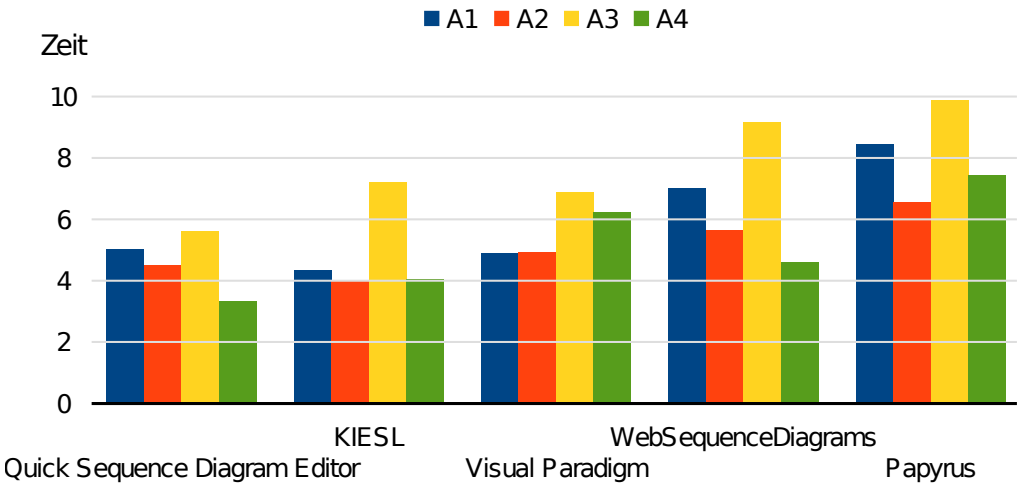


Abbildung 5.2. Durchschnittliche Bearbeitungszeit jeder Aufgabe aus der Studie

Als Letztes sollte jeder Editor auf einer Skala von eins bis zehn bewertet werden, wobei eins die schlechteste mögliche Bewertung ist. In Abbildung 5.3 sind die Editoren nach der durchschnittlichen Bewertung der Teilnehmer geordnet. Dabei fällt auf, dass die Editoren nach dieser Metrik genauso geordnet sind, wie wenn man sie, wie in Tabelle 5.1 gezeigt, nach der Anzahl der damit produzierten Fehler ordnet. Die Skala ist natürlich rein subjektiv, da die Teilnehmer keinen Vergleich hatten, aber spiegelt den ersten Eindruck und die mögliche Zufriedenheit mit dem jeweiligen Editor wider.

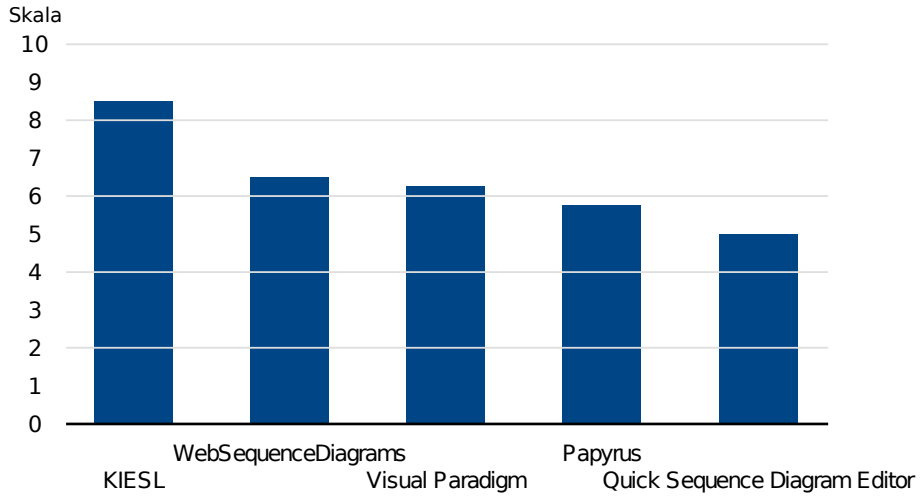


Abbildung 5.3. Durchschnittliche Bewertung der Editoren

Tabelle 5.2. Vor- und Nachteile der Editoren

Editor	Vorteile	Nachteile
Quick Sequence Diagram Editor	Automatisches Anzeigen des Diagramms	Angabe der Antwort unintuitiv Zeitliche Reihenfolge der Nachrichten unklar
KIESL	Autovervollständigung Autoformatierung Syntaxhervorhebung Intuitive Sprache Automatisches Anzeigen des Diagramms	Reihenfolge von `start/endExec` musste eingehalten werden Überflüssige Vorschläge von `endExec`, falls noch keine Sequenz gestartet wurde Keine Markierung im Diagramm, an welcher Stelle sich der Cursor im Text befindet
Visual Paradigm	Vorschläge der bereits eingegebenen Nachrichtentitel Für Nachrichten kann mit einem Klick eine Antwortnachricht erstellt werden	Zu viele Menüpunkte und Auswahlmöglichkeiten (besonders bei Fragmenten) Sequenz kann nicht gelöscht werden und wird teilweise automatisch verlängert Keine Schnellauswahl der verschiedenen Nachrichtentypen
WebSequence-Diagrams	Intuitive Sprache Übersichtlicher Aufbau	Keine Autovervollständigung Langsame Visualisierung Zerstörungs-Event zu dicht an den Nachrichten platziert
Papyrus	Einfaches Platzieren von Elementen...	...außer Nachrichten, welche zum Beispiel trotz korrekter visueller Rückmeldung teilweise nicht platziert werden können Beim ersten Bearbeiten eines Titels wird der automatisch erstellte Titel nicht überschrieben Elemente müssen nach jeder Platzierung neu gewählt werden

Kapitel 6

Schlussfolgerung

Das letzte Kapitel fasst noch einmal kurz zusammen, was im Laufe dieser Arbeit entwickelt wurde und welche Probleme noch offen stehen, beziehungsweise welche neuen Arbeiten sich daraus ergeben können.

6.1 Zusammenfassung

In dieser Arbeit wurde für das KIELER-Projekt die textuelle Sprache namens KIESL entwickelt. Mit Hilfe von Xtext wurde KIESL erstellt und ein Objektmodell generiert. Um eine Instanz des Objektmodells visuell darstellen zu können, muss sie in einen KGraph transformiert werden, damit der Layoutalgorithmus Koordinaten berechnen kann. Weiterhin werden den Knoten und Kanten während der Transformation Layoutoptionen und KRendering-Informationen angeheftet, damit KLighD Informationen über die Positionen und das Aussehen von den Knoten und Kanten aus dem KGraph erhält und das Diagramm schließlich darstellen kann.

Als Nächstes habe ich eine Studie für die Benutzbarkeit meiner Sprache durchgeführt. In der Studie wollte ich KIESL mit anderen textuellen Sprachen vergleichen und die Effektivität von textuellen Sprachen im Gegensatz zu Drag & Drop-Editoren testen. Dabei ist herausgekommen, dass beim Arbeiten mit KIESL deutlich weniger Fehler als in anderen Editoren aufgetreten sind und KIESL dabei den zweiten Platz bei der durchschnittlichen Gesamtzeit erreichen konnte. Die Bewertungsskala und die Vor- und Nachteile, die die Teilnehmer über KIESL angemerkt haben, sind ebenfalls sehr positiv ausgefallen.

Im Großen und Ganzen scheint es so, dass KIESL eine intuitive Sprache geworden ist. Durch das Feedback und die ermittelten Zeiten aus der Studie komme ich zu dem Schluss, dass sich KIESL gegen Drag & Drop-Editoren und andere textuelle Sprachen durchsetzen kann.

6.2 Ausblick

Der Layoutalgorithmus unterstützt nicht alle Spezifikationen der UML und manche Elemente werden in Kombination mit Fragmenten noch nicht korrekt im Diagramm dargestellt. Als weiterführende Arbeit kann also damit begonnen werden, diese Fehler zu beheben und

weitere Elemente der UML im Layoutalgorithmus und danach in KIESL zu unterstützen. Die nicht unterstützten Elemente sind die Folgenden: geschachtelte Fragmente, Regionen in Fragmenten, Kommentare, Interaktionsreferenzen, Verknüpfungspunkte, lokale Variablen, Zustandsinvarianten, Ordnungsbeziehungen, Zeitangaben, Zeitliche Beschränkungen und Koregionen.

KIESL selbst kann durch die Anmerkungen der Teilnehmer aus der Studie auch noch weiter verbessert werden.

1. Bei der Autovervollständigung von beispielsweise dem Schlüsselwort `lifeline`, könnte die komplette notwendige Deklaration eingefügt werden. Mit Hilfe der Tabulatortaste könnte dann zwischen den Elementen der Deklaration gewechselt werden, um diese schnell eingeben zu können.

2. Es könnte automatisch eine Antwortnachricht beim Erstellen einer synchronen Nachricht erstellt werden. Dabei könnte auch eine Sequenz mit dem Schlüsselwort `endExec` beendet werden, falls gewünscht.

3. Die Optionen von KIESL, wie zum Beispiel `sourceStartExec` oder `targetEndExex` erfordern noch eine feste Reihenfolge. Zuerst müssen Sequenzen gestartet und danach können sie beendet werden und als Letztes können Kommentare angefügt werden. Weiterhin muss `source...` immer vor `target...` stehen. Da diese Reihenfolge nicht unbedingt verständlich ist, wurde sich eine beliebige Reihenfolge für Optionen gewünscht.

4. Um die Geschwindigkeit mit KIESL noch weiter erhöhen zu können, kann jeder bereits eingegebene Nachrichtentitel gespeichert werden, damit der Titel mit Hilfe der Autovervollständigung beim nächsten Eingeben eines Nachrichtentitels vorgeschlagen werden kann.

Eine weitere Verbesserung, die allerdings in der Technologie von KIELER nicht unterstützt wird, wäre eine Markierung im Diagramm, an welcher Stelle sich der Cursor aktuell im Editor befindet. Dadurch wüsste man durch die Markierung im Diagramm sofort, ob die Zeile, die im Text bearbeitet werden muss, weiter oben oder weiter unten steht.

Als Letztes kann man darüber nachdenken, aus Java-Code automatisch Sequenzdiagramme generieren zu lassen, indem man den Java-Code in KIESL transformiert und durch KIESL dann ein Diagramm erzeugt. Das könnte man erreichen, indem jede Klasse oder Instanz einer Klasse zu einer Lebenslinie transformiert wird. Methodenaufrufe innerhalb einer Klasse werden zu eigenen Nachrichten und Methodenaufrufe, die auf andere Klassen zugreifen, finden zwischen den beiden Lebenslinien statt und starten eventuell eine Sequenz. Diese könnte durch den Rückgabewert der Methode mit der zugehörigen Antwortnachricht beendet werden.

Anhang A

Anhang

In diesem Kapitel werden die Daten der Studie aus Kapitel 5 angegeben. Zu den Daten gehören die Zeiten von jedem Teilnehmer pro Aufgabe und die Bewertungen der Skala. Darüber hinaus wird die Syntax von KIESL an zwei weiteren Beispielen gezeigt.

Tabelle A.1. Rohdaten der Studie

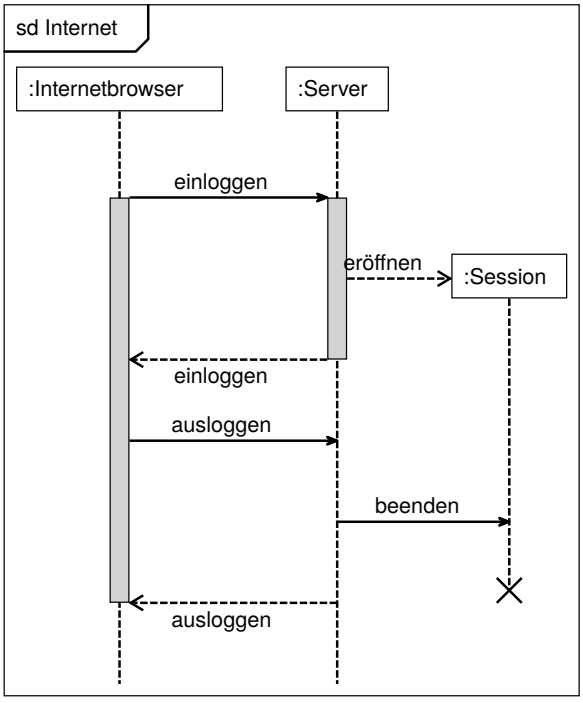
Editor	A1	A2	A3	A4	Bewertung
KIESL	3:48	3:33	7:19	3:43	9
	4:13	3:44	6:34	3:31	8
	4:52	4:09	8:24	3:48	9
	4:22	4:22	6:27	5:01	8
WebSequenceDiagrams	7:52	6:29	12:57	6:22	4
	5:37	4:29	8:46	4:14	8
	5:55	5:04	6:25	2:59	8
	8:37	6:28	8:28	4:42	6
Visual Paradigm	6:10	5:07	6:50	9:52	4
	4:22	5:06	7:27	5:51	8
	3:56	5:30	6:56	3:56	7
	5:03	3:57	6:20	5:12	6
Papyrus	8:15	6:42	9:13	9:43	7
	8:48	7:33	10:10	5:28	6
	7:20	6:18	9:53	5:18	7
	9:18	5:34	10:06	9:16	3
Quick Sequence Diagram Editor	3:40	3:12	5:55	2:34	3
	3:41	3:43	5:35	3:25	7
	5:08	6:01	4:38	3:27	6
	7:35	5:01	6:15	3:53	4

Die Syntax in KIESL zu der Abbildung 1.1:

```
sequenceDiagram "Musik abspielen"

lifeline ":Person" as pers
lifeline ":MediaPlayer" as media
lifeline "Hitparade:Playlist" as hit
lifeline "Song[0]:mp3Filelink" as song0

pers sync "erstellePlaylist" to media
        sourceStartExec
        targetStartExec
media create "erstellen" to hit
media response "erstellePlaylist" to pers
        sourceEndExec
        targetEndExec
pers async "hinzufügenSong(0)" to media
media async "hinzufügen(Songname)" to hit
hit create "erstellen" to song0
pers async "wiedergeben" to media
        targetStartExec
media sync "abspielen" to hit
        targetStartExec
hit sync "abspielen" to song0
        targetStartExec
song0 response "abspielen" to hit
        sourceEndExec
hit response "abspielen" to media
        sourceEndExec
        targetEndExec
media destroy
hit destroy
song0 destroy
```




```
sequenceDiagram "Internet"

lifeline ":Internetbrowser" as i
lifeline ":Server" as ser
lifeline ":Session" as ses

i sync "einloggen" to ser
        sourceStartExec
        targetStartExec
ser create "eröffnen" to ses
ser response "einloggen" to i
        sourceEndExec
i sync "ausloggen" to ser
ser sync "beenden" to ses
ser response "ausloggen" to i
ses destroy
```

Abbildung A.1. Ein weiteres Sequenzdiagramm mit zugehöriger KIESL Syntax

Literatur

[HDM+13] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer und Owen O'Brien. SC-Charts: Sequentially Constructive Statecharts. Presentation at Synchronous Programming (SYNCHRON'13), Schloss Dagstuhl, Germany. Nov. 2013.

[Hoo13] Gregor Hoops. „‚Automatic Layout of UML Sequence Diagrams'". Diploma thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Apr. 2013.

[JRH+04] Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler und Stefan Queins. UML 2 glasklar. Bd. 1. Hanser München/Wien, 2004.

[Pur12] Helen C Purchase. Experimental human-computer interaction: a practical guide with visual examples. Cambridge University Press, 2012.

[SFH09] Miro Spönemann, Hauke Fuhrmann und Reinhard von Hanxleden. Automatic Layout of Data Flow Diagrams in KIELER and Ptolemy II. Technical Report 0914. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Juli 2009.

[SSH13] Christian Schneider, Miro Spönemann und Reinhard von Hanxleden. „Just model! – Putting automatic synthesis of node-link-diagrams into practice". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*. San Jose, CA, USA, 15–19 09 2013, S. 75–82. DOI: 10.1109/VLHCC.2013.6645246.