Entwicklung eines GUI für die Konfiguration der Software-Komponente zur Systemprozessüberwachung und -kontrolle in einer psychologischen Versuchsumgebung

Diplomarbeit

eingereicht	Institut für Informatik Lehr- und Forschungsgebiet Softwaretechnik Mathematisch-Naturwissenschaftliche Fakultät II Humboldt-Universität zu Berlin
von	Esther Fuhrmann
am	13. Oktober 2010
Gutachter	Prof. Dr. sc. nat. Klaus Bothe Prof. Dr. sc. nat. Hartmut Wandke
Betreuerin	Dipl.-Psych. Saskia Kain

ABSTRACT

This diploma thesis is part of the project ATEO (ArbeitsTeilung Entwickler-Operateur, Division of Labour between Developer and Operator), which contributes to the research training group prometei (Prospektive Gestaltung von Mensch-Technik-Interaktion) at Technische Universität Berlin. The goal of the project ATEO is to research the collaboration between operators and developers of automation and assistance systems for the monitoring and control of complex and dynamic technical systems. In particular it will be investigated if there are tasks or classes of tasks which are generally better solved by automatic systems or operators.

To answer this question, a socially augmented microworld (SAM) was developed to serve as a test environment. This environment consists of two persons together with a tracking task, which shall be a simulation of a complex dynamic technical process. The complexity is added by the two persons, which are an unpredictable component of the system. To research the above question SAM can be assisted by an operator or automatic systems.

This diploma thesis is part of the development of automation project. Its goal is to provide the test supervisor with a graphical user interface (GUI), which is easy and intuitive to handle. Using the GUI he or she should be able to compose and configure complex automatic systems. Furthermore he should be able to save and load them and to reuse them as modules for other automatic systems. Therefore a special XML file format was designed.

This thesis describes the realisation of the graphical user interface. To begin with some observations of human factors of software systems are introduced, which play a great role for the development of user friendly interfaces. The question will be answered how software development process models can be brought in line with aspects of human factors of software systems. The main body of this thesis defines the functional requirements of the interface. Afterwards the concrete implementation in Squeak/Smalltalk is presented as well as the integration into the SAM system.

ZUSAMMENFASSUNG

Diese Diplomarbeit ist innerhalb des Projekts ATEO (ArbeitsTeilung Entwickler-Operateur) angesiedelt, welches einen Beitrag zum Graduiertenkolleg prometei (Prospektive Gestaltung von Mensch-Technik-Interaktion) der Technischen Universität Berlin liefert. Ziel des Projekts ATEO ist es, das Zusammenwirken von Operateuren (Bedienern) und Entwicklern von Automatiken und Assistenzsystemen in Bezug auf die Überwachung und Regulierung komplexer und dynamischer technischer Systeme zu untersuchen. Insbesondere interessiert hierbei die Frage, ob sich Aufgaben oder Aufgabenklassen identifizieren lassen, die sich prinzipiell besser durch eine Automatik bzw. einen Operateur lösen lassen.

Um Untersuchungen zur oben beschriebenen Fragestellung durchzuführen, wurde in dem genannten Projekt eine sogenannte Socially Augmented Microworld (SAM) als Untersuchungsumgebung entwickelt. Diese Umgebung wird von zwei Personen zusammen mit einer Trackingaufgabe gebildet, wodurch ein komplexer, dynamischer technischer Prozess abgebildet werden soll. Durch die beiden Personen erhält das System eine unvorhersagbare Komponente, die die nötige Komplexität erzeugt. Für die Untersuchung der oben beschriebenen Frage kann SAM durch einen Operateur oder durch Automatiken unterstützt werden.

Diese Diplomarbeit ist im Bereich der Automatiken-Entwicklung angesiedelt. Ziel der Arbeit ist, dem Versuchsleiter eine einfach und intuitiv zu bedienende grafische Oberfläche zur Verfügung zu stellen. Mit dieser soll er komplexe Automatiken zusammenstellen und konfigurieren können. Zudem soll er sie speichern, laden oder als Baustein für weitere Automatiken wiederverwenden können. Hierfür wurde ein spezielles XML-Speicherformat entworfen.

Im Rahmen dieser Arbeit wird die Umsetzung der grafischen Oberfläche beschrieben. Zunächst werden software-ergonomische Erkenntnisse vorgestellt, die eine Rolle bei der Erstellung nutzerfreundlicher Oberflächen spielen. Vorgehensmodelle der Softwaretechnik werden daraufhin untersucht, inwieweit sie mit den software-ergonomischen Anforderungen in Einklang zu bringen sind. Im Hauptteil der Arbeit werden die funktionalen Anforderungen an die Oberfläche definiert. Anschließend wird die konkrete Implementierung in Squeak/Smalltalk vorgestellt und die Integration in das SAM-System betrachtet.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGSVERZEICHNIS

AKRONYMVERZEICHNIS

AAF	ATEO Automation Framework
API	Application Programming Interface
ATEO	ArbeitsTeilung Entwickler-Operateur
CEM	Concurrent Engineering Model
DOM	Document Object Model
GUI	Graphical User Interface
MWB	Mikroweltbewohner
prometei	Prospektive Gestaltung von Mensch-Technik-Interaktion
SAM	Socially Augmented Microworld
SAX	Simple API for XML
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language

1 EINLEITUNG

1.1 EINBETTUNG

Diese Diplomarbeit ist innerhalb des Projekts ArbeitsTeilung Entwickler-Operateur (ATEO) angesiedelt, welches einen Beitrag zum Graduiertenkolleg Prospektive Gestaltung von Mensch-Technik-Interaktion (prometei) der Technischen Universität Berlin liefert. Ziel des Projekts ATEO ist es, das Zusammenwirken von Operateuren (Bedienern) und Entwicklern von Automatiken und Assistenzsystemen in Bezug auf die Überwachung und Regulierung komplexer und dynamischer technischer Systeme zu untersuchen.[1] Insbesondere interessiert hierbei die Frage, ob sich Aufgaben oder Aufgabenklassen identifizieren lassen, die sich prinzipiell besser durch eine Automatik bzw. einen Operateur lösen lassen. Durch diese Erkenntnis könnten die jeweiligen Stärken und Schwächen bei zukünftigen Systemen bereits während der Planung berücksichtigt werden.

Sowohl Entwickler als auch Operateure haben ihre besonderen Ressourcen und Einschränkungen (Wandke, 2003; Krinner und Gross, 2005). Die Entwickler haben vergleichsweise viel Zeit sich zu überlegen, welche Situationen eintreten könnten und wie eine Automatik darauf reagieren könnte. Ihr Nachteil ist jedoch, dass sie nur hypothetisch über die Situationen nachdenken und dabei nie alles antizipieren können, was später wirklich passiert. Dazu kommt, dass sie u. U. kein ausreichendes Sachverständnis von dem Bereich haben, für den sie entwickeln. Entwickler können nicht unmittelbar während des Betriebs in ein System eingreifen, sondern nur vermittelt über Automatiken, die sie aus ihren Überlegungen heraus implementieren.

Operateure hingegen haben typischerweise ein großes Wissen auf dem fraglichen Gebiet und sie handeln unmittelbar in der konkreten Situation; sie müssen sich also keine Szenarien ausdenken. Dafür müssen sie aber oft sehr schnell Entscheidungen treffen und eingreifen. Hierfür ist es nötig, dass sie sich schnell einen präzisen Überblick über die Situation und ihre Handlungsoptionen verschaffen können.

1 Beispiele sind u. a. Kraftwerkssteuerungen, Produktionsprozesse oder Flugzeugsteuerung. Viele der anfallenden Aufgaben wie Herunterfahren von Systemen, Nachregelung bei Normabweichungen oder Bremsen während einer Landung können prinzipiell sowohl manuell durch einen Operateur (im Falle des Fliegens einen Piloten) als auch automatisiert durchgeführt werden. Im automatisierten Fall muss sich zuvor ein Entwickler Gedanken über die Eingriffsmöglichkeiten gemacht haben.

Um Untersuchungen zur oben beschriebenen Fragestellung durchführen zu können, wurde in dem genannten Projekt bereits eine sogenannte Socially Augmented Microworld (SAM) als Untersuchungsumgebung entwickelt. Diese ist durch Bothe, Hildebrandt und Niestroj (2009) spezifiziert und wird in Abschnitt 2.3 näher vorgestellt. In dieser Umgebung sollen zwei Personen gemeinsam mit je 50 % Steueranteil ein rundes Objekt eine Strecke entlang zum Ziel führen. Es handelt sich dabei um eine Trackingaufgabe. Die beiden Personen zusammen mit der Trackingaufgabe bilden SAM, welches einen komplexen, dynamischen technischen Prozess abbilden soll (Wandke und Nachtwei, 2008). Für die Untersuchung der oben beschriebenen Fragestellung kann SAM entweder durch einen Operateur oder aber durch Automatiken unterstützt werden.

Diese Diplomarbeit ist im Bereich der Automatiken-Entwicklung angesiedelt.

1.2 MOTIVATION

Wie oben bereits beschrieben, greifen Entwickler über Automatiken vermittelt in ein System ein, hier konkret in das SAM-System. Nun ist es aber so, dass nicht nur eine einzige, sondern möglicherweise eine Vielzahl von Automatiken vorstellbar ist, die in das SAM-System integriert werden könnten (Lenkautomatik, Bremsautomatik, Kurvenautomatik, ...). Im Rahmen der parallel entstehenden Diplomarbeit von Michael Hasselmann wurde ein Framework entwickelt, um solche Automatiken auf Basis kleiner Elementarbausteine[2] zu erstellen (Hasselmann, in Bearb.). Um mit diesem direkt zu arbeiten, werden jedoch Programmierkenntnisse in der Entwicklungsumgebung Squeak/Smalltalk sowie detaillierte Kenntnis der Architektur des Frameworks benötigt. Diese Kenntnis zu erlangen, ist mit hohem Aufwand verbunden. Zudem sind in der Programmierung unerfahrene Benutzer mit dieser Aufgabe überfordert. Aufgrund dieser Situation müsste jedes Mal, wenn eine neue Automatik erstellt werden soll, ein Programmierer hinzugezogen werden, der das System hinreichend gut kennt oder sich einarbeiten muss. Hier soll diese Arbeit Abhilfe schaffen.

1.3 ZIEL DER ARBEIT

Ziel dieser Arbeit ist es, eine grafische Oberfläche, also ein Graphical User Interface (GUI), für die Konfiguration von Automatiken zur Prozessüberwachung und -führung zu erstellen.

2 Elementarbausteine realisieren eine einzige klar abgegrenzte Automatikfunktion wie z. B. Bremsen, Spurhalten, Hinweise anzeigen o. ä.

Die Konfiguration wird vom Versuchsleiter vor einer Untersuchungsreihe durchgeführt und beinhaltet die Erstellung und Aktivierung der benötigten Automatiken. Die grafische Oberfläche soll ihn dabei unterstützen, indem sie es ermöglicht, auf einfache und intuitive Weise Automatiken neu zu erstellen und abzuspeichern sowie abgespeicherte Automatiken zu laden und zu verändern. Für den Aufbau von Automatiken soll der Benutzer Basiselemente (sozusagen kleinste Automatiken mit nur einer elementaren Aufgabe) zur Verfügung gestellt bekommen, die er zu komplexeren Automatiken zusammenbauen kann. Ferner kann er das Verhalten der Automatiken beeinflussen, indem er Eigenschaften dieser Basiselemente verändert. Die Umsetzung wird, wie bei dem bisher vorhandenen Untersuchungssystem auch, in der Entwicklungsumgebung und Programmiersprache Squeak/Smalltalk erfolgen.

Diese Arbeit baut auf anderen im Projekt ATEO entstandenen Arbeiten zum Untersuchungssystem auf, wobei speziell die parallel entstehende Arbeit zum Automatiken-Framework von Michael Hasselmann zu nennen ist (Hasselmann, in Bearb.). Diese wird die Objekte und Strukturen liefern, auf die die grafische Repräsentation einer Automatik letztlich abgebildet werden muss, um mit dem SAM-System interagieren zu können. Ebenfalls relevant sind die thematisch ähnlich gelagerten Arbeiten von Hermann Schwarz und Christian Leonhard, die das Pendant zu diesem Automatiken-GUI, nämlich die grafische Oberfläche für den Operateursarbeitsplatz, entwickelt haben (Schwarz, 2009; Leonhard, in Bearb.). Darüber hinaus sollen auch Erkenntnisse der Software-Ergonomie zum GUI-Design berücksichtigt werden.

1.4 GLIEDERUNG DER ARBEIT

In Kapitel 2 werden zunächst die technischen Voraussetzungen beleuchtet, auf denen die in dieser Arbeit beschriebene Entwicklung basiert.

Das folgende Kapitel 3 beschäftigt sich ausführlich mit der software-ergonomischen Theorie, welche im Rahmen dieser Arbeit Anwendung findet. Hierzu gehören neben einer sorgfältigen Definition der zentralen Begriffe auf Basis relevanter Standards auch die Darstellung etablierter Gestaltungsrichtlinien. Abschließend widmet sich das Kapitel der Frage, ob und wie der Erfolg einer software-ergonomisch orientierten Entwicklung gemessen werden kann und wie Vorgehensmodelle der Softwaretechnik mit den software-ergonomischen Anforderungen vereinbar sind.

Kapitel 4 beschreibt den Entwicklungsprozess, aus welchem das in dieser Arbeit erstellte Automatiken-GUI hervorgegangen ist. Hier werden zunächst die identifizierten Anforderungen und

Anwendungsfälle dokumentiert und aus diesen ein objektorientiertes Analysemodell entworfen. Im Anschluss wird die daraus resultierende Implementierung der Anwendung detailliert dargelegt. Es folgen Betrachtungen zur Umsetzung der zuvor erläuterten software-ergonomischen Gestaltungsrichtlinien sowie zur Integration der Arbeit in die Untersuchungsumgebung des Projekts ATEO.

Im abschließenden Kapitel 5 werden die Ergebnisse der vorliegenden Arbeit noch einmal zusammengefasst. Auch werden mögliche Weiterführungen der Arbeit diskutiert. Hierbei handelt es sich insbesondere um Ideen, welche während der Entwicklung entstanden sind, aufgrund des begrenzten Zeitrahmens jedoch nicht mehr umgesetzt werden konnten.

2 VORAUSSETZUNGEN

2.1 SMALLTALK UND SQUEAK

In diesem Abschnitt soll kurz vorgestellt werden, woher Squeak kommt und was es ist. Da es sich bei Squeak um eine Implementierung von Smalltalk-80 handelt, wird zunächst einmal auf die Programmiersprache Smalltalk im Allgemeinen und danach auf Squeak im Besonderen eingegangen.

Die Anfänge von Smalltalk reichen zurück bis in die frühen 70er Jahre, als Alan Kay am Xerox PARC die Idee einer Programmiersprache hatte, die selbst für Kinder einfach zu erlernen sein sollte (Sharp, 1997). In der Folge entstanden in einem evolutionären Prozess und unter Einfluss anderer Programmiersprachen wie SIMULA-67 und LOGO (Hunt, 1997) eine Reihe von Smalltalk-Versionen. Schließlich erfolgte um 1980 herum ein Neuentwurf, der unter dem Namen Smalltalk-80 veröffentlicht wurde. Auf dieser Version basieren praktisch alle heutigen Implementierungen von Smalltalk, so auch Squeak. Der Einsatzbereich von Smalltalk liegt hauptsächlich in der Forschung und im Prototyping. Es findet aber auch Verwendung in Geschäftsanwendungen namhafter Firmen wie Hewlett-Packard oder Texas Instruments (Sharp, 1997).

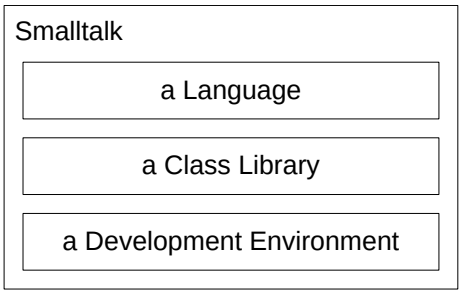
Smalltalk ist eine streng objektorientierte Programmiersprache mit einem sehr kleinen Sprachumfang. Die Sprache selbst bietet prinzipiell erst einmal nicht mehr Möglichkeiten als das Definieren von Variablen, die Zuweisung von Objekten an diese und das Senden von Nachrichten an Objekte (Lewis, 1995). Allerdings gehört zu einer Smalltalk-Implementierung auch immer eine umfangreiche Klassenbibliothek, die einen großen Fundus von wiederverwendbaren Klassen und Methoden bereitstellt. Durch diese wird auch die gesamte Basisfunktionalität realisiert, welche sonst meist Teil der Programmiersprache selbst ist.[3] Dieser Ansatz bringt gegenüber anderen Sprachen wie C oder Java einige Eigenheiten mit sich, die einer gewissen Eingewöhnung bedürfen. So sind z. B. bedingte Verzweigungen als Nachrichten an boolesche Objekte oder Zählschleifen als Nachrichten an Zahlenmengen realisiert. Ein Beispiel hierfür zeigt Listing 1.

3 Hierzu gehören u. a. bedingte Verzweigungen oder Schleifen, um nur einige zu nennen.

```
1 (1 to: 5) do: [:idx |
2   Transcript show: idx;cr.
3 ]
```

Listing 1: Zählschleife – Die Nachricht `do:` wird an eine Zahlenmenge geschickt, für jedes Element der Menge wird der Codeblock ausgeführt.

Darüber hinaus bietet jede Smalltalk-Implementierung eine fensterbasierte integrierte Entwicklungsumgebung, die zahlreiche Werkzeuge zur Erstellung und zum Testen von Programmen bereitstellt (Abbildung 1). Lewis (1995) fasst, was Smalltalk ist, so zusammen:



Bei Squeak handelt es sich um eine relativ moderne Implementierung von Smalltalk-80, eine „updated reference implementation for Smalltalk-80“ (Ingalls, Kaehler, Maloney, Wallace und Kay, 1997, S. 325). Sie wird seit Dezember 1995 von einer Gruppe um Alan Kay bei Apple Computer und mittlerweile von einer offenen Community[4] entwickelt. Als Teil der oben bereits erwähnten Klassenbibliothek, welche Teil eines jeden Smalltalk-Systems ist, bringt Squeak ein Grafiksystem namens Morphic mit. Auf dieses wird im folgenden Abschnitt 2.2 näher eingegangen.

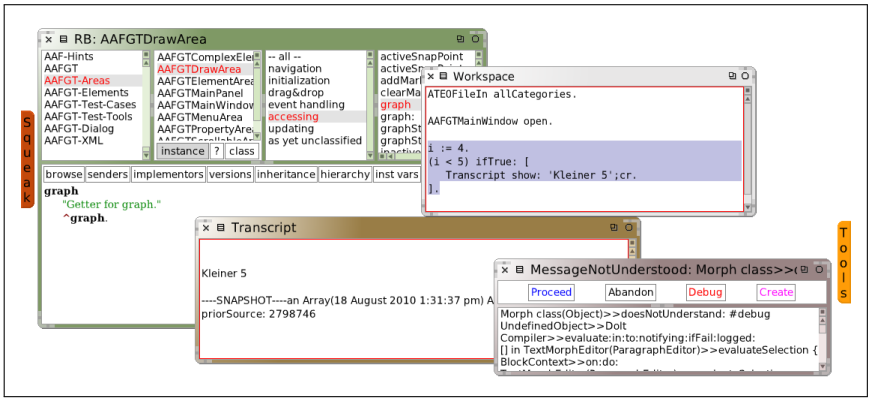


Abbildung 1: Squeak/Smalltalk ist nicht nur eine Programmiersprache, sondern auch eine integrierte Entwicklungsumgebung.

4 http://www.squeak.org/

Squeak ist im Projekt ATEO das Smalltalk-80-System der Wahl, weil es kostenlos und für verschiedene Plattformen[5] verfügbar ist. Zudem lässt die Lizenz eine Nutzung, wie sie im Projekt erfolgt, zu.

2.2 MORPHIC

Wie bereits im letzten Abschnitt erwähnt, bringt Squeak ein Grafiksystem zur Erstellung von Benutzeroberflächen namens Morphic mit. In diesem Abschnitt sollen die wesentlichen Eigenschaften dieses Systems sowie die zugrunde liegenden Entwurfsprinzipien dargelegt werden.

Erfunden wurde Morphic von John Maloney und Randy Smith in den Sun Microsystem Laboratories (Maloney, 2002). Ursprünglich wurde es als Grafiksystem für die Programmiersprache Self 4.0 entwickelt. Für Squeak wurde das System unter Beibehaltung sämtlicher Konzepte völlig neu implementiert.

Ziel von Morphic war es, den Umgang mit grafischen Objekten für den Programmierer so einfach wie möglich zu gestalten. Um das zu erreichen, stellt das System als zentrale Abstraktion eine vollständig funktionierende Basisklasse `Morph` bereit, auf der alles Weitere aufbaut. Außerdem nimmt es dem Programmierer sämtliche Aspekte ab, die mit der Behandlung von Ereignissen wie z. B. Tastatureingaben oder Mausklicks durch Morphic-Objekte zu tun haben. Auch um das Aktualisieren der Bildschirmanzeige braucht er sich nicht zu kümmern, da das Morphic-System selbst über Veränderungen wacht und veränderte Bildschirmbereiche bei Bedarf neu zeichnet.

Will nun ein Programmierer ein eigenes Grafikelement, ein sogenanntes Widget, erstellen, braucht er zunächst nicht mehr zu tun als seine Widget-Klasse von der `Morph`-Basisklasse abzuleiten. Damit hat er dann bereits ein vollständig funktionierendes Morphic-Objekt zur Hand, wie es in Abbildung 2 zu sehen ist.

Abbildung 2: Trotz seines unscheinbaren Aussehens ist ein Objekt der Basisklasse `Morph` bereits voll funktionsfähig und lässt sich aufheben, ablegen, in Größe oder Farbe ändern und vieles mehr.

Durch vordefinierte Parameterwerte für essenzielle Eigenschaften sowie Standardimplementierungen für die Behandlung von Eingabe-Ereignissen ist es sofort einsetzbar. Nun kann er nach und nach Anpassungen vornehmen, um das Grafikobjekt

5 darunter verschiedene Versionen von Windows und Linux

zu erhalten, das er im Sinn hat. Er kann z. B. einzelne Eigenschaften wie Farbe oder Größe verändern, bereits existierende Morphic-Objekte zu neuen assemblieren oder aber auch ein vollständig neues Aussehen erzeugen. Für Letzteres muss er die Methode `drawOn:` verändern. Diese ist für das Zeichnen des Morphs verantwortlich. Durch Überschreiben von Methoden, die Ereignisse wie Tastendrücke oder Mausaktionen behandeln, kann er die Interaktion des Benutzers mit dem neuen Morphic-Objekt ganz nach seinen Wünschen gestalten. Dadurch, dass bereits die leere Unterklasse von `Morph` instanziiert und angezeigt werden kann, kann er den Effekt seiner Anpassungen in jedem Stadium direkt verfolgen und testen.

Wie von Maloney (2002) erläutert wird, wurden beim Entwurf des Morphic-Systems vor allem drei grundlegende Prinzipien berücksichtigt: Konkretheit, Direktheit und Lebendigkeit.[6]

Dabei bedeutet Konkretheit, dass die Objekte möglichst weitgehend über die Eigenschaften verfügen, die sich auch bei Gegenständen der realen Welt finden lassen. So haben sie Form, Farbe und Größe und können nach Belieben aufgehoben und abgelegt, gestapelt, gedreht oder neu platziert werden. Direktheit bedeutet dabei, dass Operationen nicht über weit vom Objekt entfernte Menüs aufgerufen, sondern durch direkte Interaktion mit dem Objekt ausgelöst werden. Wo das nicht ohne Weiteres möglich ist, sind sie über den sogenannten Halo (Abbildung 3) zugänglich, den jedes Morphic-Objekt besitzt. Der Halo besteht aus mehreren Symbolen, die um das jeweilige Objekt angeordnet sind und verschiedene Aktionen auslösen können.

Abbildung 3: Über den Halo sind zahlreiche Aktionen und Eigenschaften eines jeden `Morph`-Objekts zugreifbar.

Unter Lebendigkeit wird vor allem die Möglichkeit verstanden, Morphic-Objekte zu animieren. Dazu gibt es einen sogenannten Stepping-Mechanismus, der von Maloney (2002, S. 33) als der „Herzschlag“ des Systems bezeichnet wird. In festgelegten Zeitabständen wird an dem Morph, der animiert werden

6 im Original: concreteness, directness and liveness

soll, eine bestimmte Methode aufgerufen. In dieser kann der Programmierer festlegen, wie sich die Erscheinung des Morphs über die Zeit verändern soll. So kann er hier z. B. den Morph verschieben, ihm wechselnde Farben zuweisen oder ihn sich ändernde Werte irgendeiner Datenquelle anzeigen lassen.

Abschließend soll noch darauf hingewiesen werden, dass das Morphic-System nicht nur dem Programmierer für seine Programme zur Verfügung steht, sondern auch für die komplette Oberfläche des Squeak-Systems zum Einsatz kommt.

2.3 SAM

Für die Untersuchung der in Abschnitt 1.1 dargelegten Fragestellung des Projekts ATEO wird eine Versuchsumgebung benötigt, welche die Möglichkeit bietet eine Prozessführungsaufgabe durchzuführen. Dabei muss der simulierte Prozess hinreichend komplex sein und darf nicht deterministisch ablaufen (Wandke und Nachtwei, 2008). Täte er das, so könnte eine Automatik alle Eingriffe exakt im Voraus berechnen. Je nach Fülle der Variationen könnte sich auch ein Operateur auf jeden nächsten Schritt einstellen. Somit wären die Fähigkeiten, die untersucht werden sollen, nicht mehr gefragt. Der Entwickler müsste mögliche Ereignisse nicht mehr antizipieren und der Operateur nicht mehr in kurzer Zeit auf neue Situationen reagieren.

Der Prozess darf allerdings auch nicht absolut zufällig ablaufen (Wandke und Nachtwei, 2008). In diesem Fall wären überhaupt keine sinnvollen Korrekturen durch eine Automatik mehr möglich, da der Entwickler Ereignisse nicht im Vorfeld antizipieren könnte.

Es wird also ein komplexes System benötigt, welches sich realistisch verhält und bei dem die nahe Zukunft mit einer gewissen Wahrscheinlichkeit, aber nie mit Sicherheit, vorhersehbar ist. Nur in einer solchen Umgebung kann die Frage untersucht werden, welche Aufgaben besser von einem Operateur und welche besser von einer Automatik gelöst werden können. Zudem sind die Ergebnisse nur unter solchen Umständen praxisrelevant, da auch reale technische Systeme im Normalbetrieb einigermaßen vorhersehbar sind, durch Störungen aber jederzeit unerwartetes Verhalten zeigen können.

Eine solche Versuchsumgebung stellt die im Projekt ATEO entwickelte Socially Augmented Microworld (SAM) bereit. Hierbei handelt es sich um eine Trackingaufgabe, in der zwei Personen mittels Joystickeingaben kooperativ ein Objekt eine Strecke entlang zum Ziel führen müssen (Abbildung 4). Dabei erhält jede der Personen, der sogenannten Mikroweltbewohner (MWB), eine andere Instruktion: ein MWB soll die Strecke möglichst schnell, der andere möglichst genau absolvieren. Da jeder

MWB nur seine Instruktion kennt und wahrscheinlich annimmt, der andere habe die gleiche erhalten, ist zu erwarten, dass die Steuereingaben konkurrierenden Zielen dienen. Somit wird die Verrechnung beider Eingaben zu einer Steuerung mit gewissen Fehlern führen, welche erklärbar sind, jedoch nicht exakt vorhergesehen werden können. Zu beachten ist, dass die MWB nicht die Operateure sind, sondern als die Komponente des SAM-Systems betrachtet werden, welche die benötigte Variation im Systemverhalten erzeugt.

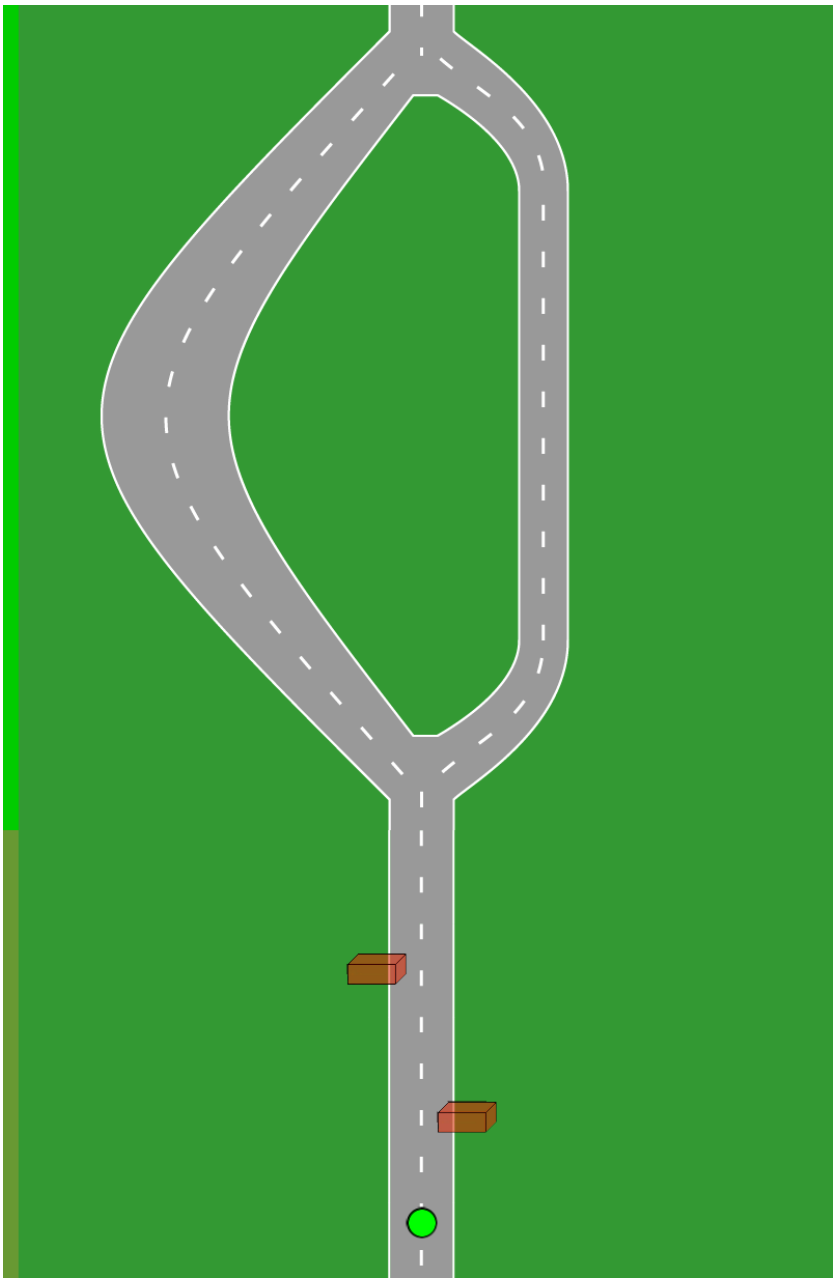


Abbildung 4: Trackingaufgabe: Zwei Personen steuern gemeinsam ein Objekt eine Strecke entlang zum Ziel.

Bei einer Untersuchung werden die MWB nun entweder von einem Operateur, der sie von einem anderen Raum aus beobachtet, oder aber von einer im Vorfeld entwickelten Automatik bei ihrer Aufgabe durch Hinweise oder auch direkte Eingriffe unterstützt. Ziel ist es jeweils, das Objekt insgesamt möglichst schnell ins Ziel zu bringen und dabei möglichst wenig von der Strecke abzuweichen. Das Ausmaß, in dem dieses Ziel erreicht werden kann, dient als Maß für die Leistung des Operateurs bzw. der Automatik, welches sich im Vergleich auswerten lässt.

Der gesamte Untersuchungsaufbau ist in Abbildung 5 schematisch dargestellt.

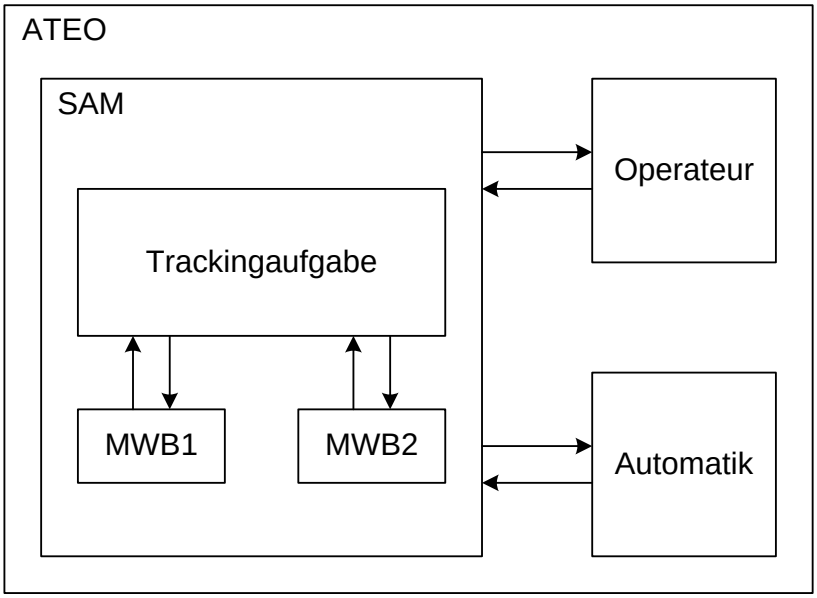


Abbildung 5: Schematische Darstellung der ATEO-Untersuchungsumgebung.

2.4 XML

Die Extensible Markup Language (XML) kommt in dieser Arbeit zum Einsatz, um die mit dem Automatiken-GUI erstellten Automatiken zur späteren erneuten Bearbeitung und zur Verwendung mit SAM zu speichern. Daher soll kurz darauf eingegangen werden, was XML ist und warum es sich für den genannten Verwendungszweck gut eignet.

Bei XML handelt es sich um eine vom World Wide Web Consortium[7] (W3C) spezifizierte Datenbeschreibungssprache. In der Literatur gilt sie bereits als die Standardsprache zur Beschreibung und zum Austausch von Inhalten (Balzert und Priemer, 2008; Vonhoegen, 2005). Sie ist universell einsetzbar, da das

7 http://www.w3.org/

durch sogenannte Tags[8] gebildete Vokabular frei und für den jeweiligen Einsatzzweck passend wählbar ist. Dennoch sind XML-Dateien maschinenlesbar, da sie wohldefinierten syntaktischen Regeln folgen. Die Daten in einer XML-Datei sind hierarchisch gegliedert und bilden eine Baumstruktur.

Die Beschreibung und Strukturierung der Daten erfolgt durch sogenannte XML-Elemente (Listing 2). Ein XML-Element wird durch ein Start-Tag und ein Ende-Tag begrenzt. Es kann Daten enthalten, weitere Unterelemente umschließen oder auch leer sein. Strikt zu beachten ist, dass zu jedem Start-Tag ein passendes Ende-Tag vorhanden sein muss, und dass die Verschachtelung streng hierarchisch erfolgen muss. Zusätzlich zu seinem Inhalt kann jedes Element durch Attribute näher beschrieben werden.

```
1  <?xml version="1.0"?>
2  <rootElement>
3      <element>
4          SomeCharacterData
5      </element>
6      <element>
7          <anotherElement>
8              SomeCharacterData
9          </anotherElement>
10     </element>
11     <element>
12         <emptyElement>
13         </emptyElement>
14         SomeCharacterData
15     </element>
16 </rootElement>
```

Listing 2: Exemplarischer XML-Code

Das Element, welches alle weiteren umschließt, bildet die Wurzel der bereits erwähnten Baumstruktur. Der Baum zu Listing 2 ist in Abbildung 6 dargestellt.

Entspricht ein XML-Dokument dem beschriebenen Aufbau, so heißt es wohlgeformt. Wohlgeformtheit ist die Minimalanforderung, die erfüllt sein muss, damit es sich um ein korrektes Dokument handelt.

Es können jedoch darüber hinausgehende Einschränkungen gemacht werden. Diese werden in einer Dokumenttypen-Definition (DTD, Document Type Definition) oder alternativ als XML-Schema festgelegt. Beide Formen ermöglichen es, genau zu definieren, welche Elemente erlaubt sind, ob sie Attribute besitzen und in welcher Reihenfolge, Verschachtelung und Häufigkeit sie

8 In der deutschen Literatur wird sowohl das englische Wort *Tag* (Vonhoegen, 2005) als auch das deutsche Wort *Markierung* (Balzert und Priemer, 2008) verwendet. Da im Alltag im Zusammenhang mit XML das englische Wort gängiger ist, soll es auch hier verwendet werden.

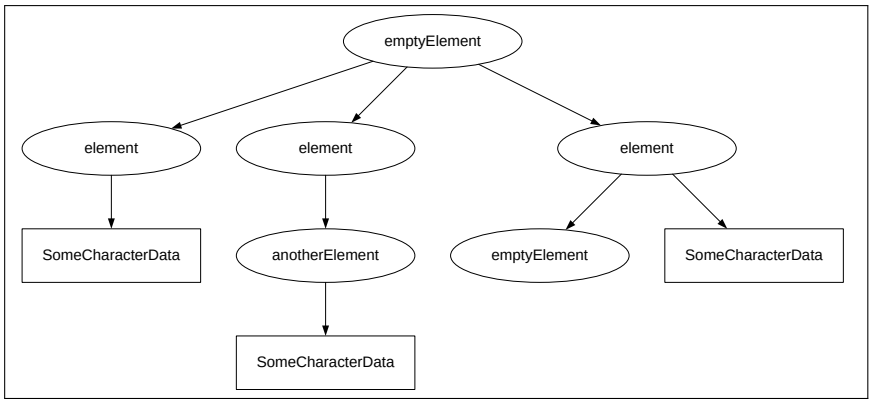


Abbildung 6: Graph zu Listing 2

auftreten müssen bzw. dürfen. XML-Schema ermöglicht darüber hinaus die Festlegung von Datentypen und Wertebereichen.

Ist ein XML-Dokument wohlgeformt und entspricht der angegebenen DTD oder dem angegebenen XML-Schema, so wird von einem gültigen oder validen XML-Dokument gesprochen.

Um die in einem XML-Dokument abgelegten Daten z. B. in einem Programm zu verwenden, muss das Dokument von einem sogenannten Parser eingelesen werden. Es gibt hierfür zwei Varianten, die beide für diverse Programmiersprachen zur Verfügung stehen und im Folgenden kurz beschrieben werden sollen.

Der erste Vertreter ist SAX[9] (Simple API for XML). Der SAX-Parser liest die XML-Daten sequenziell ein. Wenn bestimmte Situationen eintreten, also z. B. ein XML-Dokument oder ein Element startet oder endet, löst er ein Ereignis aus. Dieses wird durch zuvor festgelegte Behandlungsroutinen verarbeitet. Bei diesem Parsertyp wird nie das gesamte XML-Dokument im Arbeitsspeicher gehalten, sondern immer nur der gerade verarbeitete Teil.

Der andere Ansatz nennt sich DOM (Document Object Model) und basiert auf einer Empfehlung des W3C (Vonhoegen, 2005). Ein DOM-Parser liest das XML-Dokument ein und legt es komplett als Baumstruktur im Hauptspeicher ab. Durch diese Struktur kann dann beliebig navigiert werden und Knoten des Baumes können gelesen und auch verändert werden.

In dieser Arbeit wird ein SAX-Parser verwendet, da er durch seine Arbeitsweise die Verarbeitung beliebig großer Dokumente zulässt.

Die Wahl von XML als Speicherformat für die Automatiken hatte mehrere Gründe. Die Graphen, die die Automatiken repräsentieren, sind in hohem Maße strukturierte Daten. Daher war es naheliegend, eine Speicherform zu wählen, in der diese Struktur auf einfache Weise abgebildet werden kann. Außerdem

9 http://www.saxproject.org/

bringt die verwendete Programmierumgebung Squeak bereits sowohl einen SAX- als auch einen DOM-Parser mit, ebenso wie Klassen, durch die das Schreiben von XML-Dateien unterstützt wird. Zudem ist die Möglichkeit vorteilhaft, das verwendete Speicherformat durch das Erstellen einer DTD oder eines XML-Schemas formal spezifizieren zu können.[10] Zuletzt sei noch genannt, dass auch zahlreiche andere Programmiersprachen XML-Parser zur Verfügung stellen, sodass dieses Speicherformat unverändert beibehalten werden kann, wenn die Automatiken in anderen Implementierungen des Untersuchungssystems[11] zum Einsatz kommen sollen.

10 Leider können die von Squeak bereitgestellten Parser weder DTD noch XML-Schema verarbeiten.

11 Es gibt bereits eine teilweise Reimplementierung der SAM-Komponente in Java, die in der Diplomarbeit von Michael Hildebrandt (2009) beschrieben wird.

3 THEORETISCHE AUSEINANDERSETZUNG

Ziel dieser Arbeit ist die Entwicklung einer Anwendung, die nicht nur die notwendige Funktionalität aufweist, um Automatiken zu erstellen und zu konfigurieren, sondern die den Anwender auch bestmöglich bei der Bewältigung dieser Aufgabe unterstützt. Daher ist es notwendig, sich mit den Möglichkeiten einer benutzerorientierten Entwicklung auseinanderzusetzen.

Im folgenden Abschnitt (3.1) sollen daher zunächst einmal das Ziel der Software-Ergonomie sowie damit zusammenhängende Begriffe definiert werden.

In den beiden anschließenden Abschnitten (3.2 und 3.3) werden allgemeine Kriterien vorgestellt, die ein Dialog einhalten sollte, und konkrete Gestaltungsregeln für einzelne Aspekte der zu entwickelnden Anwendung erläutert. Die Gestaltungsregeln und Dialogkriterien stammen aus dem Gebiet der Ingenieurpsychologie. Somit bilden sie die Nahtstelle zwischen der Informatik, oder genauer gesagt der Softwaretechnik, auf der einen und der Psychologie auf der anderen Seite.

Im vierten Abschnitt (3.4) wird die Frage erörtert, wie überprüft werden kann, ob die in den Abschnitten 3.2 und 3.3 vorgestellten Regeln und Kriterien tatsächlich zum gewünschten Ziel geführt haben, nämlich zu einem nutzerorientierten, gebrauchstauglichen Entwurf der Software.

Der letzte Abschnitt (3.5) untersucht Vorgehensmodelle der Softwaretechnik daraufhin, wie sie mit den software-ergonomischen Anforderungen in Einklang zu bringen sind.

3.1 SOFTWARE-ERGONOMIE

Das Ziel der Software-Ergonomie ist bei Balzert (2001, S. 485) wie folgt formuliert:

> „Ziel der Software-Ergonomie ist die Entwicklung und Evaluierung gebrauchstauglicher Software-Produkte, die Benutzer zur Erreichung ihrer Arbeitsergebnisse befähigen und dabei ihre Belange im jeweiligen Nutzungskontext beachten."

Nach Balzert (2001) ist die angesprochene Gebrauchstauglichkeit aus Sicht eines Benutzers das wichtigste Kriterium zur Bewertung einer Software.

Dabei meint Gebrauchstauglichkeit nach DIN EN ISO 9241, Teil 11 (DIN 9241-11, 1998, S. 4):

> „das Ausmaß, in dem ein Produkt durch bestimmte Benutzer in einem bestimmten Nutzungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufriedenstellend zu erreichen“

und Nutzungskontext nach selbiger Quelle (S. 4):

> „die Benutzer, Arbeitsaufgaben, Arbeitsmittel (Hardware, Software und Materialien) sowie die physische und soziale Umgebung, in der das Produkt genutzt wird.“

Auch die drei genannten Leitkriterien Effektivität, Effizienz und Zufriedenstellung sind in DIN 9241-11 (1998, S. 4) definiert.

EFFEKTIVITÄT: Die Genauigkeit und Vollständigkeit, mit der Benutzer ein bestimmtes Ziel erreichen.

EFFIZIENZ: Der im Verhältnis zur Genauigkeit und Vollständigkeit eingesetzte Aufwand, mit dem Benutzer ein bestimmtes Ziel erreichen.

ZUFRIEDENSTELLUNG: Freiheit von Beeinträchtigungen und positive Einstellungen gegenüber der Nutzung des Produkts.

Ziel der Gestaltung unter software-ergonomischen Gesichtspunkten ist also, eine Software dahin gehend zu optimieren, dass der Nutzer alle seine Aufgaben vollständig und mit möglichst wenig Aufwand erledigen kann und bei ihm ein positives Gefühl gegenüber der Software-Nutzung entsteht. Um zu erfassen, inwieweit dieses Gestaltungsziel erreicht wurde, ist es notwendig, für die Kriterien Effektivität, Effizienz und Zufriedenstellung Maße festzulegen; Balzert (2001) schlägt vor, die Definition in die Geschäftsprozesse zu integrieren, die zur Spezifikation der Anforderungen erarbeitet werden. Allerdings ist es besonders für das dritte Kriterium schwierig ein Maß zu finden, da Zufriedenstellung in hohem Maße subjektiv ist und stark von den früheren Erfahrungen des einzelnen Benutzers abhängt (Herczeg, 2005). Dennoch gibt es eine Reihe von Kriterien und Empfehlungen, die dabei helfen können, gebrauchstaugliche Software zu entwickeln. Diese werden in den folgenden Abschnitten vorgestellt.

3.2 DIALOGPRINZIPIEN

Nach Wessel (2002, S. 21) ist es „unmöglich, auf Software-Design zu verzichten“, man kann „sich nur entweder für gutes oder

schlechtes, nicht aber für gar kein Design entscheiden“. Daher sollen in diesem Abschnitt Prinzipien zur Dialoggestaltung vorgestellt werden, deren Einhaltung zu gutem Design führen soll, und zwar im Sinne der Erreichung der im vorigen Abschnitt 3.1 erläuterten Ziele (Effektivität, Effizienz und Zufriedenheit). Diese Dialogprinzipien sind Teil der Norm DIN EN ISO 9241 und finden sich in Teil 10, wo unter einem Dialog die „Interaktion zwischen einem Benutzer und einem Dialogsystem, um ein bestimmtes Ziel zu erreichen“ verstanden wird.

Hervorgegangen sind die Prinzipien aus Untersuchungen von Dzida, Herda und Itzfeldt (1978), in denen sie die Teilnehmer dazu befragt haben, was ihnen im Umgang mit einer Software besonders wichtig ist. Da die Teilnehmer für die Arbeitswelt repräsentative Computernutzer waren, ist anzunehmen, dass die Ergebnisse in hohem Maße relevant sind. Die einzelnen Prinzipien werden im Folgenden erläutert.

Aufgabenangemessenheit

Aufgabenangemessenheit bedeutet, dass die Software den Anwender genau bei den Aufgaben unterstützt, die er mit ihr zu bewältigen hat. Dazu gehört zum einen, dass die Software alle notwendigen Funktionen und Informationen bietet, zum andern aber auch, dass sie eben nur genau diese bietet. Sie sollte den Anwender nicht mit unnötigen Funktionen und Informationen belasten oder ablenken (Dahm, 2006). Darüber hinaus genügt nicht das bloße Vorhandensein einer Funktion. Es gehört auch zur Aufgabenangemessenheit, dass gerade häufig verwendete Funktionen über möglichst kurze Wege erreichbar sind (Wessel, 2002). Auch sollten unnötige Schritte vermieden werden (Dahm, 2006), um den Anwender bestmöglich motorisch wie auch kognitiv zu entlasten.

Ebenfalls zur Aufgabenangemessenheit gehört, dass die Software die Kenntnisse und Erfahrungen der Zielgruppe beachtet. Zudem muss das in ihrem Arbeitsfeld übliche Vokabular verwendet werden (Wessel, 2002). Darüber hinaus sollte die Software Unterstützung bei häufig gleichartig ausgeführten Aktionen bieten (Dahm, 2006).

Selbstbeschreibungsfähigkeit

Selbstbeschreibungsfähigkeit einer Software meint, dass der Anwender sich immer darüber im Klaren ist, wie der aktuelle Systemzustand ist. Auch sollte er stets seine Handlungsoptionen und deren Konsequenzen kennen, ohne laufend Handbücher oder andere externe Quellen zurate ziehen zu müssen.

Dazu gehört, dass mögliche oder sinnvolle Aktionen hervorgehoben und unzulässige evtl. ausgeblendet werden. Falls Aktionen Eingaben verlangen, sollte der Anwender Hinweise zu den erwarteten Daten erhalten (Dahm, 2006). Wenn eine Aktion schwerwiegende Konsequenzen nach sich zieht, muss der Anwender hierüber informiert werden. Ebenso sollte er Feedback über das Resultat einer Aktion oder den Status gerade ausgeführter Vorgänge erhalten (Wessel, 2002). Weiterhin unterstützen kontextsensitive Hilfen, z. B. in Form sogenannter Tooltipps, die Selbstbeschreibungsfähigkeit einer Software. Es muss außerdem darauf geachtet werden, dass stets eine allgemein verständliche Sprache und ggf. das Vokabular der Zielgruppe verwendet wird.

Steuerbarkeit

Steuerbarkeit meint, dass der Nutzer die Kontrolle über den Ablauf und die Geschwindigkeit eines Dialogs hat (Dahm, 2006; DIN 9241-10, 1996) und sich „der Software nicht völlig hilflos ausgesetzt“ fühlt (Wessel, 2002, S. 64). Um das zu erreichen, ist es wichtig, dass Aktionen stets vom Nutzer angestoßen werden und die Software nicht von selbst aktiv wird. Weiterhin soll der Anwender die Möglichkeit haben, in einem Dialog nochmals zurückgehen oder ihn jederzeit abbrechen zu können. Auch sollte er Eingaben nicht in einer fest vorgegebenen Reihenfolge vornehmen müssen, wenn es nicht absolut zwingend erforderlich ist (Wessel, 2002). Ferner sollten dem Anwender verschiedene Möglichkeiten zur Verfügung stehen, den Dialog zu steuern oder Eingaben zu machen.

Erwartungskonformität

Damit eine Software erwartungskonform ist, muss sie vor allem konsistent sein. Dabei bedeutet Konsistenz erst einmal, dass Benennungen innerhalb der Software durchgängig verwendet werden und ähnliche Aktionen immer auf die gleiche Art und Weise auszuführen sind (Dahm, 2006). Es reicht jedoch nicht aus, wenn eine Software nur in sich konsistent ist, sondern sie muss auch zu den Erfahrungen des Anwenders passen. Hierbei spielt das gängige Vokabular seines Aufgabenbereichs ebenso eine Rolle wie die Vorerfahrungen, die er schon mit anderer Software gemacht hat (Wessel, 2002).

Die Erwartungen, die der Anwender aufgrund obiger Aspekte hat, können sich dabei auf verschiedene Aspekte beziehen. Dazu gehören u. a. die Benennung und Anordnung von Menüs und Optionen, das Aussehen bestimmter Symbole, die Verwendung bestimmter Tastenkürzel oder der Effekt von Mausaktionen. Um eine solche Vereinheitlichung zu erreichen, existieren

für weitverbreite GUI-Systeme wie Windows, Linux oder Mac OS sogenannte Styleguides, an die sich Softwareentwickler halten sollten, wenn sie für das entsprechende GUI-System entwickeln.

Fehlertoleranz

Fehlertoleranz bedeutet, dass die Software mit Bedienfehlern des Anwenders sinnvoll umgeht, ohne dass es zu Datenverlusten oder anderen ungewollten u. U. teuren oder gar gefährlichen Konsequenzen kommt. Außerdem soll die Software den Anwender beim Erkennen und Korrigieren der Fehler unterstützen (Dahm, 2006). So sollte es z. B. möglich sein, eine fehlerhafte Aktion wie das versehentliche Löschen von Daten zurückzunehmen (Wessel, 2002). Auch sollten Eingabewerte überprüft (Dahm, 2006) und der Anwender im Fehlerfall zur Korrektur aufgefordert werden. Manchmal kann es auch wünschenswert sein, dass eine Software im Fehlerfall automatische Korrekturen vornimmt. Solche automatischen Korrekturen sollten jedoch im Sinne der Steuerbarkeit vom Anwender abschaltbar oder konfigurierbar sein (Wessel, 2002).

Ebenfalls zur Fehlertoleranz einer Software gehört, den Anwender vor Folgen von Fehlern zu schützen, die er nicht zu verantworten hat. So sollten z. B. Datensätze nicht verloren gehen, nur weil ein Speichervorgang aufgrund eines nicht verfügbaren Datenträgers fehlgeschlagen ist.

Noch besser als Fehlertoleranz ist natürlich die Vermeidung von Fehlern durch die Software (Wessel, 2002), da dies jedoch nicht immer möglich ist, bleibt die Fehlertoleranz von großer Bedeutung.

Individualisierbarkeit

Individualisierbarkeit einer Software bedeutet, dass der Anwender sie an seinen Kenntnisstand, seine Arbeitsweise und seine Vorlieben anpassen kann. Ein fortgeschrittener Nutzer braucht z. B. weniger Hinweise durch die Software als ein Anfänger. Auch kann ein Kenner evtl. von Optionen profitieren, die einen Neuling überfordern würden. Ebenso kann es sinnvoll sein, verschiedene Ansichten für die Daten anzubieten, sodass der Anwender die wählen kann, welche er für seine Arbeitsweise und Aufgabe als angemessen empfindet (Wessel, 2002). Die Eingaben sollten nicht nur wahlweise über die Maus oder die Tastatur erfolgen können. Es sollte darüber hinaus möglich sein, dass sich der Anwender für häufig genutzte Funktionen eigene Tastenkürzel, Toolbars oder Menüs definiert. Wiederkehrende Abläufe

sollte er als Makros speichern können. Ggf. kann auch das Angebot verschiedener Ausgabemodalitäten sinnvoll sein.

Lernförderlichkeit

Um das Lernen zu unterstützen, kann es sinnvoll sein, dem Nutzer die Konzepte klarzumachen und die Metaphern zu vermitteln, die hinter der Software stecken, um so seine Modellbildung zu unterstützen (Dahm, 2006). Weiterhin kann das Erlernen durch kontextsensitive Hilfen und Tooltipps erleichtert werden. Es ist auch möglich, dem Anwender bei der Benutzung einer Funktion eine Erläuterung zu geben, wie sie zu verwenden ist. Ein solcher Hinweis sollte jedoch nur beim ersten Mal erscheinen oder zumindest für die Zukunft deaktivierbar sein. Für den Anwender kann es ebenso hilfreich sein, wenn die Software integrierte Schritt-für-Schritt-Anleitungen bereitstellt und wenn er selbst Anmerkungen und Markierungen in Hilfetexten vornehmen kann. Ideal wären adaptive Hilfen, die anhand des Verhaltens des Nutzers erkennen, was er gerade tun möchte und welche Unterstützung er braucht (Wessel, 2002).

Ebenfalls zur Unterstützung der Lernförderlichkeit gehören aber auch Hilfen, die sich außerhalb der Software befinden, wie Online-Tutorien, Handbücher, Schulungsmaterialien etc.

Nachdem die verschiedenen Dialogprinzipien nun erläutert wurden, bleibt noch darauf hinzuweisen, dass es oft nicht möglich sein wird, alle Prinzipien gleichzeitig zu erfüllen. Zum Teil ergänzen sich die Prinzipien zwar und stellen ähnliche Anforderungen an die Gestaltung, so ist z. B. Konsistenz sowohl der Lernförderlichkeit als auch der Erwartungskonformität zuträglich. Die Prinzipien können jedoch auch konkurrierende Anforderungen nach sich ziehen. So erhöhen viele verschiedene Möglichkeiten zur Einstellung z. B. die Individualisierung, sind aber wahrscheinlich der Lernförderlichkeit abträglich. Für die Auflösung solcher Konflikte gibt es keine Patentrezepte. Es muss im Einzelfall und unter Berücksichtigung der Zielgruppe entschieden werden, welches Prinzip Priorität hat und wo Kompromisse eingegangen werden können oder müssen.

3.3 SOFTWARE-ERGONOMISCHE GESTALTUNGSRICHTLINIEN

Die in den Abschnitten 3.1 und 3.2 vorgestellten Begriffe beschreiben allgemeine Ziele der Software-Ergonomie. In Abschnitt 3.2 werden bereits einige Möglichkeiten zur Umsetzung

genannt. Es gibt aber noch weitere ganz konkrete Empfehlungen, deren Einhaltung zur Erreichung der Ziele beiträgt. Sie sind aus „physischen und psychischen Gegebenheiten“ (Herczeg, 1994, S. 71) der menschlichen Wahrnehmung ableitbar. Da das „visuelle System zentraler Informationseingabekanal“ (Herczeg, 1994, S. 71) ist, betreffen sie zu einem großen Teil die grafische Gestaltung von Dialogoberflächen. Die für die vorliegende Arbeit relevanten Empfehlungen werden im Folgenden vorgestellt. Darüber hinaus spielen bei einem interaktiven System auch die angebotenen Interaktionsformen eine große Rolle für die Gebrauchstauglichkeit und sollen näher beleuchtet werden.

Codierungsformen

Prinzipiell gibt es eine ganze Reihe von Möglichkeiten, Informationen zu codieren. So können unterschiedliche Symbole ebenso verwendet werden wie verschiedene Farben oder Variationen von Form und Größe einer Schrift. Positionen und Abstände von Elementen lassen sich gleichermaßen einsetzen wie Längen, Linienarten oder Winkel und es finden sich beliebige weitere Möglichkeiten.

Es sind jedoch nicht alle Arten der Codierung gleichermaßen zur Informationsdarstellung geeignet. Ein wesentlicher Unterschied liegt darin, wie viele Abstufungen einer Darstellungsform der Mensch sicher unterscheiden kann, speziell dann, wenn er kein Bezugssystem zur Hand hat. Hinsichtlich der Unterscheidbarkeit haben sich besonders die Verwendung von Symbolen, Positionen und Piktogrammen bewährt (Herczeg, 1994). Weitere Angaben zur Unterscheidbarkeit verschiedener Codierungsformern sind Tabelle 1 zu entnehmen. Allerdings sollte darauf geachtet werden, bei der Gestaltung einer Softwareoberfläche nie die ganze Bandbreite der menschlichen Unterscheidungsfähigkeiten auszunutzen, da nur so gewährleistet ist, dass auch unter ungünstigen Bedingungen wie schlechter Beleuchtung, Müdigkeit oder hoher kognitiver Belastung und Stress eine sichere Erkennung erfolgen kann.

Geht es bei der Gestaltung nicht in erster Linie um Unterscheidbarkeit, sondern darum, die Aufmerksamkeit des Benutzers auf eine bestimmte Information zu lenken, dann bieten sich der Einsatz von Blinken oder auffälligen Farben an (Herczeg, 1994).

Für die Gestaltung des Automatiken-GUI kommt der Einsatz von Positionen zum Einsatz, um so die verschiedenen Bereiche der Oberfläche zu unterscheiden, die jeweils eine andere Aufgabe erfüllen. Die Verwendung von Farben codiert, ob ein Element ausgewählt ist. Verschiedene Werte einzelner Automatiken-Parameter werden häufig durch Zahlenwerte codiert.

Codierungsform	Anzahl Stufen	Unterscheidbarkeit
Symbol	beliebig	ausgezeichnet
bildliche Form	10	gut
Position	9	gut
Winkel	8	gut
Farbton	6	gut
Länge	6	gut
geometrische Form	5	gut
Fläche	3	gering
Schriftgröße	3	gering
Linienart	3	gering
Schraffur	3	gering
Schriftformen	3	gering
Blinkfrequenz	2	gering
Helligkeit	2	gering

Tabelle 1: Unterscheidbarkeit verschiedener Codierungsformen (Herczeg, 1994, S. 72)

Textdarstellung

Bei der Darstellung von Text spielt vor allem die Lesbarkeit eine große Rolle. Diese hängt von verschiedenen Faktoren wie Form und Größe der Buchstaben, Linienstärke, Abständen zwischen Buchstaben, Wörtern und Zeilen u. a. m. ab. Für diese Faktoren gibt es Empfehlungen, die aus der Physiologie des menschlichen Auges und Kenntnissen über die menschliche Wahrnehmung abgeleitet werden können. Die folgenden Angaben stützen sich auf Herczeg (1994) und BGI 650 (2007). Die verschiedenen Buchstaben sollten in ihrer Gestalt hinreichend verschieden sein, um Verwechselungen zu vermeiden. Dies gilt besonders für sehr ähnliche Zeichen wie den Großbuchstaben „O“ und die Ziffer Null. Nach Herczeg (1994) sollte die Schrifthöhe eines Großbuchstaben ohne Oberlänge unter einem Sehwinkel von mindestens 18 Bogenminuten erscheinen, allerdings sollte sie auch nicht zu hoch sein. BGI 650 (2007) nennt 22 Bogenminuten als Unter- und 31 Bogenminuten als Obergrenze. Was diese Angaben für die jeweils einzustellende minimale Schriftgröße bedeuten, lässt sich abhängig vom Sehabstand mittels der in Abbildung 7 dargestellten Formel aus BGI 650 (2007) berechnen. Für die Ermittlung der maximalen Schriftgröße wird laut derselben Quelle der Divisor 155 durch 110 ersetzt. So sollten die Zeichen z. B. bei einem Sehabstand von 50 cm eine Höhe von 3,2 mm nicht unterschreiten, jedoch auch nicht größer als 4,5 mm sein. Die Zeichenbreite

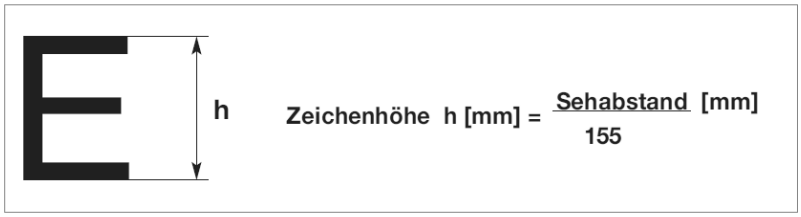


Abbildung 7: Minimale Zeichenhöhe (BGI 650, 2007)

sollte etwa 70 % und die Strichbreite 8 % bis 17 % der Zeichenhöhe betragen. Horizontal oder vertikal benachbarte Zeichen dürfen sich nicht berühren und sollten einen Abstand von mindestens 1 Pixel oder 15 % der Schrifthöhe bzw. -breite zueinander haben. Wortabstände sollten wenigstens die Breite des Zeichens „N“ haben.

Für die Gestaltung des Automatiken-GUI bedeuten diese Angaben, dass eine Schrift ausgewählt werden muss, die die genannten Kriterien erfüllt. Außerdem muss die gewählte Schriftgröße ausreichend groß und im Idealfall an die konkreten Sichtverhältnisse anpassbar sein.

Farbe

Auch die Wahl der Farben spielt für die klare Darstellung von Informationen auf dem Bildschirm und die Lesbarkeit von Text eine wesentliche Rolle. Aus diesem Grund machen die bereits im vorigen Abschnitt genannten Quellen (Herczeg, 1994; BGI 650, 2007) wie auch Balzert (2005) hierzu ebenfalls Angaben. Obwohl der Einsatz von Farben durchaus sinnvoll sein kann, z. B. um Aufmerksamkeit zu lenken oder Selektionen zu markieren, sollten nicht zu viele verschiedene Farben verwendet werden. Heide Balzert (2005, S. 246) empfiehlt, „farbig und nicht bunt“ zu gestalten, konkret empfiehlt die Literatur den Einsatz von ca. 5 bis 7 Farben. Außerdem ist auf eine hinreichende Unterscheidbarkeit der Farben zu achten, wobei gleichzeitig extreme Farbkombinationen wie der Einsatz von Komplementärfarben vermieden werden sollte. Für die Darstellung von Text ist der Einsatz von schwarzer Schrift auf weißem oder zumindest sehr hellem Grund zu favorisieren. Weitere Angaben zur Eignung bestimmter Kombinationen können Abbildung 8 entnommen werden. BGI 650 (2007) weist darauf hin, dass positive Darstellungen, also dunkle Schrift auf hellem Grund, grundsätzlich gegenüber negativen zu bevorzugen sind, wenn nicht andere Faktoren dagegen sprechen. Von der Verwendung der Farben Blau und Rot wird prinzipiell eher abgeraten, weil sie höhere Anforderungen an den Scharfstellungsmechanismus des Auges stellen. Sollen diese Farben dennoch zum Einsatz kommen, wird

Empfohlene Farbkombinationen für Zeichen und Untergrund

Untergrundfarbe	Zeichenfarbe							
	Schwarz	Weiß	Purpur	Blau	Cyan	Grün	Gelb	Rot
Schwarz		+	+	–	+	+	+	–
Weiß	++		+	+	–	–	–	+
Purpur	+	+		–	–	–	–	–
Blau	–	+	–		+	–	+	–
Cyan	+	–	–	+		–	–	–
Grün	+	–	–	+	–		–	–
Gelb	+	–	+	+	–	–		+
Rot	–	+	–	–	–	–	+	

Bedeutung:

\+ Farbkombination gut geeignet; helle Untergrundfarben (Positivdarstellung) sind vorzuziehen; nur für Bildschirme, bei denen dabei ein Flimmern auftritt, sollte eine dunkle Untergrundfarbe (Negativdarstellung) gewählt werden.

– Farbkombination nicht geeignet, da entweder Farborte zu nahe beieinander liegen, dünnlinige Zeichen nicht erkennbar sind oder zu hohe Anforderungen an den Scharfeinstellungsmechanismus der Augen gestellt werden.

Abbildung 8: Verschiedene Farbkombinationen unterstützen die Lesbarkeit unterschiedlich gut (BGI 650, 2007)

empfohlen, sie nicht für Text und dünne Linien und nicht gemeinsam zu verwenden.

Für das Automatiken-GUI bedeutet das, dass schwarze Schrift auf weißem Hintergrund zum Einsatz kommt. Ansonsten werden für die verschiedenen Bereiche und Elemente nur wenige Farben zum Einsatz kommen, die sich klar vom Untergrund abheben.

Anordnung

Bei der Anordnung von Informationen und Elementen in einem GUI kommt es immer sehr stark auf den jeweiligen Einsatzzweck an, sodass es schwierig ist, allgemeingültige Vorgaben zu formulieren. Dennoch gibt es einige empirisch gestützte Empfehlungen, die für praktisch jeden Anwendungsfall berücksichtigt werden können und bei Herczeg (1994) und Shneiderman und Plaisant (2010) beschrieben werden. Zusammengehörige Informationen sollten gruppiert werden. Auf diese Weise lassen sich Bezüge zueinander ebenso schnell erkennen wie die Abgrenzung von anderen Informationen. Für gleichartige Informationen wird die Darstellung in Tabellenform empfohlen. Weiterhin ist darauf zu achten, dass die Reihenfolge der Darbietung

von Information sinnvoll ist. Was sinnvoll bedeutet, hängt jedoch wieder vom Einzelfall ab. So kann es hilfreich sein, Informationen in der Abfolge darzustellen, die der Arbeitsaufgabe entspricht. Es kann jedoch auch besser sein, gängige Konventionen wie z. B. die übliche Reihenfolge bei Adressen einzuhalten oder sich schlicht für eine alphabetische Sortierung zu entscheiden. Egal, wie die Informationen letztlich angeordnet sind, es sollte immer darauf geachtet werden, dass die Informationsdichte nicht größer als 60 % ist, also z. B. nicht mehr als 60 % des verfügbaren Raumes von Buchstaben eingenommen werden. Im Normalfall sollte sie sogar deutlich unter diesem Wert bleiben.

Für das Automatiken-GUI spielt besonders die Gruppierung von Elementen in klar nach ihrer Funktion getrennte Bereiche eine Rolle. Innerhalb der Elementauswahlbereiche werden die Elemente praktisch als einspaltige Tabelle repräsentiert. Eine Anordnung nach ähnlicher Automatikfunktion oder Häufigkeit der Benutzung ist leider nicht ohne Weiteres möglich, da die Elemente automatisch aus dem Angebot aller verfügbaren Agenten entstehen, ohne dass die Anwendung Kenntnis der jeweiligen Funktion und Nutzungshäufigkeit hat.

Interaktionsform

Nachdem nun einiges zur Darstellung gesagt wurde, soll es in diesem Abschnitt darum gehen, wie der Benutzer mit der Anwendung interagiert. Die Gestaltung in dieser Arbeit ist vorwiegend auf die direkte Manipulation ausgelegt. Das bedeutet, dass Objekte visualisiert werden und durch direktes Ziehen oder Anklicken mit der Maus manipuliert werden können. Diese Form der Interaktion hat große Vorteile, die bei Herczeg (1994) und Dahm (2006) dargelegt werden. So ist sie sehr intuitiv und einfach zu erlernen. Nach Wessel (2002, S. 95) sind die zugrunde liegenden Techniken „so banal und einfach, dass der Benutzer sie gar nicht mehr bewusst wahrnimmt“. Die Gedächtnisbelastung ist gering, da der Benutzer sich keine Kürzel oder Kommandos merken muss, sondern seine Möglichkeiten immer direkt vor Augen hat. Er erhält sofortiges Feedback über das Ergebnis seiner Aktionen, sodass er nicht intendierte Effekte unmittelbar bemerkt. Diese sollten dann über eine entsprechende Funktion direkt zurückgenommen werden können. Auf diese Art fühlt sich der Benutzer schnell sicher und hat das Gefühl, die Anwendung kontrollieren zu können. Das steigert bei ihm Vertrauen und Zufriedenheit und beeinflusst somit die Gebrauchstauglichkeit der Software positiv. Allerdings merkt Wessel (2002, S. 95) an, dass darauf zu achten ist, die Objekte „einladend“ genug zu gestalten, damit sie vom Nutzer überhaupt als manipulierbar erkannt werden. Außerdem sollten Mausaktionen

wie Klicken oder Ziehen das übliche und erwartete Verhalten zeigen. Ergänzt wird diese Interaktionstechnik durch den Einsatz von Kontextmenüs für Operationen, die nicht unmittelbar durch Klicken oder Ziehen umsetzbar sind.

Den zahlreichen Vorteilen der direkten Manipulation stehen aber auch einige Nachteile gegenüber. So erfordert diese Form der Interaktionen einen hohen zeitlichen und motorischen Aufwand. Wenn nicht alle Aufgaben direktmanipulativ erledigt werden können, macht sie u. U. ein häufiges Wechseln zwischen Maus und Tastatur erforderlich. Außerdem ist direkte Manipulation auf die Arbeit mit einzelnen Objekten ausgelegt. Soll auf einer großen Menge von Objekten die gleiche Operation durchgeführt werden, so erweist es sich als unhandlich, jedes Objekt einzeln zu adressieren.

Daher sollte die direkte Manipulation normalerweise nicht in Reinform auftreten, sondern durch andere Techniken ergänzt werden. Eine Möglichkeit hierfür sind die u. a. von Balzert (2001) empfohlenen Tastenkürzel. Zwar erfordern diese einen erheblich größeren Lern- und Gedächtnisaufwand. Dafür ermöglichen sie aber gerade bei häufig ausgeführten Routineaufgaben eine enorme Effizienzsteigerung gegenüber dem Auffinden und Auswählen von Objekten und Operationen mit der Maus. Dahm (2006) weist darauf hin, dass beim Einsatz von Tastenkürzeln gängige Konventionen unbedingt zu beachten sind. So sollten sich weitverbreitete Kombinationen wie [STRG+A] für das Markieren aller Elemente oder [STRG+S] für Speichern stets so verhalten, wie der Nutzer es aus anderen Anwendungen gewöhnt ist und erwartet. Weitergehende Möglichkeiten der effizienten und komplexen Manipulation eröffnen sogenannte Kommandosprachen. Da diese jedoch im Automatiken-GUI keine Relevanz haben, wird an dieser Stelle auf Ausführungen hierzu verzichtet.

Für das entwickelte GUI bedeutet das, dass so gut wie alle Aktionen mit der Maus direktmanipulativ ausgeführt werden können. Elemente können durch Anklicken und Ziehen zu einer Automatik hinzugefügt und platziert werden. Durch entsprechende Mausaktionen lassen sich Elemente miteinander verbinden oder Eigenschaften einzelner Elemente anzeigen und editieren. Aktionen wie Laden, Speichern oder Verlassen des GUI können ebenfalls mit der Maus ausgeführt werden. Die Tastatur muss nur selten verwendet werden, z. B. zum Eingeben von Dateinamen beim Speichern.

3.4 USABILITY-EVALUATION

Da „die Evaluation neben der Anforderungsermittlung und der Erstellung von Bedienkonzepten das zentrale Element des Usability-Engineerings“ (Sarodnick und Brau, 2006, S. 113) darstellt,

werden im Folgenden Ansätze vorgestellt, mit deren Hilfe sich Softwareanwendungen auf ihre Gebrauchstauglichkeit hin untersuchen lassen. Bei den Methoden wird zwischen analytischem und empirischem Vorgehen unterschieden (Sarodnick und Brau, 2006). Beim analytischen Ansatz sind nur Usability-Experten beteiligt, während bei den empirischen Methoden Vertreter der späteren Nutzergruppe beteiligt sind. Beide Wege dienen dazu, Usability-Probleme aufzudecken und ggf. Alternativen zu generieren. Der Fokus der einzelnen Methoden kann jedoch variieren.

3.4.1 *Analytische Methoden*

Viele analytische Methoden der Usability-Evaluation können schon sehr früh im Designprozess angewendet werden, da bereits Papierprototypen, Beschreibungen des Systems oder auch nur das Lastenheft zur Durchführung ausreichend sind (Sarodnick und Brau, 2006). Daher eignen sie sich gut dafür, Entwürfe von Anfang an auf mögliche Probleme hin zu untersuchen. Dies kann eine enorme Zeit- und Kostenersparnis für ein Projekt bedeuten (Shneiderman und Plaisant, 2010).

Ein Problem, das die analytischen Methoden haben, ist, dass sie nur dann gute Ergebnisse liefern können, wenn es dem Evaluator gelingt, sich in die Rolle des Nutzers hineinzuversetzen. Außerdem fehlen den Usability-Experten in der Regel fundierte Kenntnisse der Anwendungsdomäne. Daher können sie Probleme, die auf fachlichen Aspekten oder typischen Arbeitsabläufen beruhen, oft nicht gut vorhersehen (Shneiderman und Plaisant, 2010; Balzert, 2001).

Die einfachsten Ansätze beruhen darauf, einen Entwurf mit sogenannten Styleguides abzugleichen oder auf Konsistenz zu überprüfen (Shneiderman und Plaisant, 2010).

Als wichtige Vertreter komplexerer Methoden seien an dieser Stelle die Heuristische Evaluation und der Cognitive Walkthrough[12] genannt. Für die Bewertung benötigen beide einen Prototypen der Anwendung. Bei der Heuristischen Evaluation wird dieser anhand einer Liste von Kriterien auf „erwünschte Eigenschaften der Interaktion“ (Sarodnick und Brau, 2006, S. 135) hin untersucht. In einem Cognitive Walkthrough versuchen die Evaluatoren, Probleme bei der Erlernbarkeit durch explorative Nutzung aufzudecken (Sarodnick und Brau, 2006; Shneiderman und Plaisant, 2010).

12 An dieser Stelle wird bewusst der englische Begriff beibehalten, da er auch in der deutschen Literatur durchgängig verwendet wird.

3.4.2 *Empirische Methoden*

Bei den empirischen Methoden werden Vertreter der späteren Nutzergruppe mit einem Prototyp des Systems in Berührung gebracht. Typischerweise sollen sie vorgegebene Aufgaben lösen und werden dabei beobachtet und anschließend eventuell noch befragt. Die Fragestellung der Evaluation kann verschieden sein. So kann z. B. untersucht werden, wie schnell Nutzer eine Aufgabe lösen, wie zufrieden sie mit der Benutzung der Software sind oder wie viele und welche Fehler sie machen.

Meist wird die empirische Evaluation eines Systems mit realen oder realistischen Nutzern im Usability-Labor oder anhand von Fragebögen durchgeführt. Im Usability-Labor arbeiten die Teilnehmer mit dem fraglichen System. Die Interaktion wird beobachtet oder kann durch technische Hilfsmittel zur späteren Auswertung aufgezeichnet werden. Bei der Fragebogenmethode beantworten die Probanden eine Reihe von Fragen zur Nutzung einer Software, meist in Form von Multiple-Choice-Aufgaben oder durch Angaben auf Ratingskalen.

3.5 VORGEHENSMODELL

Nachdem sich dieses Kapitel bislang ausschließlich mit softwareergonomischen Fragestellungen beschäftigt hat, soll nun betrachtet werden, inwieweit die gängigen Entwicklungsprozesse der Softwaretechnik mit der Entwicklung nutzerfreundlicher Software in Einklang zu bringen sind.

Das älteste Modell, das jedoch auch heute noch häufig angeführt wird, ist das Wasserfall-Modell (Abbildung 9). Dieses geht davon aus, dass sich der Entwicklungsprozess in mehrere Phasen gliedern lässt, nämlich Anforderungsdefinition, System- und Softwareentwurf, Implementierung und Komponententest, Integration und Systemtests sowie Betrieb und Wartung (Sommerville, 2007). Diese Phasen werden nacheinander abgearbeitet. Bei Bedarf kann eine Phase wiederholt werden, nach einigen Iterationen werden die Ergebnisse jedoch eingefroren. Überlappung findet nicht statt; bevor der Prozess in eine neue Phase eintritt, müssen die vorherigen abgeschlossen sein.

Ein Vorteil dieses Modells liegt nach Sommerville (2007) darin, dass jede Phase Dokumentation erzeugt. Zudem bietet dieser Prozess einen guten Überblick über den Projektfortschritt. Ein bedeutender Nachteil liegt allerdings darin, dass dieses Modell keine Korrektur von Problemen vorsieht, die erst in späteren Projektphasen auffallen. So weist Sommerville (2007) auch darauf hin, dass ein solches überlappungsfreies Modell nicht den in der Praxis üblichen Abläufen entspricht.

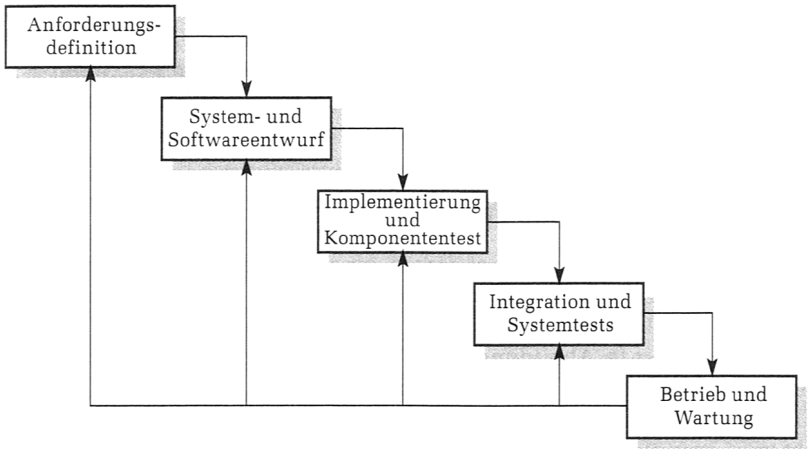


Abbildung 9: Wasserfall-Modell (Sommerville, 2007, S. 97)

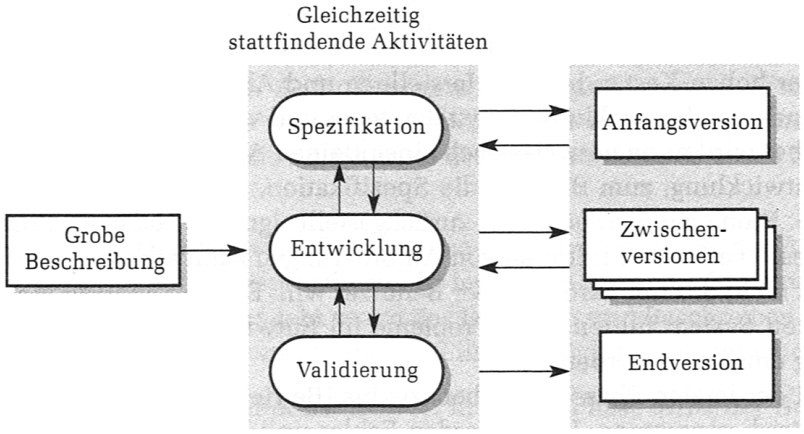


Abbildung 10: Evolutionäre Entwicklung (Sommerville, 2007, S. 98)

Ein anderes Modell, welches den praktischen Schwächen des Wasserfall-Modells Rechnung trägt, ist der Ansatz der Evolutionären Entwicklung (Abbildung 10). Hier wird davon ausgegangen, dass die Phasen Spezifikation, Entwicklung und Validierung in mehreren Iterationen parallel stattfinden und untereinander in Wechselwirkung stehen, bis das gewünschte Entwicklungsziel erreicht ist (Sommerville, 2007). Dieses Modell erlaubt es, flexibel auf Anforderungsänderungen zu reagieren, die u. a. aus einem im Laufe der Zeit immer tieferen Problemverständnis bei Kunden und Entwicklern resultieren. Dadurch ist dieses Vorgehen besonders dann sehr effektiv, wenn es darum geht, Kundenbedürfnisse zu befriedigen. Nachteile dieses Ansatzes sind, dass der Fortschritt des Entwicklungsprozesses kaum messbar ist und häufig schlecht strukturierte Systeme entstehen. Dennoch empfiehlt Sommerville (2007) dieses Vorgehen gerade für die Erstellung von Benutzeroberflächen.

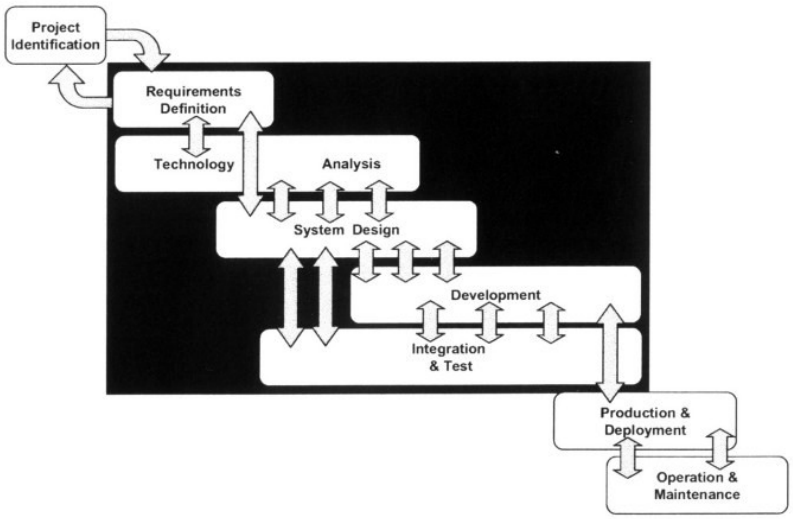


Abbildung 11: Concurrent Engineering Model (Endsley et al., 2003, S. 44)

Als Kompromiss stellen Endsley, Bolté und Jones (2003) das Concurrent Engineering Model (CEM) vor (Abbildung 11). Dieser Ansatz vereint die beiden oben vorgestellten Modelle, indem er zwar ein dem Wasserfall-Modell sehr ähnliches Vorgehen definiert, dabei jedoch eine breite Überlappung der einzelnen Phasen ausdrücklich zulässt. Dadurch kann die notwendige Wechselwirkung zwischen den Phasen stattfinden. Zudem wird die Entwicklung in interdisziplinären Teams unterstützt, da alle Teilnehmer parallel arbeiten und sich rechtzeitig austauschen können, bevor bestimmte Aspekte unverrückbar festgeschrieben sind. Die einzigen Tätigkeiten, die nach Endsley et al. (2003) nicht zu stark überlappen sollten, sind Test und produktiver Einsatz. Zwar würde dies eine schnelle Markteinführung begünstigen, jedoch wäre Frustration beim Benutzer, der mit dem noch fehlerhaften System umgehen muss, vorprogrammiert.

Während die oben vorgestellten Modelle sich allgemein dem Entwurf von Software widmen, geht Balzert (2001) konkret auf den Entwurf von Benutzeroberflächen ein. Hierfür schlägt er folgendes fünfschrittige Vorgehen vor:

1. Fenster und ihre gegenseitige Interaktion skizzieren
2. Bestandteile jedes Fensters festlegen
3. Benutzerereignisse festlegen, auf die reagiert werden soll
4. Entwurf des in den vorigen Schritten erstellten Konzeptes in der Ziel-Programmiersprache
5. Programmierung

Die vorliegende Arbeit ist in ein interdisziplinäres Projekt eingebettet. Der Entwurf einer Benutzeroberfläche bildet den zentralen Bestandteil und es war bereits frühzeitig absehbar, dass auf geänderte Anforderungen und Voraussetzungen flexibel reagiert werden muss. Daher wurde aufgrund der vorangegangenen Betrachtungen das CEM in Kombination mit dem Entwurf von Benutzeroberflächen nach Balzert (2001) als das am besten geeignete Vorgehensmodell ausgewählt und angewendet. Für die Entwicklungsarbeit bedeutete das konkret eine breite Überlappung zwischen den Phasen Entwurf und Implementierung. Durch wiederholte Diskussion innerhalb der Projektgruppe und Bewertung verschiedener Prototypen der Oberfläche unter technischen und software-ergomischen Gesichtspunkten wurden fortlaufend neue Erkenntnisse gewonnen. Diese wurden jeweils in einen verbesserten Entwurf umgesetzt. Der Prozess wurde wiederholt, bis das Ergebnis zufriedenstellend ausfiel. In diesem Vorgehen wird die evolutionäre Komponente des CEM sehr deutlich. Sie beschränkt sich jedoch hauptsächlich auf die beiden genannten Phasen. Die übrigen sind weitgehend, wenn auch nicht streng, sequenziell erfolgt, sodass hier keine Anwendung der Evolutionären Entwicklung vorliegt.

4 PRAKTISCHE UMSETZUNG

4.1 ANFORDERUNGSANALYSE

„Anforderungen legen die qualitativen und quantitativen Eigenschaften eines Produkts aus der Sicht des Auftraggebers fest“ (Balzert, 2001, S. 98) und bilden somit den Ausgangspunkt eines jeden Software-Projekts. Auf der Ermittlung der Anforderungen bauen alle weiteren Schritte des Entwicklungsprozesses auf. Daher ist es wichtig, gerade diesen Schritt mit äußerster Sorgfalt durchzuführen und sicherzustellen, dass genau das Produkt definiert wird, das der Auftraggeber erwartet. Fehler in der Ermittlung führen schnell zu hohen Zusatzkosten im Projekt (Grechenig, 2010), weil die Änderung von Anforderungen in späteren Phasen der Entwicklung sehr aufwendig durchzuführen ist und Konsequenzen für alle nachfolgenden, bereits durchgeführten Phasen nach sich zieht.

Für die Dokumentation der Anforderungen gibt es verschiedene Ansätze. So empfiehlt Balzert (2001) die Aufteilung in Muss-, Wunsch- und Abgrenzungskriterien. Statt von Abgrenzungskriterien spricht Grechenig (2010) auch von Nicht-Zielen, die explizit festschreiben, welche Funktionen das System nicht bieten soll. Auf diese Weise soll verhindert werden, dass ein Projekt durch immer neue Ideen für weitere Funktionen ausufert, bis es nicht mehr handhabbar ist. Eine andere Aufteilung der Anforderungen, die Grechenig (2010) vorschlägt, ist die in funktionale und nicht-funktionale Anforderungen. Dabei sind sämtliche Anforderungen, die das Kernverhalten des Systems beschreiben, funktional. Zu den nicht-funktionalen Anforderungen hingegen gehören Rahmenbedingungen wie z. B. eine vorgegebene Programmiersprache oder Vorgaben aus Normen, Standards und Gesetzen.

Unabhängig davon, welche Aufgliederung für die Dokumentation gewählt wird, müssen die Anforderungen vollständig, konsistent und eindeutig sein. Zudem muss überprüfbar sein, ob sie erfüllt sind. Dies ist auch deswegen wichtig, weil die Anforderungen, z. B. in Form eines Pflichtenhefts, häufig Vertragsbestandteil sind und zugleich die Kriterien für den Endabnahmetest bilden (Balzert, 2001; Grechenig, 2010).

Um Anforderungen zu erheben, bieten sich mehrere Quellen an. Als die vielleicht wichtigsten sind der Auftraggeber und die späteren Benutzer zu nennen, wobei gerade Letztere für die benutzerorientierte Entwicklung eine besondere Rolle einnehmen.

Anforderungen können aber auch, um nur einige weitere Möglichkeiten zu nennen, aus Dokumentationen von Arbeitsprozessen gewonnen werden, aus der Analyse von Altsystemen stammen oder Gesetzen, Normen und Standards entnommen werden.

Die Anforderungen, die dem Automatiken-GUI in dieser Arbeit zugrunde liegen, stammen vom Auftraggeber. Sie wurden zu großen Teilen bereits im Rahmen einer früheren Diplomarbeit (Kesselring, 2009) erhoben, in der die genannte Software-Komponente jedoch noch nicht umgesetzt wurde.

Eine häufig verwendete Methode, um Anforderungen für eine Software zu ermitteln und zu verfeinern, bildet das Sammeln und Analysieren von Anwendungsfällen[13] [14] (engl.: Use Cases). Die Anforderungen sind vollständig, wenn sich mit ihnen alle gefundenen Anwendungsfälle realisieren lassen. Dabei meint Anwendungsfall eine „Sequenz von Aktionen, einschließlich möglicher Varianten, die das System in Interaktion mit Akteuren ausführt“ (Balzert, 2005, S. 62). Ein Akteur ist „eine Rolle, die ein Benutzer des Systems spielt“ (Balzert, 2005, S. 63). Akteure interagieren mit dem System, um bestimmte Ziele zu erreichen, befinden sich selbst jedoch außerhalb des Systems. Akteure sind häufig Personen, es können aber auch Dinge wie z. B. andere Systeme sein, die mit dem System in Kontakt treten. Jeder Anwendungsfall dient der Erreichung von genau einem klar umrissenen Ziel. Gibt es mehrere Ziele, die mit dem System erreicht werden sollen, so ist für jedes ein separater Anwendungsfall zu erstellen. Die Gesamtheit aller Anwendungsfälle spiegelt den kompletten Funktionsumfang des Systems wider. Eine gute Übersicht über das Zusammenwirken verschiedener Anwendungsfälle bieten Use-Case-Diagramme in der Unified Modeling Language (UML).

4.2 ANFORDERUNGEN

Im Folgenden werden die Anforderungen beschrieben, die innerhalb des Projekts für das Automatiken-GUI identifiziert wurden. Dabei werden sie in Muss-, Wunsch- und Abgrenzungskriterien

13 In der Literatur wird zwischen Anwendungsfällen auf Systemebene (System Use Cases) und Anwendungsfällen auf Unternehmensebene (Business Use Cases) unterschieden (Grechenig, 2010; Balzert, 2005). Da es in dieser Arbeit jedoch nur um Anwendungsfälle auf Systemebene geht, wird der Begriff Anwendungsfall ohne weitere Zusätze in diesem Sinne verwendet.

14 Bei Helmut Balzert (2001) findet sich hierfür der Begriff *Geschäftsprozess im Kleinen* oder auch nur *Geschäftsprozess*. Da dieser Begriff jedoch nahe legen könnte, es ginge um Geschäftsanwendungsfälle (Business Use Cases) oder es sei eine Geschäftsprozessmodellierung (Grechenig, 2010), ein vollständig anderes Verfahren zur Anforderungsgewinnung, durchgeführt worden, wird der Begriff *Geschäftsprozess* hier bewusst vermieden.

aufgeschlüsselt. In Tabellen 2 und 3 ist zu sehen, welche der Kriterien funktional bzw. nicht-funktional sind.

4.2.1 *Musskriterien*

ERSTELLEN VON AUTOMATIKEN AUS MODULEN Der Anwender soll Elementar-Automatiken, sogenannte Agenten, als Bausteine zur Verfügung gestellt bekommen. Jede Elementar-Automatik hat eine klar umrissene Basisfunktionalität (z. B. Bremsen, Lenken, etc.), die u. U. durch Parameter in einem gewissen Rahmen variiert werden kann. Aus diesen Bausteinen kann der Anwender komplexere Automatiken aufbauen, indem er sie miteinander kombiniert.

SPEICHERN VON AUTOMATIKEN Es soll möglich sein, erstellte Automatiken abzuspeichern. Dabei sollen sowohl Automatiken speicherbar sein, die gültig sind[15], als auch Automatiken, die sich noch in einem Entwurfsstadium befinden. Der Dateibasisname ist frei wählbar, die Endung ist vorgegeben und für alle Automatiken gleich.

LADEN VON AUTOMATIKEN Abgespeicherte Automatiken sollen im Automatiken-GUI wieder geladen werden können.

VERÄNDERN VON AUTOMATIKEN Abgespeicherte Automatiken sollen im Automatiken-GUI verändert werden oder als Grundlage für neue Automatiken dienen können.

EINSTELLEN DER PARAMETER EINES AGENTEN Für jeden Agenten, der Parameter besitzt, sollen diese im Automatiken-GUI sichtbar und einstellbar sein. Soweit möglich, soll das GUI die Eingabe von Parametern überwachen und Fehleingaben wie unzulässige Werte oder Formate verhindern.

EINSTELLEN VISUELLER HINWEISE Diese Anforderung bildet eine spezialisierte Unteranforderung zur zuvor genannten Anforderung. Visuelle Hinweise müssen im Automatiken-GUI eingestellt werden können. Hierzu gehören die Auswahl des Hinweises und des Streckenabschnitts, in dem dieser angezeigt werden soll. Die Position des Hinweises im Streckenabschnitt muss einstellbar sein. Weiterhin müssen Schrifttyp und -größe, Schriftfarbe und die angezeigte Größe des Hinweises veränderbar sein. Es muss eine Vorschau geben, in der zu sehen ist, wie der Hinweis

15 *Gültig* meint hier, dass sie in einem Versuch eingesetzt werden können, also Ausgaben liefern, die während eines Versuchs in das SAM-System eingespeist werden können.

später während einer Untersuchung erscheinen wird. In dieser Vorschau soll als Alternative zur direkten Eingabe von Zahlenwerten die Positionierung des Hinweises durch Verschieben mit der Maus möglich sein.

AKTIVIEREN EINER AUTOMATIK FÜR EINEN VERSUCH Eine abgespeicherte gültige Automatik muss für einen Versuch als die zu verwendende gesetzt werden können.

AKTIVIERUNG UND DEAKTIVIERUNG EINZELNER AGENTEN FÜR BESTIMMTE STRECKENABSCHNITTSTYPEN Es soll möglich sein, jeden eingesetzten Agenten separat für bestimmte Streckenabschnittstypen[16] zu aktivieren oder zu deaktivieren. Da alle Agenten diese Konfigurationsmöglichkeit besitzen, soll es möglich sein, diese Eigenschaft für mehrere Agenten gleichzeitig zu bearbeiten, also z. B. drei Agenten zugleich für den Einsatz in Kurven zu deaktivieren.

SQUEAK/SMALLTALK Da im Projekt ATEO bislang alle Komponenten in Squeak/Smalltalk realisiert wurden, soll auch das Automatiken-GUI in dieser Programmierumgebung und -sprache implementiert werden.

4.2.2 *Wunschkriterien*

VERWENDEN EINER BESTEHENDEN AUTOMATIK ALS MODUL Eine abgespeicherte gültige Automatik soll in gleicher Weise wie die Agenten als Baustein zur Verfügung gestellt werden, den der Anwender in neue Automatiken einbauen kann. Der einzige Unterschied im Verhalten gegenüber den Agenten soll sein, dass der Anwender in ein solches Modul hinein verzweigen und Änderungen an der enthaltenen Automatik vornehmen kann. Solche Änderungen sollen keine Auswirkungen auf die zugrunde liegende abgespeicherte Automatik oder auf andere Instanzen des Moduls haben, sondern auf das bearbeitete Element beschränkt sein.

GESTALTUNG NACH SOFTWARE-ERGONOMISCHEN GESICHTSPUNKTEN Soweit die zu verwendende Entwicklungsumgebung und -sprache Squeak/Smalltalk es zulässt, soll die Gestaltung des Automatiken-GUI nach software-ergonomischen Gesichtspunkten erfolgen.

16 Kurven, Gabelungen, gerade Strecken usw.

FUNKTIONALE ANFORDERUNGEN
Erstellen von Automatiken aus Modulen
Speichern von Automatiken
Laden von Automatiken
Verändern von Automatiken
Einstellen der Parameter eines Agenten
Einstellen visueller Hinweise
Aktivieren einer Automatik für einen Versuch
Aktivierung und Deaktivierung einzelner Agenten für bestimmte Streckenabschnittstypen
Verwenden einer bestehenden Automatik als Modul

Tabelle 2: Funktionale Anforderungen

4.2.3 *Abgrenzungskriterien*

KEINE BEARBEITUNGSMÖGLICHKEITEN DER AUTOMATIKEN WÄHREND EINER LAUFENDEN UNTERSUCHUNG. Die benötigten Automatiken sind vor einer Untersuchung zu erstellen und zu konfigurieren. Es ist nicht vorgesehen, die verwendeten Automatiken wechseln oder Parameter verändern zu können, während eine Untersuchung läuft.

NICHT-FUNKTIONALE ANFORDERUNGEN
Squeak/Smalltalk
Gestaltung nach software-ergonomischen Gesichtspunkten

Tabelle 3: Nicht-funktionale Anforderungen

4.3 ANWENDUNGSFÄLLE

Im Folgenden werden zunächst die für das Automatiken-GUI identifizierten Anwendungsfälle dargestellt. Zur detaillierten Beschreibung der einzelnen Fälle kommt die bei Helmut Balzert (2001) und Heide Balzert (2005) vorgeschlagene Schablone zum Einsatz. Für jeden Anwendungsfall wird ein kurzer Name vergeben. Es folgt eine umgangssprachliche Formulierung des Ziels, welches der jeweilige Anwendungsfall verfolgt. Die Kategorie kann einen der Werte primär, sekundär oder optional annehmen. Dabei bedeuten primär und sekundär, dass die Anwendungsfälle notwendiges Verhalten beschreiben. Primäre Anwendungsfälle beschreiben häufig und sekundäre eher selten verfolgte Ziele.

Optional wird vergeben, wenn das Verhalten zwar nützlich, aber nicht unbedingt notwendig für das System ist.

4.3.1 *Beschreibung der Anwendungsfälle*

/ F10 /
Anwendungsfall: Erstellen einer neuen Automatik
Ziel: Eine neue Automatik steht für Versuche zur Verfügung
Kategorie: primär
Vorbedingung: keine
Nachbedingung Erfolg: Eine neue Automatik steht für Versuche zur Verfügung
Nachbedingung Fehlschlag: Es steht keine neue Automatik zur Verfügung
Akteure: Versuchsleiter
Auslösendes Ereignis: Das Automatiken-GUI wird gestartet oder in den Anfangszustand zurückgesetzt
Beschreibung:
1. Elemente zur Automatik hinzufügen
2. Verbindungen zwischen Elementen hinzufügen
3. Parameter der Automatik einstellen
4. Automatik abspeichern

/ F20 /
Anwendungsfall: Abspeichern einer Automatik
Ziel: Die Automatik ist zur Wiederverwendung gespeichert
Kategorie: primär
Vorbedingung: Es liegt eine Automatik vor
Nachbedingung Erfolg: Die Automatik ist gespeichert
Nachbedingung Fehlschlag: Die Automatik ist nicht gespeichert, es wird eine Fehlermeldung angezeigt
Akteure: Versuchsleiter
Auslösendes Ereignis: Eine Automatik soll zur späteren Wiederverwendung gespeichert werden
Beschreibung:
1. Speicherort auswählen
2. Name für die Automatik vergeben
3. Speichern bestätigen

/ F30 /
Anwendungsfall: Laden einer Automatik
Ziel: Eine Automatik ist geladen und steht zur Bearbeitung zur Verfügung
Kategorie: primär
Vorbedingung: Es liegt eine Automatik vor
Nachbedingung Erfolg: Die Automatik ist geladen und steht zur Bearbeitung zur Verfügung

Nachbedingung Fehlschlag: Die Automatik ist nicht geladen, es wird eine Fehlermeldung angezeigt
Akteure: Versuchsleiter
Auslösendes Ereignis: Das Automatiken-GUI wird gestartet mit der Absicht, eine bestehende Automatik zu laden
Beschreibung:
1. Speicherort auswählen
2. Automatik auswählen
3. Laden bestätigen

/ F40 /
Anwendungsfall: Erstellen einer neuen Automatik auf Basis einer bereits vorliegenden
Ziel: Eine neue Automatik steht für Versuche zur Verfügung
Kategorie: primär
Vorbedingung: Es existieren gespeicherte Automatiken
Nachbedingung Erfolg: Eine neue Automatik steht für Versuche zur Verfügung
Nachbedingung Fehlschlag: Es steht keine neue Automatik zur Verfügung
Akteure: Versuchsleiter
Auslösendes Ereignis: Das Automatiken-GUI wird gestartet mit der Absicht, eine neue Automatik auf Basis einer existierenden zu erstellen
Beschreibung:
1. Automatik laden
2. weitere Elemente zur Automatik hinzufügen
3. Parameter überprüfen und ggf. ändern
4. Automatik unter neuem Namen abspeichern
Alternativen:
2a. Elemente aus der Automatik entfernen
2b. Verbindungen von Elementen hinzufügen
2c. Verbindungen von Elementen entfernen

/ F50 /
Anwendungsfall: Verändern einer Automatik
Ziel: Eine veränderte Automatik steht für Versuche zur Verfügung
Kategorie: primär
Vorbedingung: Es existieren gespeicherte Automatiken
Nachbedingung Erfolg: Eine veränderte Automatik steht für Versuche zur Verfügung
Nachbedingung Fehlschlag: Es steht keine veränderte Automatik zur Verfügung
Akteure: Versuchsleiter
Auslösendes Ereignis: Das Automatiken-GUI wird gestartet mit der Absicht, eine bestehende Automatik zu verändern

Beschreibung:
1. Automatik laden
2. weitere Elemente zur Automatik hinzufügen
3. Parameter überprüfen und ggf. ändern
4. Automatik unter dem alten Namen abspeichern
Alternativen:
2a. Elemente aus der Automatik entfernen
2b. Verbindungen von Elementen hinzufügen
2c. Verbindungen von Elementen entfernen

/ F60 /
Anwendungsfall: Einstellen eines visuellen Hinweises
Ziel: Für eine Automatik ist ein visueller Hinweis eingestellt
Kategorie: primär
Vorbedingung: Im Automatiken-GUI liegt eine Automatik vor, die einen Agenten für visuelle Hinweise enthält
Nachbedingung Erfolg: Der Agent für visuelle Hinweise ist eingestellt
Nachbedingung Fehlschlag: Der Agent für visuelle Hinweise ist nicht eingestellt
Akteure: Versuchsleiter
Auslösendes Ereignis: Eine Automatik soll einen visuellen Hinweis in einer bestimmten Weise anzeigen
Beschreibung:
1. Element auswählen, welches den Agenten für visuelle Hinweise enthält
2. Hinweisbild auswählen
3. Streckenabschnitt auswählen
4. Position des Hinweises mit Zahlenwerten einstellen
5. Aussehen der Schrift einstellen
6. Größe des Hinweises einstellen
Alternativen:
2a. Hinweis als Freitext eingeben
4a. Die Vorschau öffnen und die Position des Hinweises mit der Maus einstellen

/ F70 /
Anwendungsfall: Aktivieren einer Automatik für eine Untersuchung
Ziel: Eine Automatik ist für einen Versuch aktiviert
Kategorie: primär
Vorbedingung: keine
Nachbedingung Erfolg: Eine Automatik ist für einen Versuch aktiviert
Nachbedingung Fehlschlag: Für den Versuch ist keine Automatik aktiviert
Akteure: Versuchsleiter

Auslösendes Ereignis: Eine Automatik soll in einer Untersuchung eingesetzt werden
Beschreibung:
1. Automatik im Automatiken-GUI erstellen
2. Automatik aktivieren
Alternativen:
1a. Eine Automatik laden

4.3.2 *Grafische Repräsentation der Anwendungsfälle*

In Abbildung 12 sind die Anwendungsfälle in Form eines Use-Case-Diagramms in UML grafisch repräsentiert. Dieses zeigt auf einem hohen Abstraktionsniveau die Interaktion zwischen Akteur und Anwendungsfällen sowie das Zusammenspiel der Anwendungsfälle untereinander. Die Ovale stellen die Anwendungsfälle dar, Akteure werden durch Strichmännchen repräsentiert. Die include-Beziehung beschreibt, dass ein Anwendungsfall einen anderen verwendet, um sein Ziel zu erreichen.

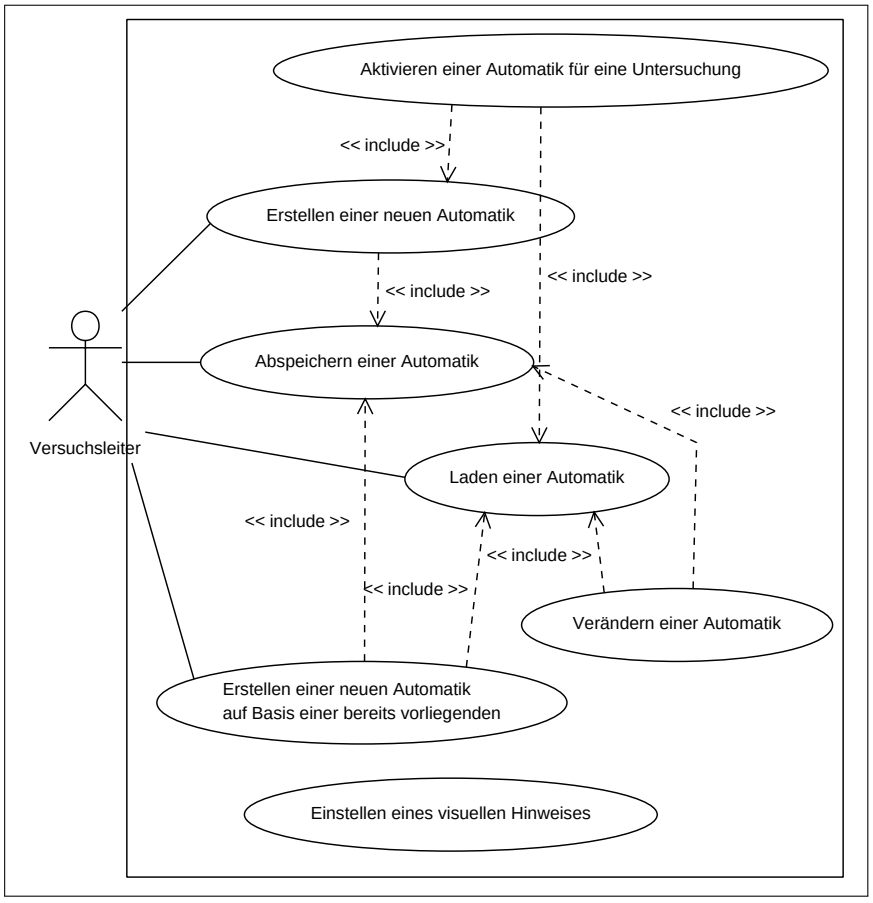


Abbildung 12: Use-Case-Diagramm des Automatiken-GUI

4.4 OOA- UND OOD-MODELL

Aus den oben vorgestellten Anforderungen und Anwendungsfällen wurde zunächst ein OOA-Modell (objektorientierte Analyse) erstellt, welches die identifizierten Klassen, ihre Assoziationen sowie die Vererbungsstrukturen beinhaltet. Im Laufe des Entwurfsprozesses wurde dieses zum OOD-Modell (objektorientiertes Design) verfeinert, welches die anschließende Implementierung widerspiegelt (Balzert, 2005). Sommerville (2007) spricht hierbei auch von objektorientierter Dekomposition. Die resultierenden UML-Klassendiagramme sind in Anhang B zu sehen.

Beim Entwurf der Klassen wurde vor allem Wert auf einfache Wartbarkeit und Erweiterbarkeit gelegt. Das ist vor allem deswegen wichtig, weil das Projekt ATEO nach Abschluss dieser Arbeit weitergeführt wird, sodass zukünftige Änderungen am Automatiken-GUI wahrscheinlich sind. Um die Wiederverwendbarkeit der entstandenen Klassen zu gewährleisten, wurden sie so entworfen, dass sie jeweils nur eine Zuständigkeit haben und alle durch sie realisierten Funktionalitäten eng verwandt sind, sie also eine hohe Kohäsion besitzen (McLaughlin, Pollice, West und Schulten, 2008).

Weiterhin wurde darauf geachtet, dass das Automatiken-GUI leicht um weitere Automatiken zu erweitern ist, die im Laufe der Zeit von Entwicklern entworfen werden und integriert werden müssen. Hierfür wurde eine klare Schnittstelle entworfen. Das Vorgehen zur Integration wird in Anhang F erläutert.

4.5 AAF

Die Schnittstelle zwischen dem Automatiken-GUI und SAM bildet das ATEO Automation Framework (AAF), welches in einer parallelen Diplomarbeit von Michael Hasselmann (in Bearb.) entwickelt wird. Zum einen definiert das AAF Agenten nebst Methoden, die diese implementieren müssen, um als Bestandteil einer komplexeren Automatik zum Einsatz kommen zu können. Zum anderen stellt das AAF eine Datenstruktur zur Repräsentation von Graphen zur Verfügung, in welcher einzelne Agenten zu komplexen Automatiken verbunden werden können. Eine weitere wichtige, jedoch für das Automatiken-GUI irrelevante, Funktion des AAF ist die Interaktion mit SAM, um während einer Untersuchung die Überwachung und Steuerung des Trackings mittels Automatiken durchzuführen.

Es ist durchaus möglich, eine Automatik direkt auf Programmcode-Ebene in Squeak/Smalltalk zu erstellen und zu konfigurieren, ohne dafür das Automatiken-GUI zu verwenden. Dieser Weg setzt jedoch entsprechende Kenntnisse des AAF und der

Smalltalk-Programmierung voraus (Abbildung 3). Durch Verwendung des Automatiken-GUI ist ein solches Wissen für den Anwender nicht notwendig. Zudem ist dieser Weg der Automatiken-Erstellung weniger fehleranfällig, da bereits während der Erstellung die Einhaltung der für AAF-Graphen definierten Regeln überwacht wird.

```
1 g := AAFGraph new.
2 n := AAFNode new.
3 n delegate: AAFBreaksAgent new.
4 g connectParent: g root withChild: n.
5 g connectParent: n withChild: g sink.
```

Listing 3: Diese Codezeilen definieren einen Graphen mit nur einem Agenten. Es werden die Standardeigenschaften des Agenten verwendet, Anpassungen finden nicht statt.

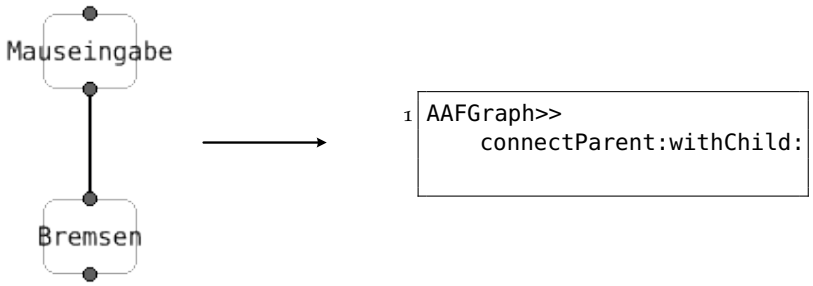


Abbildung 13: Das Herstellen einer Verbindung im Automatiken-GUI bewirkt den Aufruf der entsprechenden Methode im AAF.

Das GUI bildet die Aktionen des Nutzers unmittelbar auf die vom AAF bereitgestellten Strukturen ab, indem es die entsprechenden Klassen und Methoden verwendet (Abbildung 13). Diese direkte Verwendung ist dadurch leicht möglich, dass die Graphenstruktur des AAF in enger Abstimmung mit dem Projekt dieser Diplomarbeit entworfen wurde und somit die Erfordernisse des GUI berücksichtigt. Zwar muss das GUI den Graphen noch eine grafische Repräsentation hinzufügen, es ist jedoch keine Transformation der logischen Strukturen notwendig, um die erstellten Automatiken in einer Untersuchung verwenden zu können.

In Abbildung 14 ist schematisch dargestellt, wie das AAF zwischen dem Automatiken-GUI und dem SAM-System vermittelt. Im Folgenden sollen nun die Agenten sowie die Graphenstruktur des AAF näher erläutert werden. Ein Klassendiagramm der hier relevanten Komponenten des AAF findet sich in Anhang A.

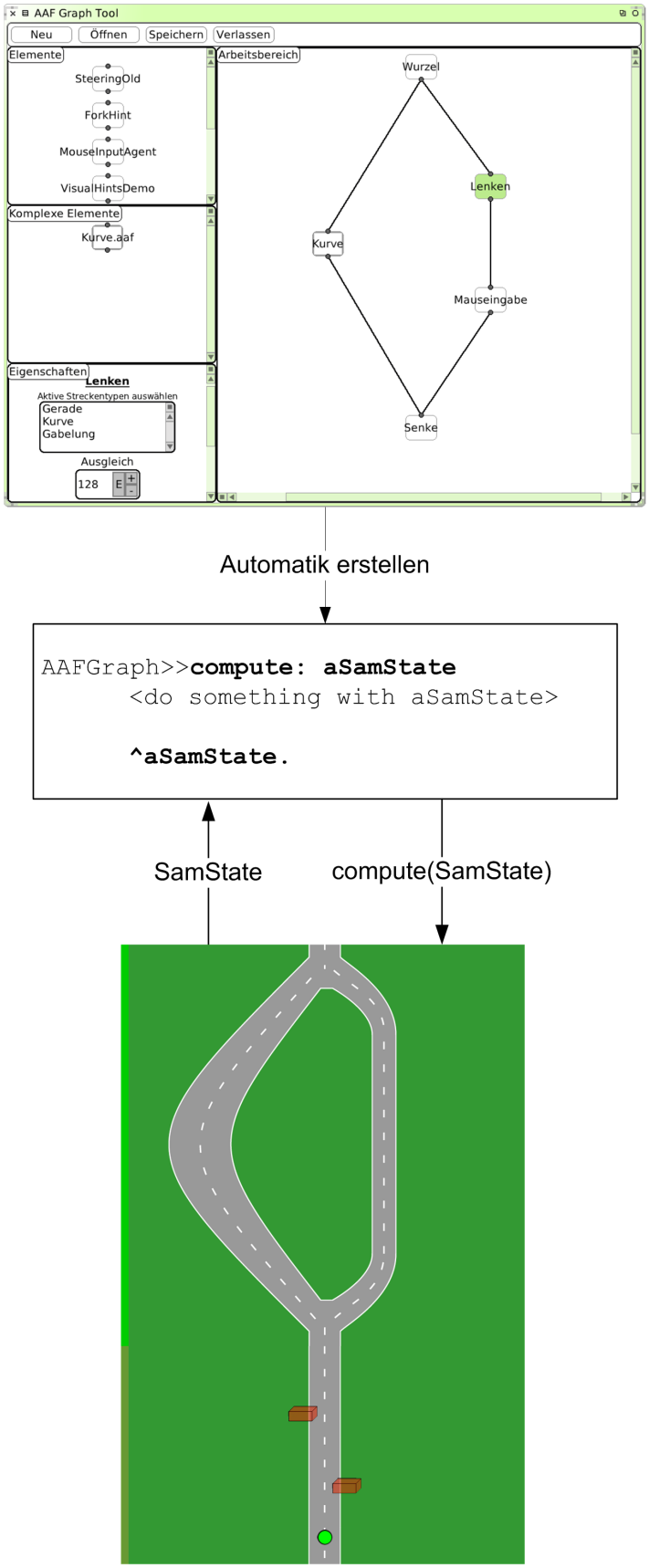


Abbildung 14: Das AAF bildet die Vermittlungsschicht zwischen dem Automatiken-GUI und SAM. Mit dem GUI werden AAF-Automatiken erstellt. Während einer Untersuchung verwendet AAF diese, um die Steuerung zu beeinflussen.

4.5.1 *Agenten*

Die Agenten bilden die elementaren Automatiken im AAF. Jeder Agent soll nur eine einzige, klar umrissene Automatikfunktion ausführen wie z. B. Bremsen oder Spurhalten.

Agentenklassen müssen von der Basisklasse `AAFAgent` abgeleitet werden. Von dieser erben sie eine Liste, in der gespeichert wird, für welche Streckenabschnittstypen wie Kurve, Gabelung usw. sie aktiv sein sollen. Darüber hinaus kann jeder Agent weitere Eigenschaften definieren, die er für seine Aufgabe benötigt.

Jeder Agent muss die `compute:`-Methode implementieren. In dieser Methode findet die eigentliche Automatikfunktion statt. Sie erhält als Parameter einen `SamState`, also ein Objekt, in dem sämtliche relevanten Zustandsinformationen des Trackings[17] enthalten sind. Diese Informationen werden je nach Aufgabe des Agenten verarbeitet und das Ergebnis wird, wiederum als `SamState`, von der `compute:`-Methode zurückgeliefert.

Definiert ein Agent Eigenschaften über die geerbten hinaus, so muss er auch die Methoden `getAllProps` und `setAllProps:` erweitern. Sie bilden die Schnittstelle, über die alle Eigenschaften des Agenten abgefragt bzw. gesetzt werden können, und werden z. B. zum Speichern und Wiederherstellen von Automatiken benötigt.

Wie ein einsatzfähiger Agent zur Verwendung ins Automatiken-GUI integriert wird, erklärt Anhang F.

4.5.2 *Graphen*

Graphen ermöglichen es, aus den Agenten durch Kombination komplexere Automatiken zu erstellen. Dabei bilden logisch gesehen die einzelnen Agenten die Knoten.[18] Die Abarbeitungsreihenfolge wird durch gerichtete Kanten definiert. Da jeder Agent einen `SamState` als Eingabe erhält und als Ausgabe wieder einen `SamState` liefert, können die Agenten in beliebiger Reihenfolge zusammengeschaltet werden.

Die Graphen werden durch die Klasse `AAFGraph` implementiert. Ein `AAFGraph` besteht aus je genau einem Wurzel- und Senkenknoten und beliebig vielen sonstigen Knoten. Ein Wurzelknoten darf keine Vorgänger (`parents`) haben, jedoch beliebig viele Nachfolger (`children`). In einem Senkenknoten dürfen beliebig viele Kanten zusammenlaufen, er darf aber keine Nachfolger haben. Die übrigen Knoten haben genau einen Vorgänger und einen Nachfolger. Über den Wurzelknoten wird der `SamState`, wie er aus SAM ausgelesen wird, in den Graphen eingespeist.

17 s. hierzu Abschnitt 2.3, Beschreibung der Untersuchungsumgebung SAM

18 Tatsächlich bilden die Knoten in AAF nur die Struktur des Graphen ab, die Automatikfunktion wird von ihnen an die Agenten delegiert.

Über den Senkenknoten wird das Objekt vom Typ `SamState`, welches das Ergebnis der Automatikausführung ist, wieder in SAM zurückgeschrieben.

Alle Knoten haben als gemeinsame Basisklasse `AAFBaseNode`. Die für das GUI relevanten Instanzvariablen, die sie von dieser Klasse erben, sind die Referenzen auf Vorgänger- und Nachfolgerknoten, über die die gerichteten Kanten des Graphen realisiert werden. Von `AAFBaseNode` werden insgesamt 3 weitere Knotentypen abgeleitet, nämlich `AAFMuxNode`, `AAFDemuxNode`, `AAFNode`.

`AAFMuxNode` ist der Knotentyp, der für den Wurzelknoten benötigt wird. Seine Menge von Vorgängern ist stets leer und die Menge von Nachfolgern kann beliebig groß sein.

Der Senkenknoten ist immer vom Typ `AAFDemuxNode`. Ein solcher Knoten darf eine beliebig große Menge von Vorgängern haben, jedoch muss die Menge seiner Nachfolger immer leer sein. Eine wichtige Aufgabe, die einem `AAFDemuxNode` zukommt, ist, die Ergebnisse aller seiner Vorgänger zu einem einzigen Ergebnis zu verrechnen. Dieses ist dann das Ergebnis der gesamten Automatik.

Alle übrigen Knoten sind vom Typ `AAFNode`. Dies sind die Knoten, die gemeinsam die eigentliche Automatikfunktion bilden. Jeder dieser Knoten hat noch ein zusätzliches Member `delegate`, welches einen Verweis auf einen der oben bereits erläuterten Agenten enthält. Dieser Agent führt die eigentliche Verarbeitung des `SamStates` aus, der durch den Knoten läuft.

Von den `AAFNodes` abgeleitet ist noch eine besondere Art von Knoten: die `AAFComplexNodes`. Sie funktionieren im Prinzip genauso wie die `AAFNodes`. Ihre Variable `delegate` verweist jedoch nicht auf einen Agenten, sondern wiederum auf ein Objekt vom Typ `AAFGraph` und damit auf eine komplexe Automatik. Diese Variante von Knoten ist deshalb möglich, weil der hier relevante Teil der Schnittstelle von `AAFAgent` und `AAFGraph` gleich ist. Beide verarbeiten einen `SamState` und generieren einen `SamState`. So lassen sich bereits bestehende komplexe Automatiken in neue Automatiken einbauen, ohne dass die Interna der Automatik eine Rolle spielen; auf oberster Ebene verhalten sie sich genauso wie Elementar-Automatiken.

4.6 XML-SPEICHERFORMAT

Um Automatiken persistent abzuspeichern, werden sie in Dateien geschrieben. Hierfür wurde ein Format definiert, in dem die Informationen zu einer Automatik so abgelegt werden können, dass daraus die vollständige Rekonstruktion möglich ist. Die Wahl fiel hierfür auf die Nutzung der in Abschnitt 2.4 bereits vorgestellten Beschreibungssprache XML. Der wichtigste Grund

für diese Entscheidung war, dass es sich bei XML um einen etablierten Quasi-Standard[19] zur Beschreibung von Daten handelt, weswegen es für viele Programmiersprachen, u. a. für Squeak, bereits Unterstützung zum Schreiben und Lesen von XML gibt. Dadurch wird die Verwendung in eigenen Programmen stark erleichtert. Im Folgenden wird das entworfene Format detailliert beschrieben, ein Beispielgraph ist in Abbildung 15 zu sehen.

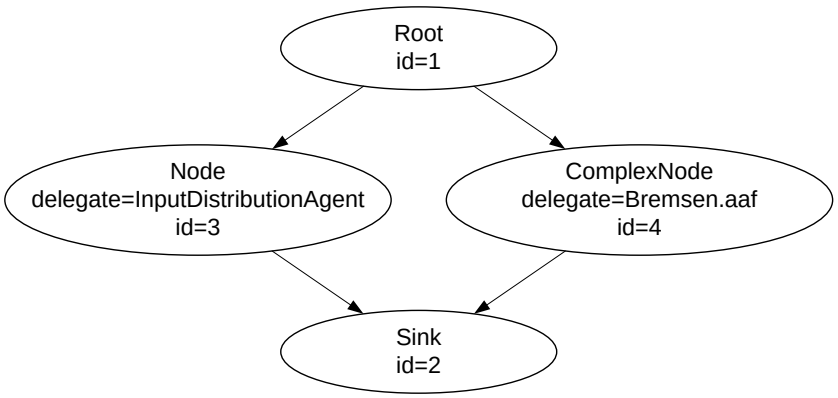


Abbildung 15: Beispielgraph zu den XML-Listings

Das *graph*-Element bildet das Wurzelelement eines XML-Dokuments. Es hat keine Attribute und umschließt die Elemente, in denen die Eigenschaften des Graphen sowie die Wurzel- und Senkenknoten, alle übrigen Knoten und die Verbindungen der Knoten untereinander beschrieben werden. Ein Beispiel hierfür ist in Listing 4 zu sehen.

Wurzel- und Senkenknoten des Graphen werden durch die Elemente *root* und *sink* codiert, die übrigen Knoten als *node* und *complexNode* (Listings 5 bis 8).

Allen diesen Knoten-Elementen ist gemein, dass sie die Elemente *id*, *label* und *position* umschließen müssen. Bei diesen handelt es sich um leere Elemente, welche ihre Informationen in einem obligatorischen Attribut enthalten. Bei *id* ist im *value*-Attribut die eindeutige Nummer des Knotens innerhalb des Graphen abgelegt. Der Anzeigename des Knotens wird als *name*-Attribut des *label*-Elements gespeichert und die angezeigte Position als *x*- und *y*-Attribute von *position*.

Die Elemente *node* und *complexNode* (Listings 7 und 8) enthalten jeweils noch ein weiteres Element für ihr `AAFDelegate`[20]. *node* speichert seinen Agenten in einem *agent*-Element. Dieses legt in einem *type*-Attribut ab, um welchen Typ von Agent es sich handelt und umschließt ein Element, in dem seine Eigenschaften gespeichert sind (Listing 9). *complexNode* hat als `AAFDelegate` einen

19 Die XML-Spezifikation ist kein Standard, sondern eine *W3C Recommendation*, also eine Empfehlung. Diese hat jedoch in etwa die Bedeutung eines Standards und kommt auf breiter Ebene zum Einsatz.

20 Zur Beschreibung von `AAFDelegate` s. Abschnitt 4.5.

```
<?xml  version="1.0" encoding="utf-8" ?>
<!DOCTYPE graph SYSTEM "aafgt.dtd">

<graph>
  <properties>
    <proplist key="inactiveValues">
      <propval value="false"/>
      <propval value="false"/>
      <propval value="false"/>
    </proplist>
  </properties>

  <root>
    ...
  </root>

  <node>
    ...
  </node>

  <complexNode>
    ...
  </complexNode>

  <sink>
    ...
  </sink>

  <connection .../>
  ...
  <connection .../>

</graph>
```

Listing 4: Das Wurzelelement *graph* umschließt alle weiteren Elemente.

```
<root>
    <id value="1"/>
    <label name="Root"/>
    <position x="398" y="10"/>
</root>
```

Listing 5: XML-Darstellung eines Wurzelknotens

```
<sink>
    <id value="2"/>
    <label name="Sink"/>
    <position x="398" y="843"/>
</sink>
```

Listing 6: XML-Darstellung eines Senkenknotens

```
<node>
    <id value="3"/>
    <label name="InputDistribution"/>
    <position x="140" y="554"/>
    <agent type=...>
    ...
    </agent>
</node>
```

Listing 7: XML-Darstellung eines einfachen Knotens

```
<complexNode>
    <id value="4"/>
    <label name="Bremsen.aaf"/>
    <position x="85" y="321"/>
    <graph>
    ...
    </graph>
</complexNode>
```

Listing 8: XML-Darstellung eines komplexen Knotens

Graphen, der in einem *graph*-Element abgelegt wird. Dieses folgt dem gleichen Aufbau wie das Wurzelelement. An dieser Stelle wird also die in einem Graphen mögliche Rekursion abgebildet.

Die Eigenschaften eines Graphen oder Agenten werden in einem *properties*-Element gruppiert. Dieses besitzt keine Attribute und umschließt mindestens ein *prop*- oder *proplist*-Element. In diesen werden schließlich die tatsächlichen Eigenschaften gespeichert. Beim *prop*-Element handelt es sich um ein leeres Element mit einem zwingenden Attributpaar *key* und *value*, welches den Namen einer Eigenschaft sowie ihren konkreten Wert enthält. Mit einem *proplist*-Element lässt sich eine Eigenschaft abspeichern, die aus einer Liste von Werten besteht. Das *proplist*-Element speichert den Namen der Eigenschaft wiederum in einem Attribut *key*, die Werte stehen in leeren *propval*-Unterelementen, welche die Werte in *value*-Attributen speichern.

```
<agent type="AAFInputDistributionAgent">
    <properties>
        <proplist key="inactiveValues">
      <propval value="false"/>
      <propval value="true"/>
      <propval value="false"/>
    </proplist>
        <prop key="mwi1" value="0.5"/>
        <prop key="mwi2" value="0.5"/>
    </properties>
</agent>
```

Listing 9: XML-Darstellung eines Agenten

Ganz zuletzt vor dem schließenden *graph*-Tag werden die Verbindungen der Knoten im Graphen angegeben[21] (Listing 10). Hierfür kommt pro Verbindung ein *connection*-Tag zum Einsatz. Dieses ist wieder ein leeres Element, das die Informationen in zwei zwingenden Attributen *parent* und *child* beinhaltet. Die Werte dieser Attribute sind die IDs der verbundenen Knoten.

```
1 <connection parent="1" child="3"/>
2 <connection parent="1" child="4"/>
3 <connection parent="3" child="2"/>
4 <connection parent="4" child="2"/>
```

Listing 10: XML-Darstellung von Verbindungen

Das beschriebene Format wurde als XML-Schema formal definiert. Es wird am Anfang jeder XML-Datei referenziert und ist in Anhang C zu finden. Durch die Beschreibung als XML-Schema ist das Format eindeutig und in einer standardisierten Form spezifiziert. Dadurch wird vermieden, dass Interpretationsspielraum durch Unklarheiten bei rein verbaler Beschreibung entsteht. Zudem ist es möglich, mit einem geeigneten XML-Parser eine gespeicherte Automatik gegen dieses XML-Schema zu validieren, also auf Einhaltung aller Regeln zu prüfen. Auch kann ein solcher Parser anhand des Schemas bereits zahlreiche Typüberprüfungen von Werten durchführen, die dann nicht mehr im Programmcode realisiert werden müssen. Dieser wird dadurch übersichtlicher und weniger fehleranfällig.

Leider bietet der Squeak-SAX-Parser keine Unterstützung für XML-Schema.[22] Es wurde dennoch entschieden, die Referenzierung des XML-Schemas mit in die XML-Dateien aufzunehmen. Auf diese Weise können die Dateien schon beim Einlesen validiert werden, sobald ein passender Parser zum Einsatz kommt, sei es wegen einer Umstellung des Projekts auf eine andere Sprache wie z. B. Java[23], sei es, weil der Squeak-Parser in zukünftigen Zeiten die passende Unterstützung bietet.

Die starke Nutzung von XML-Attributen anstelle von Unterelementen ist dadurch begründet, dass es beim Squeak-SAX-Parser wesentlich einfacher ist, auf die Attribute eines Elements zuzugreifen als festzustellen, von welchem Eltern-Element ein Element umschlossen wird.

21 Sie müssen deswegen ganz hinten stehen, weil zum Zeitpunkt ihres Einlesens bereits alle Elemente im Programm erzeugt sein müssen. Andernfalls führt der Versuch die Elemente anhand ihrer IDs zu verbinden zu einem Fehler.

22 Auch die Unterstützung für DTD (Document Type Definition), die einige Methodennamen erahnen lassen, funktioniert nicht. Ebenso verhält es sich beim DOM-Parser.

23 Es gibt bereits eine teilweise Reimplementierung der SAM-Kompente in Java, die in der Diplomarbeit von Michael Hildebrandt (2009) beschrieben wird.

4.7 IMPLEMENTIERUNG

In den folgenden Abschnitten wird die Implementierung des Automatiken-GUI vorgestellt. Dabei ist zu beachten, dass Klassen, deren Namen mit `AAFGT` oder `AAF` beginnen, im Zuge dieser Arbeit implementiert wurden, während Squeak die übrigen Klassen bereitstellt.

In dieser Arbeit werden die Funktion und die Arbeitsweise von Klassen und Methoden vorgestellt. Der zugehörige Quelltext befindet sich vollständig auf der mit dieser Arbeit eingereichten DVD (Anhang H).

4.7.1 *Layout der Oberfläche*

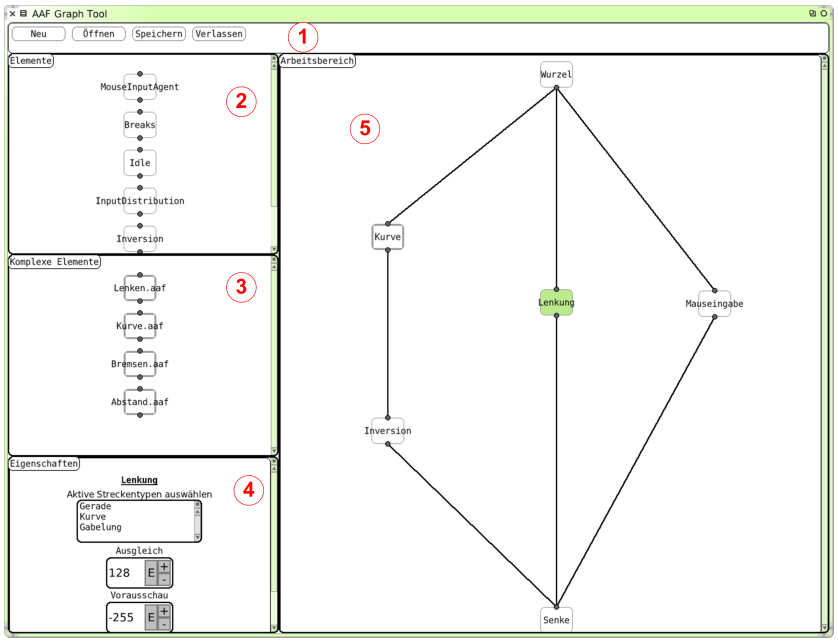


Abbildung 16: Die Oberfläche des Automatiken-GUI ist in mehrere Bereiche aufgeteilt, von denen jedem eine bestimmte Funktion zukommt.

Die Oberfläche des Automatiken-GUI ist in mehrere Bereiche eingeteilt, die in Abbildung 16 dargestellt sind. Ganz oben im Programmfenster befindet sich der Menübereich (1), welcher die Schaltflächen *Öffnen*, *Speichern*, *Verlassen* und *Neu* aufnimmt. *Neu* bedeutet hierbei, dass das GUI wieder in den Zustand gebracht wird, den es auch direkt nach dem Starten hat. Darunter befinden sich auf der linken Seite drei Bereiche. Im oberen (2) und mittleren (3) werden die Elemente angezeigt, aus denen der Benutzer seine Automatiken aufbauen kann. Bei den Elementen im oberen Bereich handelt es sich um die Repräsentationen der elementaren Automatiken, also der Agenten. Im mittleren Bereich sind die Darstellungen der komplexen Automatiken zu sehen,

die in weiteren Automatiken wiederverwendet werden können. Der große Bereich auf der rechten Seite bildet die Arbeitsfläche (5). Hier fügt der Benutzer durch Ziehen mit der Maus Elemente aus den beiden eben beschriebenen Bereichen ein und kann sie so anordnen und miteinander verbinden, dass die von ihm gewünschte Automatik entsteht. Außerdem kann er auch die Eigenschaften der Elemente bearbeiten, hierbei kommt der noch nicht erläuterte unterste Bereich auf der linken Seite zum Einsatz (4). Klickt der Benutzer im Arbeitsbereich ein Element an, so wird dieses markiert und seine Eigenschaften werden im Bereich links unten dargestellt. Hier können sie auch bearbeitet werden. Details zur Umsetzung der einzelnen Bereiche werden in den folgenden Abschnitten vorgestellt.

4.7.2 *Basismorph*

Die Klasse `AAFGTBaseMorph` ist eine Unterklasse von `BorderedMorph` und bildet die Basisklasse, von der alle weiteren rechteckigen Morphe für das Automatiken-GUI abgeleitet werden. Im Wesentlichen handelt es sich bei dieser Klasse auch nur um einen `BorderedMorph`, also einen Morph, der eine rechteckige Form und eine Umrandung ausweist. Es war jedoch nicht möglich, direkt `BorderedMorph` als Basisklasse für die Morphe des Automatiken-GUI zu verwenden, da der Aufruf der Methode `perform:orSendTo:` an einigen dieser Morphe notwendig ist. Weil diese Methode jedoch in der verwendeten Squeak-Version fehlerhaft implementiert ist, war es notwendig sie zu überschreiben. Um das nicht für jede betroffene Morph-Klasse einzeln machen zu müssen und so Code-Duplikate zu erzeugen, wurde die Klasse `AAFGTBaseMorph` sozusagen als Zwischenschicht implementiert. In dieser wird die fehlerhafte Methode durch eine korrigierte Fassung überschrieben. Sie wird an alle ihre Unterklassen weitervererbt, sodass die `perform:orSendTo:`-Aufrufe sich bei diesen nun korrekt verhalten. Darüber hinaus wurde die Klasse dafür verwendet, bereits alle Eigenschaften zu setzen, die die abgeleiteten Morph-Klassen gemeinsam haben. Auf diese Weise wurde auch hier Code-Redundanz vermieden, welche die Wartung unnötig erschweren würde.

4.7.3 *Anwendungsfenster*

Das Anwendungsfenster ist ein Objekt der Klasse `AAFGTMainWindow`. Diese ist von der Squeak-Klasse `SystemWindow` abgeleitet, wodurch sich das Automatiken-GUI in Erscheinung und grundlegender Bedienung nahtlos in die Squeak-Umgebung einfügt (Abbildung 17).

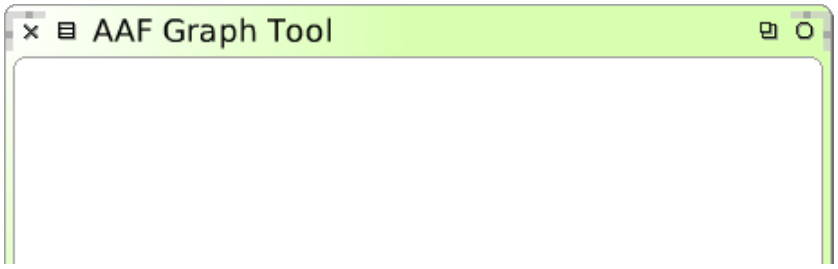


Abbildung 17: Das Fenster des Automatiken-GUI fügt sich nahtlos ins Look&Feel von Squeak ein.

Die Ableitung einer eigenen Klasse wurde der direkten Verwendung der Squeak-Klasse vorgezogen, weil dadurch erreicht werden konnte, dass das Schließen der Anwendung über das Kreuz in der Titelleiste exakt dieselbe Aktion auslöst wie das Betätigen der *Verlassen*-Schaltfläche in der Menüleiste. Unabhängig davon, über welchen der Wege der Benutzer das Automatiken-GUI verlässt, geschieht das Schließen in identischer Weise. Dies ist vor allem deshalb wichtig, weil so das Prüfen auf noch nicht gespeicherte Veränderungen in jedem Fall sichergestellt wird.

Eine weitere wichtige Funktion der Klasse ist die Initialisierung des Automatiken-GUI. Die einzelnen in Abschnitt 4.7.1 bereits erwähnten Bereiche werden erzeugt und als Inhalt des Systemfensters bereitgestellt. Hierfür bedient sich das Anwendungsfenster der Haupt-Panel-Klasse, welche im folgenden Abschnitt beschrieben wird. Anschließend steht das GUI zur Benutzung zur Verfügung.

4.7.4 *Haupt-Panel*

Das Haupt-Panel ist in der Klasse `AAFGTMainPanel` implementiert und bildet einen Container für die verschiedenen Bereiche des GUI. Sie werden in ihm angeordnet, bevor dieser Container als Ganzes dem Anwendungsfenster hinzugefügt wird. Der Weg über eine solche Zwischenstufe wurde der Anordnung direkt im Anwendungsfenster vorgezogen, weil er das Platzieren der einzelnen Bereiche erleichtert.

Weiterhin ist das Haupt-Panel für einige Verwaltungsinformationen zuständig, auf die alle oder zumindest mehrere der Unterbereiche zugreifen können müssen. So besitzt es ein sogenanntes *Dirty Bit*[24], in welchem angezeigt wird, ob im Automatiken-GUI nicht gespeicherte Änderungen vorliegen.

Ebenso wurden die Methoden zum Speichern und Laden von Automatiken sowie zum Zurücksetzen in den Anfangszustand

24 Hierbei handelt es sich nicht wirklich um ein Bit, sondern um ein boolesches Objekt.

und Schließen des GUI in der Haupt-Panel-Klasse implementiert. Logisch gehören diese zwar zum Menübereich. Da sie jedoch auch mit anderen Bereichen interagieren müssen, auf die ein Zugriff vom Haupt-Panel aus leichter möglich ist, wurden sie hier implementiert. Eine nähere Erläuterung ihrer Funktionsweise findet dennoch bei der Vorstellung des Menübereichs statt, da sie diesen funktional realisieren.

Eine letzte Aufgabe, die das Haupt-Panel übernimmt, ist das Überschreiben der Squeak-Methode, welche dafür sorgt, Morphe der aktuellen Squeak-Welt hinzuzufügen, und sie so für den Benutzer zugänglich zu machen. Diese Anpassung war notwendig, da die originale Methode Teile des GUI fehlplatziert hat, was auf die Behandlung von Positionsangaben in Squeak zurückzuführen ist.[25]

4.7.5 *Menübereich*

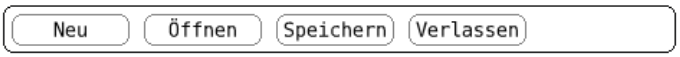


Abbildung 18: Der Menübereich nimmt die Schaltflächen zur Steuerung des GUI auf.

Der Menübereich wird durch die Klasse `AAFGTMenuArea` implementiert (Abbildung 16 (1), Abbildung 18). Die Aufgabe des Menübereichs ist, die Schaltflächen aufzunehmen, über die der Benutzer Automatiken speichern und laden und das Automatiken-GUI in den Startzustand zurückversetzen (*Neu*) oder verlassen kann.

Die Funktionalität der Menübereich-Klasse besteht darin, Schaltflächen zu erzeugen und anzuordnen. Bei der Erzeugung werden für die Schaltflächen nicht nur grafische Eigenschaften sowie Beschriftungen gesetzt, sondern sie werden auch jeweils mit einer Methode verbunden, welche die zugehörige Aktion ausführt.[26] Die Umsetzung dieser Aktionen wird im Folgenden erläutert.

Speichern von Automatiken

Wählt der Benutzer die *Speichern*-Schaltfläche an, wird die Methode zum Schreiben einer Automatik aufgerufen.

25 Je nachdem, ob ein Morph bereits als Teil der Welt sichtbar ist oder nicht, werden die Angaben auf einen unterschiedlichen Bezugspunkt angewendet.

26 Implementiert sind diese aus den in Abschnitt 4.7.4 dargelegten Gründen an anderer Stelle, hier findet nur die Verknüpfung statt.

Diese ermittelt zunächst das Speicherverzeichnis, welches durch eine Einstellung des Automatiken-GUI vorgegeben ist. Auf diese Weise muss der Benutzer nicht mithilfe des etwas unhandlichen Datei-Auswahldialogs, welchen Squeak bietet, im Dateisystem nach einem Speicherort suchen. Außerdem wird durch dieses Vorgehen dafür gesorgt, dass sich alle abgespeicherten Automatiken dort befinden, wo das SAM-System sie benötigt.

Anschließend wird der Benutzer nach einem Namen für die Datei gefragt. Ist bereits eine Datei mit dem gewünschten Namen vorhanden, so bietet das Automatiken-GUI an, einen anderen Namen anzugeben, die Datei zu überschreiben oder aber den Speichervorgang abzubrechen.

Sobald geklärt ist, wie die Datei heißen soll, wird die aktuelle Automatik in diese gespeichert. Hierfür kommt ein eigens für die Automatiken-Repräsentation entworfenes XML-Format zum Einsatz, welches in Abschnitt 4.6 vorgestellt wird. Wie das Schreiben der Daten in diesem Format konkret realisiert ist, erläutert Abschnitt 4.7.16.

Laden von Automatiken

Betätigt der Nutzer die *Öffnen*-Schaltfläche zum Laden einer Automatik, so wird zunächst geprüft, ob es Änderungen an der aktuellen Automatik gibt, welche noch zu speichern sind. Ist dem so, bietet das GUI die Wahl zwischen Speichern und Verwerfen der Änderungen an.

Sind keine Daten (mehr) zu sichern, so beginnt der eigentliche Ladevorgang. Hierfür wird zunächst das Verzeichnis ermittelt, in welchem die Automatiken gespeichert sind. Dieses ist durch die Anwendung vorgegeben. Aus diesem Verzeichnis werden alle zur Verfügung stehenden Automatiken ermittelt und dem Nutzer zur Auswahl angeboten.

Nachdem er eine ausgewählt hat, wird die entsprechende Datei eingelesen, die Automatik erzeugt und dem Arbeitsbereich zur Anzeige und Bearbeitung übergeben. Wie das Einlesen der gespeicherten Dateien und das Erzeugen der Automatik konkret umgesetzt sind, wird in Abschnitt 4.7.15 beschrieben.

Zurücksetzen und Verlassen des GUI

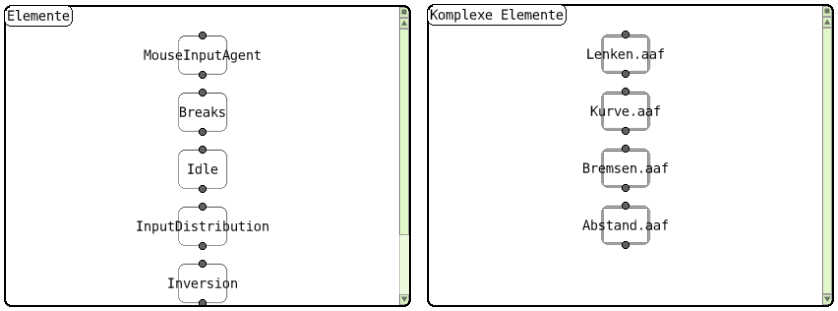
Wird eine der Schaltflächen *Neu* oder *Verlassen* angewählt, so wird zunächst geprüft, ob ungesicherte Änderungen vorliegen. Ist dem so, wird der Benutzer gefragt, ob er sie speichern oder verwerfen möchte.

Anschließend wird im Falle von *Neu* das GUI auf seinen initialen Zustand zurückgesetzt und es kann mit dem Entwurf einer

neuen Automatik begonnen werden. Fiel die Wahl auf *Verlassen*, so wird die Anwendung geschlossen.

4.7.6 *Bereich für einfache Elemente*

Die Aufgabe der Klasse `AAFGTElementArea` besteht darin, den Bereich aufzubauen und zur Verfügung zu stellen, der die Agenten als Elemente im Automatiken-GUI anbietet (Abbildung 16 (2), Abbildung 19a).



(a) Einfache Elemente (b) Komplexe Elemente

Abbildung 19: Die verfügbaren Elemente stehen in zwei Bereichen zur Auswahl.

Hierfür sind zunächst die Agenten festzustellen, die repräsentiert werden müssen. Dazu werden alle Ableitungen des Basis-Agenten `AAFAgent` ermittelt und anschließend wird jeder dieser Agenten daraufhin überprüft, ob er zum Einsatz in Automatiken freigegeben wurde. Zu diesem Zweck wird eine spezielle Eigenschaft der Agenten abgefragt, die verhindern soll, dass unfertige Agenten, welche sich noch in der Entwicklung befinden, verwendet werden.[27] Nachdem die darzustellenden Agenten ermittelt wurden, wird nun für jeden von ihnen ein Element erzeugt, welches dem Elementbereich hinzugefügt wird und damit zur Auswahl bereitsteht. Wie das Erzeugen der Elemente funktioniert, wird in Abschnitt 4.7.10 näher erläutert.

4.7.7 *Bereich für komplexe Elemente*

Die Klasse `AAFGTComplexElementArea` realisiert den Bereich, in dem die bereits existenten Automatiken zur Verwendung als komplexe Elemente zur Verfügung gestellt werden (Abbildung 16 (3), Abbildung 19b). Die Funktionsweise ist der in Abschnitt 4.7.6 beschriebenen sehr ähnlich.

Zuerst wird ermittelt, welche Automatiken bereits abgespeichert wurden. Diese befinden sich alle in einem vom Programm

27 Es liegt in der Verantwortung des Entwicklers der Agentenklasse, dass ein Agent nur dann freigegeben wird, wenn er tatsächlich einsatzfähig ist.

vorgegebenen Standardverzeichnis in Dateien mit der Endung `.aaf`. Anschließend wird jede dieser Automatiken daraufhin überprüft, ob sie sich in einem gültigen, also einsatzfähigen, Zustand befinden. Das ist notwendig, weil auch das Abspeichern noch unfertiger Automatiken während des Entwicklungsprozesses möglich sein soll. Sind alle anzuzeigenden Automatiken ermittelt, so wird für jede ein Element erzeugt und dem Bereich für komplexe Elemente hinzugefügt. Für die Erläuterung der Element-Erzeugung sei auch hier auf Abschnitt 4.7.10 verwiesen.

4.7.8 *Eigenschaftenbereich*

Der Eigenschaftenbereich des Automatiken-GUI wird durch die Klasse `AAFGTPropertyArea` implementiert. In ihm wird der Dialog eines markierten Elements angezeigt, sodass die entsprechenden Elementeigenschaften sichtbar werden und angepasst werden können (Abbildung 16 (4), Abbildung 20).

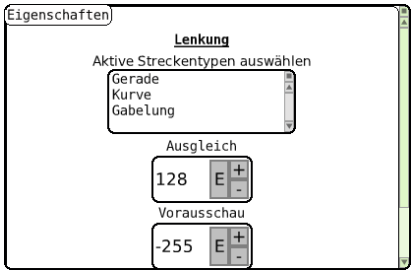


Abbildung 20: Im Eigenschaftenbereich können die Agenten konfiguriert werden.

Ist kein Element markiert, so bleibt der Eigenschaftenbereich leer. Sind mehrere gleichzeitig markiert, wird im Eigenschaftenbereich nur angezeigt, für welche Streckenabschnittstypen sie aktiv bzw. inaktiv sind, da dies die einzige Eigenschaft ist, die alle Elemente teilen. Auf diese Weise kann sie für mehrere Elemente gleichzeitig editiert werden.

Darüber hinaus bietet der Eigenschaftenbereich keine weitere Funktionalität. Es ist ein reiner Anzeigebereich, in den die Eigenschaften-Dialoge eingeblendet werden. Die eigentliche Darstellung und Bearbeitung der Eigenschaften ist in den Dialogklassen realisiert, welche in Abschnitt 4.7.14 beschrieben werden.

4.7.9 *Arbeitsbereich*

Der Arbeitsbereich ist in der Klasse `AAFGTDrawArea` realisiert. Seine Aufgabe ist es, dem Benutzer eine Fläche zur Verfügung zu stellen, in der er Automatiken aus den Elementen zusammenstellen kann (Abbildung 16 (5)).

Da jeder Graph genau ein Wurzel- und ein Senkenelement haben muss, werden diese bei der Initialisierung der Arbeitsfläche bereits angelegt. Sie sind vom Benutzer nicht veränderbar. Weitere Elemente fügt der Benutzer durch Auswahl aus den beiden Elementbereichen hinzu.

Auch wenn sie über eine Referenz auf den aktuell angezeigten und verarbeiteten Graphen verfügt, ist die Arbeitsfläche lediglich für die Anzeige desselben zuständig. Der Aufbau des Graph-Objekts während der Bearbeitung durch den Benutzer wird von anderen Klassen übernommen. Hier sei speziell die in Abschnitt 4.7.11 beschriebene Verbindungspunkt-Klasse genannt.

Mit der Aufgabe der Anzeige des aktuellen Graphen sind einige Verwaltungsinformationen verbunden, für die ebenfalls der Arbeitsbereich zuständig ist. So verfügt er über das Wissen darüber, welche Elemente aktuell markiert sind. Auch hat er Kenntnis darüber, auf welcher Verschachtelungsebene der Graph gerade angezeigt wird, falls er komplexe Elemente und somit Untergraphen enthält.

4.7.10 *Elemente*

Es gibt insgesamt drei Elementtypen, die die verschiedenen Arten von AAF-Graphenknoten im GUI repräsentieren. Zum einen sind das die Typen für die Graphenwurzel und -senke, welche die AAF-Knotentypen `AAFMuxNode` und `AAFDemuxNode` grafisch darstellen. Zum anderen ist das ein Typ, der die Knoten der Klassen `AAFNode` und `AAFComplexNode` umsetzt. Für Details zu den verschiedenen AAF-Knotentypen sei auf Abschnitt 4.5 verwiesen.

Die Implementierung der drei Elementtypen wird in den folgenden Unterabschnitten erläutert.

Basiselemente

Auch wenn die drei Elementtypen einige wichtige Unterschiede aufweisen, sodass eine Implementierung in getrennten Klassen sinnvoll erschien, haben sie dennoch viele Gemeinsamkeiten. Diese sind in der Klasse `AAFGTBaseElement` zusammengefasst, von der sie an die drei Elementtypen vererbt werden.

Zunächst sind einige Instanzvariablen gleich. Alle Elemente haben eine im GUI sichtbare Beschriftung, einen Verweis auf einen AAF-Knoten und eine Referenz auf einen Eigenschaften-Dialog.

Zwar besitzen derzeit nur Objekte vom Untertyp `AAFGTElement` Dialoge. Da jedoch bereits absehbar ist, dass zumindest die Elemente für Senkenknoten in Zukunft ebenfalls

Konfigurationsmöglichkeiten erhalten werden, wurde entschieden, die Dialoge in der Basisklasse zu verwalten. Auch ein Einsatz in Wurzelelementen ist auf lange Sicht denkbar. Der Aufbau der Eigenschaften-Dialoge wird in Abschnitt 4.7.14 beschrieben.

Eine weitere gemeinsame und daher in der Basisklasse umgesetzte Eigenschaft ist die optische Erscheinung von Elementen. Es bestehen lediglich kleine Detailunterschiede zwischen den Elementtypen, entsprechende Anpassungen finden in den spezialisierten Klassen statt.

Darüber hinaus teilen sich die verschiedenen Elementtypen einige Methoden, mit denen sie sich innerhalb der Morphstruktur des GUI orientieren können. Da ist zunächst eine Methode zu nennen, mit der ein Element feststellen kann, ob es sich im Arbeitsbereich befindet und ggf. eine Referenz auf diesen erhalten kann. Diese Funktionalität ist aus verschiedenen Gründen notwendig, als Beispiele seien die korrekte Positionierung, die Markierung mehrerer Elemente gleichzeitig oder das Entfernen eines Knotens aus dem Graphen genannt.

Über eine weitere Orientierungsmethode können die Elemente eine Referenz auf das Haupt-Panel erfragen. Diese ist notwendig für eine dritte gemeinsame Methode, die durch Setzen des bereichsübergreifenden *Dirty Bit* noch nicht abgespeicherte Änderungen am Graphen markiert. Näheres hierzu ist in Abschnitt 4.7.4 zu finden.

Die drei genannten Methoden sind nicht in der `AAFGTBaseElement`-Klasse selbst implementiert, sondern werden ihr durch sogenannte `Traits` zur Verfügung gestellt. Der Grund hierfür ist die Vermeidung von Code-Redundanzen, da die Methoden auch an anderer Stelle in gleicher Form benötigt werden. Auswirkungen auf die Verwendung innerhalb der Klasse bestehen jedoch nicht. Näheres zu `Traits` sowie zur Implementierung der genannten Methoden ist Abschnitt 4.8 zu entnehmen.

Einfache und komplexe Elemente

Die Klasse `AAFGTElement` hat die Aufgabe, Knoten vom Typ `AAFNode` und `AAFComplexNode` zu repräsentieren.

Zusätzlich zu den geerbten Eigenschaften und Methoden besitzen `AAFGTElement`-Objekte zwei anklickbare Verbindungspunkte vom Typ `AAFGTSnapPoint` am oberen und unteren Elementrand. Diese bilden Eingang und Ausgang des zugehörigen Graphenknoten ab und sorgen dafür, dass sich Elemente im GUI per Mausklick verbinden lassen. Die auf diese Weise erzeugten Kanten werden in die entsprechende Repräsentation im AAF-Graphen abgebildet. Näheres hierzu findet sich in Abschnitt 4.7.11.

Über eine weitere Methode lässt sich für ein `AAFGTElement`-Objekt der AAF-Knoten festlegen, der durch das Element repräsentiert wird. An dieser Stelle wird auch dafür gesorgt, dass Änderungen an diesem Knoten überwacht werden, um mit entsprechenden Aktualisierungen reagieren zu können. Beispiele hierfür sind die Anzeige einer geänderten Aufschrift oder Position. Auch werden komplexe Knoten durch einen dickeren Rahmen dargestellt als einfache.

Weitere Methoden der Klasse sind für die Ereignisbehandlung zuständig, wobei jeweils unterschieden werden muss, ob ein Ereignis innerhalb der Arbeitsfläche oder innerhalb eines der Elementbereiche stattgefunden hat, da abhängig davon eine unterschiedliche Aktion ausgelöst wird. Genaueres zu den behandelten Ereignissen und den durch sie ausgelösten Aktionen ist in Abschnitt G zu finden, in dem die Bedienung des GUI erläutert wird.

Wurzel- und Senkenelemente

`AAFGTMux`-Elemente und `AAFGTDemux`-Elemente repräsentieren die Wurzel- und Senkenknoten. Da jeder Graph genau eine Wurzel und Senke besitzt, enthält jede Anzeige eines Graphen jeweils genau ein Element der in diesem Abschnitt beschriebenen Elementtypen.

Wie auch die Elemente für einfache und komplexe Knoten (Abschnitt 4.7.10) haben Wurzel- und Senkenelemente anklickbare Verbindungspunkte vom Typ `AAFGTSnapPoint`, über die Kanten zu anderen Elementen erzeugt werden können. Da jedoch Wurzelelemente keine Vorgänger und Senkenelemente keine Nachfolger haben dürfen, besitzen sie jeweils nur einen solchen Punkt am entsprechenden Rand. Auch hier werden gezogene Kanten auf die entsprechenden AAF-Strukturen abgebildet; für die genauere Beschreibung der Funktionsweise sei auf Abschnitt 4.7.11 verwiesen.

Ebenfalls analog zu einfachen und komplexen Elementen lässt sich der zugehörige AAF-Knoten über eine entsprechende Methode setzen. Dabei ist auch hier zu beachten, dass Änderungen am Knoten überwacht werden müssen, um mit entsprechenden Aktualisierungen der Anzeige reagieren zu können.

Anders als für einfache und komplexe Elemente, welche sich prinzipiell überall auf der Arbeitsfläche befinden können, sind für die Wurzel- und Senkendarstellung feste Positionen vorgesehen. Diese sind im zugehörigen AAF-Knoten gespeichert, werden jedoch bei der Zuordnung zu einem entsprechenden Element überprüft und ggf. korrigiert. Durch die Ereignisbehandlung wird sichergestellt, dass der Benutzer des GUI diese Position nicht durch Verschieben mit der Maus verändern kann.

4.7.11 *Verbindungspunkte*

Die Klasse `AAFGTSnapPoint` erbt von `EllipseMorph` und dient, wie oben bereits erwähnt, dazu, die Elemente im Arbeitsbereich durch Mausklicks miteinander zu verbinden.

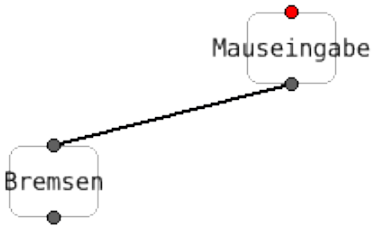


Abbildung 21: Über die Verbindungspunkte können Elemente miteinander verknüpft werden. Ist ein Punkt aktiviert, so wird er rot dargestellt.

Ein Objekt der Klasse `AAFGTSnapPoint` besitzt zwei wichtige Eigenschaften. Die erste zeigt an, ob es sich bei einem Verbindungspunkt um die Repräsentation eines Ein- oder Ausgangs handelt. Die zweite speichert, ob der Verbindungspunkt momentan aktiviert ist oder nicht. Aktiviert ist er dann, wenn er als Startpunkt einer Verbindung angeklickt wurde und nun darauf wartet, dass die Verbindung durch Klick auf einen weiteren Punkt hergestellt wird.

Das Herzstück der Verbindungspunkt-Klasse bildet die Ereignisbehandlung. Sie sorgt dafür, dass beim Verbinden der Elemente im Arbeitsbereich die entsprechenden Graphenstrukturen auf AAF-Ebene korrekt aufgebaut werden.

Klickt der Benutzer einen noch unverbundenen Verbindungspunkt-Morph an, während kein anderer aktiv ist, wird dieser aktiviert. Markiert wird dieser Zustand durch eine rote Einfärbung. Ist ein Punkt bereits verbunden, so kann er nicht mehr aktiviert werden. Da jeder Knoten nur einen Vorgänger und einen Nachfolger haben darf[28], steht er nicht mehr für eine Verbindung zur Verfügung. Ausnahmen bilden der Ausgang der Wurzel und der Eingang der Senke, diese dürfen beliebig oft verbunden sein.

Wird ein bereits aktiver Punkt angeklickt, so wird er wieder deaktiviert und nimmt seine ursprüngliche graue Farbe an.

Ist ein Punkt aktiviert, so kann ein weiterer angeklickt werden, um eine Verbindung herzustellen. Damit jedoch tatsächlich eine Kante in den Graphen eingefügt wird, müssen verschiedene Bedingungen erfüllt sein. So muss einer der Punkte ein Eingang und der andere ein Ausgang sein, denn Eingänge lassen sich ebenso wenig mit Eingängen verbinden wie Ausgänge mit Ausgängen. Auch ist es nicht zulässig, einen Verbindungspunkt mit

28 Diese Einschränkungen liegen in der Definition der Graphen im AAF begründet.

dem anderen an demselben Element zu verbinden oder sonstige Kreise zu erzeugen. Zu guter Letzt darf der angeklickte Punkt nicht schon verbunden sein. Ausnahmen bilden wieder der Ausgang der Wurzel sowie der Eingang der Senke.

Sind die Bedingungen zur Verbindung der Punkte erfüllt, so wird die entsprechende Kante unter Verwendung der passenden AAF-Methode in den repräsentierten Graphen eingefügt. Anschließend muss die neue Kante grafisch abgebildet werden. Hierfür wird eine entsprechende Linie erzeugt und der Arbeitsfläche hinzugefügt. Diese Linie ist ein Objekt der Klasse `AAFGTConnection`, welche in Abschnitt 4.7.12 beschrieben ist.

Sind alle diese Schritte abgeschlossen, so werden die beteiligten Verbindungspunkte inaktiviert und es kann auf dieselbe Weise die nächste Verbindung hergestellt werden.

4.7.12 *Kanten*

Die Klasse `AAFGTConnection` erbt von der Squeak-Klasse `PolygonMorph` und realisiert die Kanten des im GUI repräsentierten Graphen (Abbildung 21).

Um ihre Aufgabe erfüllen zu können, besitzt eine Kante Referenzen auf die durch sie verbundenen AAF-Knoten. Sie benötigt Kenntnis dieser Knoten, um beim Löschen der Kante im GUI ebenfalls die Verbindung zwischen den Knoten auf AAF-Ebene entfernen zu können. Zudem muss sie sich bei diesen als Beobachter registrieren, um ihre Darstellung an Positionsänderungen der verbundenen Knoten anpassen zu können. Weiterhin verwaltet eine Kante Referenzen auf die optisch verbundenen Elemente im GUI, um die Endpunkte für den anzuzeigenden Linienmorph ermitteln zu können.

Die Ereignisbehandlung beschränkt sich auf die Behandlung von Rechtsklicks. Diese öffnen ein Kontextmenü, über das der Benutzer eine Kante löschen kann. Die ausgelöste Aktion sorgt dafür, dass nicht nur die im GUI sichtbare Linie entfernt, sondern auch die Verbindung in der zugrunde liegenden AAF-Struktur gelöst wird. Damit die Ausführung der Menüoption korrekt funktioniert, musste an dieser Stelle wiederum die fehlerhafte Implementierung der von Squeak angebotenen Methode `perform:orSendTo:` durch eine korrigierte Fassung überschrieben werden (vgl. Abschnitt 4.7.2).

Zuletzt sei noch erwähnt, dass der Klasse `AAFGTConnection` ebenso wie den Elementklassen (Abschnitt 4.7.10) die durch `Traits` implementierten Methoden zur Orientierung in der Morphstruktur sowie zur Kennzeichnung nicht gespeicherter Änderungen zur Verfügung stehen. Für Details zu `Traits` und zur Implementierung der Methoden sei auf Abschnitt 4.8 verwiesen.

4.7.13 *Vorgabewerte*

Für die erleichterte Anpassung stehen Wertangaben für Größen, Farben, Positionen, Texte etc. nicht direkt an der Stelle im Quelltext, an der sie verwendet werden. Sie sind stattdessen in der Klasse `AAFGTConsts` gesammelt. Auf diese Weise ist es bei Änderungen nicht erforderlich den gesamten Quelltext zu durchsuchen, um den entsprechenden Wert zu finden. Außerdem werden Inkonsistenzen vermieden, wenn Werte an mehreren Stellen verwendet werden, weil keine notwendige Anpassung übersehen werden kann.

Da Squeak keine Konstanten bietet, wurde die Verwaltung der Wertangaben durch Klassenmethoden der Klasse `AAFGTConsts` implementiert, die ausschließlich unveränderliche Werte zurückliefern. Diese Umsetzung kommt Konstanten sehr nahe. Um einen Wert für die gesamte Anwendung zu ändern, muss lediglich der Rückgabewert der entsprechenden Klassenmethode angepasst werden.

4.7.14 *Eigenschaften-Dialoge*

Die Aufgabe der Eigenschaften-Dialoge ist es, dem Benutzer die Eigenschaften der in den Graphen-Knoten enthaltenen und in Abschnitt 4.5 erläuterten `AAFDelegate`-Objekte zugänglich zu machen (Abbildung 22). Diese `AAFDelegate`-Objekte übernehmen die eigentliche Automatikfunktion und somit kann diese über die Eigenschaften-Dialoge innerhalb der vorgesehenen Grenzen direkt beeinflusst werden.

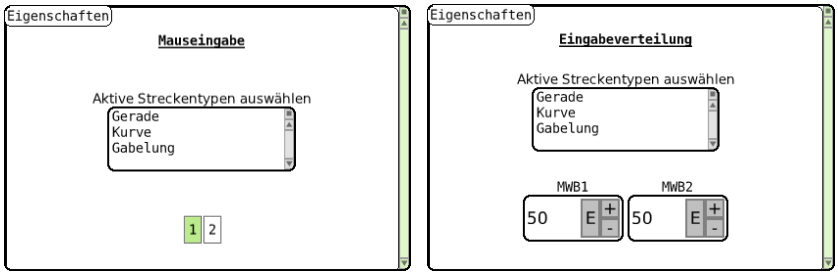


Abbildung 22: Für verschiedene Agenten werden verschiedene Eigenschaften-Dialoge bereitgestellt.

Jeder Eigenschaften-Dialog baut auf der Basisklasse `AAFAgentDialog` auf. Diese erzeugt eine Dialog-Überschrift sowie eine Liste für die Auswahl der Streckenabschnittstypen, für die das Element aktiv sein soll. Anhand der Überschrift kann der Benutzer den angezeigten Dialog eindeutig einem Element im Arbeitsbereich zuordnen. Weiterhin bereitet die Basisklasse einen Bereich vor, in dem abgeleitete Dialoge die ihnen eigenen

Bedienelemente platzieren können. Zudem verwaltet sie eine Referenz auf das zugehörige `AAFDelegate`-Objekt, welche für das Auslesen und Setzen der Eigenschaften notwendig ist.

Bietet ein `AAFDelegate`-Objekt weitergehende Eigenschaften an, so muss ein spezialisierter Dialog von der Basisklasse abgeleitet werden. Dieser muss zwei Methoden implementieren, die er zusätzlich zu denen der Basisklasse aufruft, nämlich eine, welche die Bedienelemente für die zusätzlichen Eigenschaften anlegt und platziert, sowie eine weitere, die für das Aktualisieren der angezeigten Werte bei Änderungen am `AAFDelegate`-Objekt sorgt.

Eine Ausnahme stellt der Fall dar, in dem das `AAFDelegate`-Objekt kein Agent, sondern ein Graph ist. In diesem Fall wird der Eigenschaften-Dialog ebenfalls von der genannten Basisklasse abgeleitet. Er implementiert jedoch nicht die beiden oben genannten zusätzlichen Methoden, sondern stellt eine Schaltfläche zur Verfügung, über die der Benutzer in den Graphen hinein verzweigen kann, sodass nun dieser in der Arbeitsfläche angezeigt wird und bearbeitet werden kann.

Für die Erzeugung eines Dialog-Objekts ist eine spezielle Hilfsklasse zuständig. Diese besitzt eine Klassenmethode, welcher ein `AAF`-Knoten als Parameter übergeben wird. Anhand dieses Knotens ermittelt sie den Typ des enthaltenen `AAFDelegate`-Objekts und erzeugt den passenden Dialog.

Im Folgenden wird exemplarisch die Implementierung des spezialisierten Dialogs zur Einstellung visueller Hinweise vorgestellt.

Dialog zur Einstellung visueller Hinweise

Da während der Entstehung dieser Arbeit noch kein Agent für visuelle Hinweise zur Verfügung stand, wurde zu Demonstrationszwecken die Agenten-Klasse `AAFVisualHintsDemoAgent` implementiert. Diese besitzt keine echte Funktionalität. Sie dient lediglich der Skizzierung eines Dialogs, mit dem visuelle Hinweise konfiguriert werden können. Es soll an dieser Stelle ausdrücklich darauf hingewiesen werden, dass die Möglichkeiten visueller Hinweise vielfältig sind und der Dialog stark vom zugrunde liegenden Agenten abhängt. Daher darf die hier gezeigte Lösung nur als eine Möglichkeit von vielen verstanden werden.

Ein `AAFVisualHintsDemoAgent` besitzt als Eigenschaft zunächst das Bild des SAM-Streckenabschnitts, in dem der Hinweis angezeigt werden soll.[29] Weitere Eigenschaften verwalten ein Bild, welches den darzustellenden Hinweis enthält, sowie

29 Genau genommen handelt es sich um zwei Streckenabschnitte. So ist es möglich, Ansichten zusammenzusetzen, die nicht als einzelnes Bild vorliegen, während einer Untersuchung aber auftreten können.

die Koordinaten, die festlegen, wo der Hinweis im Streckenbild angezeigt werden soll.

Diese Eigenschaften müssen durch den passenden Dialog bearbeitet werden können. Hierzu wurden die beiden bereits weiter oben erwähnten Methoden implementiert, welche den Dialog aufbauen und die angezeigten Werte aktuell halten.

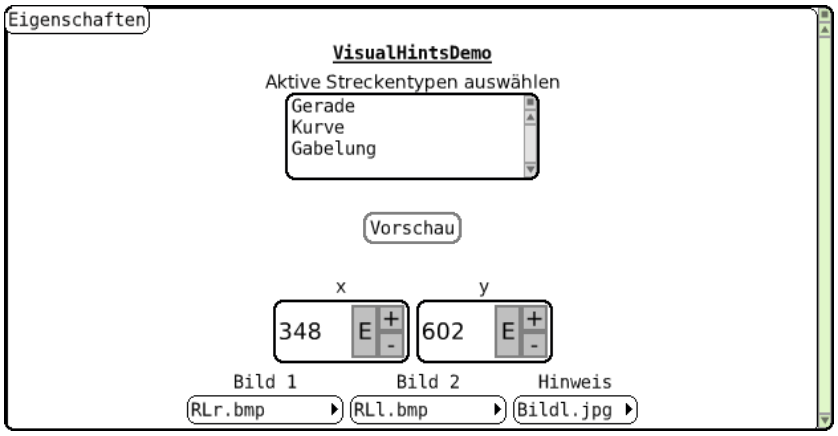


Abbildung 23: Dialog für visuelle Hinweise

Wie in Abbildung 23 zu sehen ist, können über den Dialog die verwendeten Streckenabschnitte sowie der anzuzeigende Hinweis ausgewählt werden. Zwei Spin-Button-Elemente ermöglichen es, den Hinweis innerhalb des Streckenbildes in vertikaler und horizontaler Richtung zu verschieben.

Über die *Vorschau*-Schaltfläche kann eine Vorschau eingeblendet werden, die zeigt, wie der Hinweis mit den aktuellen Einstellungen in der Untersuchung angezeigt würde. Zur Erleichterung der Positionierung ist es möglich, den Hinweis mit der Maus direkt zu platzieren, indem er per Drag&Drop verschoben wird. Die Vorschau zum Dialog in Abbildung 23 ist in Abbildung 24 zu sehen.

4.7.15 *Lesen von XML-Dateien: XML-Parser*

Zum Einlesen der XML-Dateien wird ein SAX-Parser verwendet, der durch Ableitung aus der von Squeak bereitgestellten Klasse `SAXHandler` entstanden ist. Die Wahl des SAX-Parsers ist dadurch begründet, dass dieser immer nur die gerade verarbeiteten Daten im Speicher hält und somit beliebig große XML-Dateien verarbeiten kann. Auf diese Weise wird die mögliche Größe und damit Komplexität einer Automatik nicht unnötigerweise durch den Parser beschränkt.

Wie in Abschnitt 2.4 bereits erläutert, arbeitet ein SAX-Parser ereignisorientiert, ruft also beim Eintreten bestimmter Situationen wie dem Erreichen des Elementanfangs oder -endes entsprechende Behandlungsroutinen auf. Abhängig davon, durch

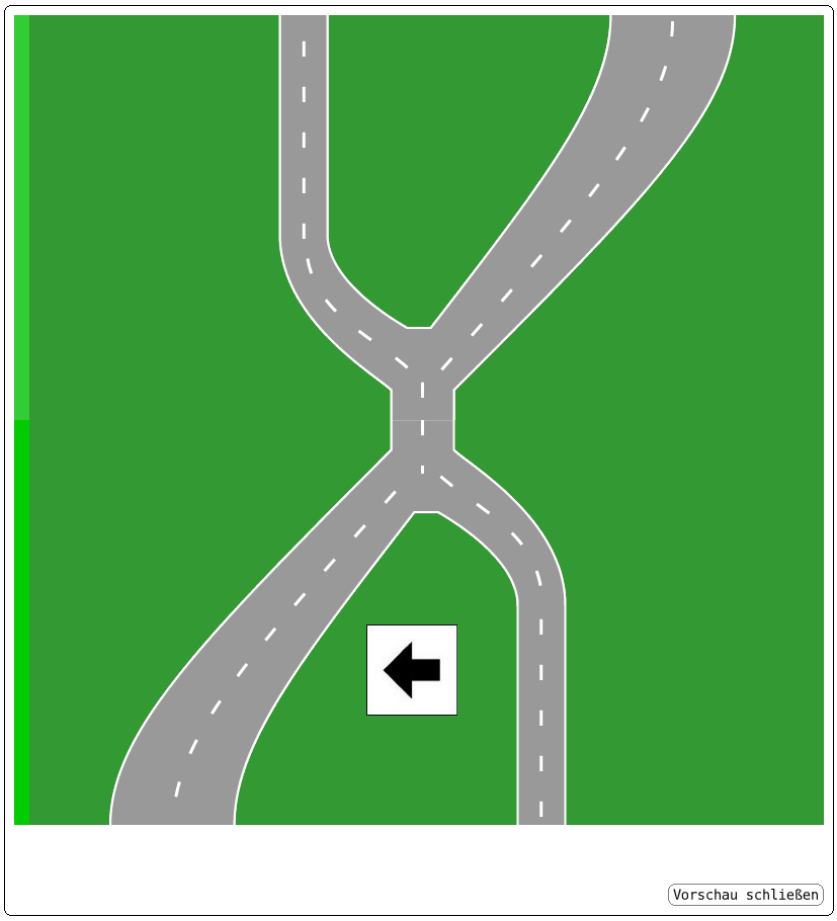


Abbildung 24: Vorschau für visuelle Hinweise

welches XML-Element das Ereignis ausgelöst wurde und welche Attribute es trägt, werden bestimmte Aktionen ausgeführt wie das Erzeugen oder Verbinden von Knoten sowie das Setzen von Eigenschaften für die in den Knoten enthaltenen `AAFDelegate`-Objekte. Auf diese Weise wird während des Lesens des XML-Dokuments die Graphen-Struktur so aufgebaut und konfiguriert, wie sie abgespeichert wurde, und steht am Ende des Einlese-Vorgangs für die weitere Nutzung zur Verfügung.

Für das Setzen der `AAFDelegate`-Eigenschaften kommt eine Hilfsfunktion zum Einsatz, welche die als Zeichenketten eingelesenen Werte jeweils in die korrekten Typen umwandelt. Zu bevorzugen wäre die Nutzung von XML-Schema-Definitionen für diesen Zweck, sodass bereits der Parser die Korrektheit der Typen gewährleisten kann. XML-Schema wird jedoch vom Squeak-Parser nicht unterstützt.

Die starke Nutzung von Attributen anstelle von Unterelementen beim Entwurf des in Abschnitt 4.6 beschriebenen Speicherformats ist damit begründet, dass es beim Squeak-SAX-Parser wesentlich einfacher ist auf Attribute zuzugreifen als festzustellen, in welchem Kontext ein Unterelement steht.

Details zum XML-Speicherformat der durch den Parser gelesenen Dateien finden sich in Abschnitt 4.6.

4.7.16 *Schreiben von XML-Dateien*

Soll eine Automatik in eine XML-Datei geschrieben werden, so wird am entsprechenden `AAFGraph`-Objekt die Methode `printXMLOn:` aufgerufen. Dieser wird ein Objekt der von Squeak mitgelieferten Klasse `XMLWriter` als Parameter zur Verfügung gestellt, welches dafür sorgt, dass die Ausgaben in die gewünschte Datei erfolgen. Außerdem stellt es sicher, dass die XML-Elemente im korrekten Format erzeugt werden, und übernimmt die Einrückungen.

Die Schreibmethode der Graphen-Klasse gibt nun die Informationen über die Eigenschaften sowie die Kanten des Graphen aus. Die Ausgabe der Knoten-Informationen wird delegiert, indem auf jedem Knoten wiederum die Methode `printXMLOn:` aufgerufen wird. Nun gibt jeder Knoten seine Eigenschaften wie z. B. seinen Typen oder seine Position aus. Die Ausgabe der Informationen über das enthaltene `AAFDelegate`-Objekt wird durch den Aufruf von `printXMLOn:` an selbigem erneut delegiert. Handelt es sich dabei um einen Agenten, so schreibt dieser nun seinen Typen sowie seine Eigenschaften in die Datei. Handelt es sich um einen Graphen, so beginnt hier ein rekursiver Abstieg durch erneuten Aufruf der `printXMLOn:` auf einem `AAFGraph`-Objekt.

Natürlich wäre es auch möglich gewesen, einen Graphen auszugeben, indem es nur eine einzige Schreibmethode gibt, die die gesamte Ausgabe übernimmt. Dazu müsste diese jedoch Kenntnis vom Aufbau der Knoten- und Agenten-Klassen haben und wissen, wie sie auf deren Eigenschaften zugreifen kann. Zudem müsste sie stets erweitert oder angepasst werden, wenn neue Typen von Knoten oder Agenten hinzukommen oder sich an den bestehenden etwas ändert.

Daher wurde die Entscheidung getroffen, das Schreiben der Eigenschaften jeweils in die Verantwortung der Klasse zu legen, die ein bestimmtes Objekt implementiert. Auf diese Weise genügt es, wenn eine Klasse nur ihre eigenen Interna kennt. Zudem ist sie nur von ihren eigenen Änderungen betroffen, nicht jedoch von denen anderer Klassen.

Für Details zum XML-Speicherformat, in dem die Graphen gespeichert werden, sei auch hier wieder auf Abschnitt 4.6 verwiesen.

4.8 TRAITS

Sollen Klassen gemeinsames Verhalten haben, so kommt meist der Mechanismus der Vererbung zum Einsatz. Eine Basisklasse stellt Variablen und Methoden zur Verfügung und jede von ihr abgeleitete Klasse kann diese ebenso verwenden, als wären sie direkt in ihr implementiert worden.

Oft reicht dieses Vorgehen aus. Es gibt jedoch auch Fälle, in denen es wünschenswert ist, wenn Klassen Verhalten gemeinsam nutzen können, die nicht in einer Vererbungsbeziehung zueinander stehen. In manchen Programmiersprachen kann dieser Wunsch durch den Einsatz von sogenannten Schnittstellen (engl. Interfaces) realisiert werden, so z. B. in Java. Squeak kennt jedoch keine Schnittstellen.

```
1 Trait named: #GetMainPanel
2     uses: {}
3     category: 'AAFGT'
```

Listing 11: Anlegen eines `Traits`

Dennoch ist es durch die in Squeak-Version 3.9 eingeführten `Traits` möglich, dass Klassen, die keine Vererbungsbeziehung haben, Methoden miteinander teilen (Black, Ducasse, Nierstrasz und Pollet, 2009). Bei `Traits` handelt es sich um nichts weiter als Sammlungen von Methoden. Definiert werden solche Sammlungen so wie in Listing 11 dargestellt. Sollen die Methoden einer solchen Sammlung einer Klasse zur Verfügung stehen, so muss ihr der entsprechende `Trait` über die Angabe mit `uses:` bei der Klassendefinition bekannt gemacht werden. Ein Beispiel zeigt Listing 12. Fortan können die enthaltenen Methoden in der Klasse so verwendet werden, als hätte sie sie selbst implementiert oder geerbt. Es ist auch möglich, dass eine Klasse mehrere `Traits` verwendet oder ein `Trait` selbst auf bereits existierende `Traits` zurückgreift.

```
1 AAFGTBaseMorph subclass: #AAFGTBaseElement
2     uses: GetMainPanel + GetDrawArea + GetElement + SetDirty
3     instanceVariableNames: 'node dialog label labelMorph'
4     classVariableNames: ''
5     poolDictionaries: ''
6     category: 'AAFGT-Elements'
```

Listing 12: Verwenden eines `Traits`

Bei der Verwendung von `Traits` ist besonders darauf zu achten, dass die Klasse, der ein `Trait` zur Verfügung gestellt wird, alle in den `Trait`-Methoden verwendeten Aufrufe auch tatsächlich ausführen kann. Dies wird von Squeak nicht überprüft. Ist

METHODENNAME	FUNKTION
`getMainPanel`	Liefert eine Referenz auf das Haupt-Panel. Liefert `nil` für Morphe, die sich außerhalb des Haupt-Panels befinden.
`getDrawArea`	Liefert eine Referenz auf die Arbeitsfläche. Liefert `nil` für Morphe, die sich außerhalb der Arbeitsfläche befinden.
`getElement`	Liefert eine Referenz auf das Element, zu dem ein Morph gehört. Liefert `nil` für Morphe, die sich nicht in einem Element befinden.
`setDirty`	Markiert die Existenz noch nicht gespeicherter Änderungen am Graphen.

Tabelle 4: `Trait`-Methoden

eine Ausführung nicht möglich, so tritt der Fehler erst zur Laufzeit auf.

Trotz dieser Stolperfalle wurden `Traits` bei der Implementierung des Automatiken-GUI eingesetzt, um bestimmte Methoden zu realisieren, welche einige Klassen gemeinsam nutzen sollen, die nicht über Vererbung in Beziehung zueinander stehen. Die Alternative wäre gewesen, die Methoden mehrfach zu implementieren. Auf diese Weise wäre jedoch redundanter Code entstanden, Änderungen hätten jeweils an mehreren Stellen durchgeführt werden müssen. Das ist ineffizient und führt zu einer erhöhten Fehleranfälligkeit, weil leicht eine Stelle übersehen werden kann. Zudem ist die Anzahl der in den `Traits` verwendeten Methodenaufrufe überschaubar, sodass hier zugunsten der `Traits` entschieden wurde.

Die meisten der Methoden, die in `Traits` implementiert sind, sind Navigationsmethoden, mit denen ein Morph eine Referenz auf bestimmte andere Morphe innerhalb des GUI erhalten kann. Diese Methoden sind `getMainPanel`, `getDrawArea` und `getElement`, deren jeweilige Funktion in Tabelle 4 beschrieben ist.

Die Implementierung dieser drei Methoden ist sehr ähnlich und soll am Beispiel von `getMainPanel` kurz erläutert werden. Wird die Methode an einem Objekt aufgerufen, so prüft es zunächst, ob es selbst schon das gesuchte Haupt-Panel ist. Wenn dem so ist, wird eine Referenz auf das Objekt selbst zurückgegeben. Ist das Objekt nicht das gesuchte, so wird die `getMainPanel`-Methode rekursiv am übergeordneten Morph, dem `owner`, aufgerufen. Diese rekursiven Aufrufe wiederholen sich solange, bis das Haupt-Panel gefunden wurde oder aber ein Morph keinen `owner` hat. Gibt es keinen übergeordneten Morph mehr und

wurde das Haupt-Panel noch nicht gefunden, so liefert die Methode `nil` zurück und informiert so darüber, dass sich das ursprünglich rufende Objekt nicht innerhalb des Haupt-Panels befindet.

Die anderen Navigationsmethoden arbeiten analog und werden hier daher nicht gesondert beschrieben. Für Details sei auf den Quelltext verwiesen.

Neben den Navigationsmethoden ist noch eine weitere Methode in einem `Trait` implementiert, nämlich `setDirty`. Auch ihre Funktion ist in Tabelle 4 beschrieben.

4.9 UMSETZUNG DER DIALOGPRINZIPIEN UND GESTALTUNGSRICHTLINIEN

Im Folgenden soll kurz darauf eingegangen werden, inwieweit die in den Abschnitten 3.2 und 3.3 vorgestellten Dialogprinzipien und Gestaltungsrichtlinien bei der Erstellung des Automatiken-GUI berücksichtigt wurden. Hierfür werden zu den verschiedenen Prinzipien und Richtlinien jeweils die Punkte genannt, in denen sie umgesetzt wurden.

Dialogprinzipien

Für die Umsetzung der Aufgabenangemessenheit wurde versucht, den Funktionsumfang des GUI genau auf das notwendige Maß zu beschränken. Das bedeutet, dass die Software den Anwender vollständig dabei unterstützen soll, Automatiken für das SAM-System zusammenzustellen und zu konfigurieren. Gleichzeitig wurde jedoch darauf geachtet, keine Funktionen bereitzustellen, die über diese Aufgabe hinausgehen und somit störend, ablenkend oder gar belastend wirken könnten.

Die Selbstbeschreibungsfähigkeit wird zunächst durch Tooltipps, in Squeak `Balloon-Hints` genannt, unterstützt. Diese erscheinen als Textblase, sobald der Benutzer mit der Maus über einen Bereich fährt, und informieren ihn darüber, wofür der Bereich dient und welche Handlungsmöglichkeiten er dort hat. Des Weiteren wird dieses Prinzip dadurch umgesetzt, dass ein Großteil der Handlungsoptionen wie z. B. verfügbare Schaltflächen stets sichtbar ist. Bei Beschriftungen wurde darauf geachtet, dass diese bestmöglich die Funktion des zugehörigen Bedienelements beschreiben.

Alle zur Selbstbeschreibungsfähigkeit genannten Punkte dienen auch gleichzeitig der Lernförderlichkeit. Für dieses Prinzip kommt noch hinzu, dass die Modellbildung über die Funktionsweise einer Automatik für den Benutzer dadurch erleichtert wird, dass der zugrunde liegende AAF-Graph im GUI auf eine für Graphen typische Weise visuell repräsentiert wird.

Zur Steuerbarkeit ist zu sagen, dass das Automatiken-GUI von sich aus keinerlei Aktionen anstößt und der Benutzer somit vollständige Kontrolle über sämtliche Abläufe hat. Zudem kann er die Bedienschritte in weitestgehend freier Reihenfolge vornehmen. So muss ein Element zwar existieren, bevor es verbunden werden kann. Es bleibt jedoch dem Nutzer überlassen, in welcher Reihenfolge er die Kanten einfügt. Ebenso steht es ihm frei, ob er zunächst alle Elemente platziert und dann verbindet oder ob er nach dem Verbinden einiger noch weitere hinzufügt. Ebenso spielt es beim Bearbeiten von Eigenschaften eines Elements keine Rolle, welche Werte zuerst bearbeitet werden. Auch kann beliebig zwischen dem Einstellen von Werten und dem Hinzufügen oder Entfernen von Kanten und Elementen gewechselt werden.

Das Prinzip der Erwartungskonformität wird dadurch unterstützt, dass die Interaktion in gängiger Weise umgesetzt wurde. So kommen zum Platzieren, Verbinden oder Löschen von Elementen Bedientechniken zum Einsatz, die in gleicher Weise aus ähnlichen Programmen bekannt sind. Genannt seien hier Drag&Drop, Kontextmenüs bei Rechtsklick oder Auswahl durch Linksklick.

Ein Verstoß gegen das Prinzip der Erwartungskonformität, der an dieser Stelle genannt werden muss, ist das gleichzeitige Markieren mehrerer Elemente, indem sie bei gehaltener [ALT]-Taste mit der rechten Maustaste angeklickt werden. Die übliche Vorgehensweise wäre, sie bei gedrückter [STRG]-Taste mit der linken Maustaste anzuklicken. Da Squeak die Verwendung dieser Tastenkombination jedoch nicht zulässt, da es sie selbst abfängt und auf andere Art verarbeitet, wurde notgedrungen auf die genannte Kombination zurückgegriffen.

Zur Fehlertoleranz sei gesagt, dass, wo immer möglich, versucht wurde, Fehleingaben gar nicht erst zuzulassen. So können z. B. nur Verbindungen zwischen Elementen hergestellt werden, die aufgrund der zugrunde liegenden Graphen-Definition im AAF zulässig sind. Bei der Eingabe von Werten für Eigenschaften wird das benötigte Datenformat meist bereits durch das verwendete Bedienelement erzwungen. Auch findet eine Wertüberprüfung statt, wann immer diese realisierbar ist.[30] Sind Werte voneinander abhängig, wurden auch die Bedienelemente so gekoppelt, dass sich bei Änderungen an einem der Werte die anderen in der Weise anpassen, dass die Gesamtheit der Angaben gültig bleibt. Ein Beispiel hierfür ist die Konfiguration der Steuergewalt der MWB, die in der Summe immer 100 % betragen muss.

30 Da zum Zeitpunkt der Erstellung des Automatiken-GUI Agenten nur zu Zwecken der Demonstration vorlagen und die zulässigen Wertebereiche für deren Eigenschaften nicht immer klar waren, konnte an einigen Stellen keine Überprüfung vorgenommen werden.

Fehler, die bei der Benutzung des GUI auftreten können, sind der Versuch, beim Speichern einer Automatik eine andere zu überschreiben oder eine noch nicht gespeicherte Automatik versehentlich zu verwerfen. In diesen Fällen wird der Nutzer vom System gewarnt und ihm wird die Möglichkeit gegeben, den Fehler zu beheben.

Das siebte Prinzip, die Individualisierbarkeit, wurde in der jetzigen Form des Automatiken-GUI nicht berücksichtigt. Der Grund hierfür ist, dass es bei dem klar umrissenen und nicht übermäßig großen Funktionsumfang sowie der zahlenmäßig derzeit stark beschränkten Benutzergruppe nicht notwendig erschien, sondern sinnvoller war, die zur Verfügung stehenden Ressourcen in wichtigere Anforderungen zu investieren. Dennoch gibt es ein paar Ideen, wie das Automatiken-GUI individualisiert werden könnte. Diese finden in Abschnitt 5.2 im Abschlusskapitel dieser Arbeit kurz Erwähnung.

Gestaltungsrichtlinien

Die visuelle Codierung des Graphen bedient sich der auch anderweitig häufig verwendeten Darstellung als durch Kanten verbundene Knoten. Die Knoten werden durch Rechtecke symbolisiert, die Kanten durch Linien abgebildet. Die farbliche Gestaltung wurde bewusst auf wenige Farben beschränkt. Neben Schwarz für Schrift und Linien und Weiß für den Hintergrund kommen nur noch einige Abstufungen eines Grüntons sowie an wenigen Stellen Rot zum Einsatz. Es wurde versucht, durchgängig positive Farbdarstellungen zu wählen, also dunkle Schrift auf hellem Grund zu verwenden. Dies folgt der Empfehlung in BGI 650 (2007). Bei der Anordnung der Elemente wurde versucht, solche zu gruppieren, die sich ähnlich sind. So wurden normale und komplexe Elemente in jeweils eigenen Auswahlbereichen zusammengefasst. Die Schaltflächen zur Bedienung des GUI wurden ebenfalls in einem eigenen Bereich angeordnet. Mit Blick auf die Intuitivität wurde die Bedienung des Automatiken-GUI so weit wie möglich auf direkte Manipulation ausgelegt.

4.10 INTEGRATION DER AUTOMATIKEN IN SAM

Soll eine Automatik in einem Versuch eingesetzt werden, so muss sie zunächst gemäß der Bedienungsanleitung in Anhang G erstellt und abgespeichert werden. Für diesen Schritt kommt das in dieser Arbeit beschriebene GUI zum Einsatz.

Liegt die Automatik nun als XML-Datei vor, so muss sie dem SAM-System bekannt gemacht werden. Zu diesem Zweck wird

auf die Datei *steps.txt*[31] zurückgegriffen, in der die vollständige Konfiguration einer Untersuchung zu finden ist. Diese Datei enthält mehrere Zeilen und Spalten, wobei jede Zeile eine Strecke der Untersuchung repräsentiert und in den Spalten die verschiedenen Angaben zum Aufbau der Strecke, zu Hindernissen, zum aktiven MWB usw. enthalten sind. Für eine genauere Beschreibung der Konfiguration sei hier auf die Dokumentation des SAM-Systems verwiesen, welche bei Bothe et al. (2009) zu finden ist.

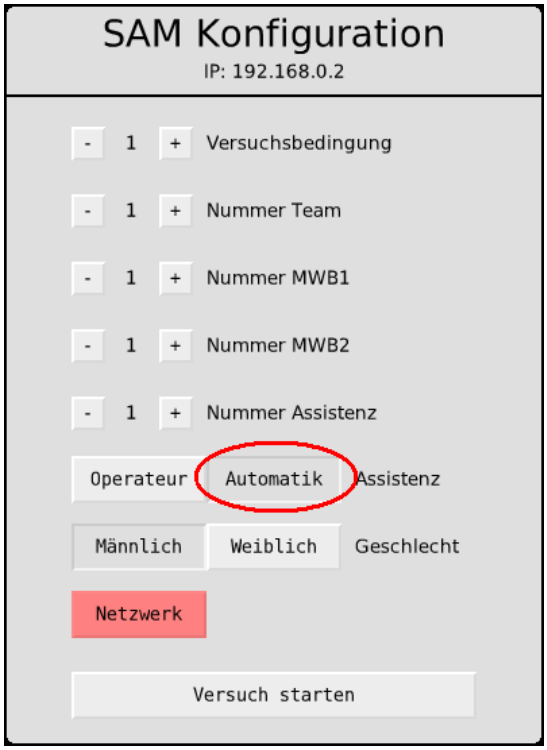


Abbildung 25: Versuchsleiter-Arbeitsplatz

Um nun die Information über die zu verwendende Automatik in die Datei *steps.txt* zu integrieren, wurde das Format um eine weitere Spalte ergänzt. In dieser wird pro Zeile, also pro untersuchter Strecke, der Dateiname der zu verwendenden Automatik angegeben. Hierbei ist zu beachten, dass die Datei mit der Automatik im Verzeichnis *automations* unterhalb des Verzeichnisses zu finden sein muss, in dem sich das Squeak-Image mit dem SAM-System befindet. Wird für eine Untersuchungsstrecke keine Automatik angegeben, so wird für diese die leere Automatik verwendet, die nur aus Wurzel- und Senkenknoten besteht und keinerlei Aktionen ausführt.

Sind nun in der Datei *steps.txt* alle gewünschten Automatiken eingetragen, so kann die Untersuchung gestartet werden. Wird

31 Die Datei befindet sich im Verzeichnis *trackConfig* unterhalb des Verzeichnisses, in dem das Squeak-Image mit dem SAM-System liegt.

die Automatik-Unterstützung im Versuchsleiter-Arbeitsplatz aktiviert (s. Abbildung 25), so liest das SAM-System die Automatiken ein und verwendet sie jeweils im angegebenen Schritt der Untersuchung.[32]

4.11 TESTS

Da zum Entwickeln einer Anwendung immer auch Tests gehören müssen, um die korrekte Arbeitsweise sicherzustellen, soll diesem Aspekt ein eigener Abschnitt gewidmet werden. Es ist jedoch unmöglich, das Thema Tests im Rahmen dieser Arbeit detailliert abzudecken. Die große Breite dieses Feldes zeigt sich eindrucksvoll z. B. bei Rätzmann (2003). So kann die Herangehensweise eher explorativ sein mit Testfällen, die sich während des Testprozesses ergeben. Sie kann aber auch stark formalisiert sein und von einem festen Katalog zuvor festgesetzter Testfälle ausgehen. Der Testprozess kann in den Entwicklungsprozess integriert sein, er kann jedoch auch davon losgelöst ablaufen. Die Tester können u. U. die Entwickler selbst sein, aber auch andere Mitglieder eines Projekt-Teams oder unabhängige Personen können diese Rolle übernehmen. Tests können manuell oder automatisiert ausgeführt werden. Zudem kann das Testen völlig verschiedene Aspekte eines Programms betreffen, wie z. B. das Vorhandensein bestimmter Funktionen, das Look&Feel einer Anwendung, die Performanz oder das Systemverhalten unter Last.

Im Folgenden soll besonders auf eine bestimmte Art der automatisierten Tests, das sogenannte Unit-Testing, sowie sein Einsatz bei der Entwicklung des Automatiken-GUI eingegangen werden. Darüber hinaus soll beleuchtet werden, inwieweit sich diese Art des Testens für die Überprüfung eines GUI eignet.

Beim Unit-Testing handelt es sich um ein Vorgehen, bei dem einzelne Programmteile automatisiert auf korrektes Verhalten getestet werden. Die Automatisierung erfolgt über die Erstellung von Programmcode, welcher die fragliche Aktion nicht nur ausführt, sondern auch Vor- und Nachbedingungen prüft und somit über Erfolg oder Misserfolg entscheiden kann. Im Idealfall deckt die Gesamtheit aller Test-Skripte die Funktionalität eines Programms vollständig ab, wobei Rätzmann (2003) explizit darauf hinweist, dass in der Realität nie alles getestet werden kann.

Die Wahl gerade dieser Test-Methode ist vor allem dadurch begründet, dass Squeak mit `SUnit` ein Werkzeug bereitstellt, welches die Erstellung und Durchführung von Unit-Tests unterstützt. Ein weiterer Vorteil ist, dass keine weiteren Testpersonen benötigt werden, da die Testfälle vom Entwickler selbst erstellt

32 Der hierfür zuständige Code befindet sich in der Methode `configureStep:` der Klasse `SAMControllerExperiment`.

werden. Zudem können die Tests während der laufenden Entwicklung immer wieder als sogenannte Regressionstests (Rätzmann, 2003) ausgeführt werden, sodass Fehler, die nach Programmänderungen an zuvor korrekten Programmteilen auftreten, sofort bemerkt und behoben werden können. Ein Nachteil ist jedoch, dass mit `SUnit` keine Tests der GUI-Interaktion möglich sind. Daher wurde es nur zum Testen von Programmkomponenten eingesetzt, welche keine Nutzerinteraktion benötigen.

Für Tests der GUI-Interaktion wurde versucht, `SUnit` mit einem von Squeak bereitgestellten Ereignis-Rekorder zu kombinieren. Dieser erlaubt die Aufzeichnung von Tastatureingaben und Mausaktionen, die sich anschließend in Dateien abspeichern und später erneut wiedergeben lassen. Eine sehr starke Einschränkung hierbei ist jedoch, dass sich die Wiedergabe der aufgezeichneten Ereignisse nicht aus dem Testfall heraus steuern lässt. Sobald sie gestartet wurde, läuft sie komplett durch. Daher lassen sich Werte eines Graphen oder der Zustand des GUI erst am Ende der Wiedergabe überprüfen und nicht nach jedem beliebigen Schritt, z. B. direkt nach einem bestimmten Mausklick. Auch die Idee, die Aufzeichnung in mehrere Dateien zu zerlegen, und diese nacheinander abzuspielen, um zwischen zwei Wiedergaben Wertprüfungen durchführen zu können, führte zu keinem Erfolg. Der Grund hierfür ist, dass der Versuch mehrerer Wiedergaben nacheinander zu Laufzeitfehlern des Ereignis-Rekorders führte.

Ausgeklügelte Test-Frameworks für grafische Oberflächen, die die vollständig automatisierte Steuerung des GUI erlauben und dabei an beliebiger Stelle Wertüberprüfungen zulassen, stehen für Squeak leider nicht zur Verfügung.[33]

Die Konsequenz hieraus ist, dass sich das GUI für Testzwecke nicht in praktikabler Weise automatisiert bedienen lässt. Daher kann die Interaktion nur manuell getestet werden. Hierfür wurde eine Prüfliste erstellt, die nach größeren Änderungen am GUI abzuarbeiten und ggf. zu erweitern ist. Es sei allerdings darauf hingewiesen, dass es nicht möglich ist, dabei jede erdenkliche Eingabe zu berücksichtigen. Durch die große Fülle möglicher Testfälle wäre ein solcher Test durch seinen zeitlichen Aufwand ohne Automatisierung kaum leistbar; erst recht nicht, wenn er wiederholt durchgeführt werden muss. Deshalb beschränkt sich die Prüfliste auf eine Reihe von stichprobenartigen Testfällen, wobei versucht wurde, durch die Auswahl einen möglichst großen Teil der Funktionalität abzudecken.

33 Für andere Programmiersprachen wie z. B. Java sind solche Werkzeuge zahlreich vorhanden. Die unvollständige Liste unter http://en.wikipedia.org/wiki/List_of_GUI_testing_tools (Version vom 4.8.2010, 16:56 Uhr) vermittelt bereits einen Eindruck davon.

Die Prüfliste für manuelle Tests findet sich ebenso wie die `SUnit`-Tests in Anhang E. Im Folgenden soll für jedes der beiden Verfahren ein Testfall exemplarisch vorgestellt werden.

Manueller Test

Für die manuellen Testfälle wurden jeweils Vor- und Nachbedingungen beschrieben, die vor bzw. nach der Testausführung erfüllt sein müssen. Außerdem wurde beschrieben, was der manuelle Tester ausführen soll. Zur Auswertung der Vor- und Nachbedingungen kommt neben dem Code-Inspector[34] auch Programmcode zum Einsatz, welcher im Squeak-Workspace[35] ausgeführt wird. Stimmt das Ergebnis nicht mit den angegebenen Nachbedingungen überein, wird der Quelltext überarbeitet, bis das erwartete Verhalten vorliegt.

Im hier vorgestellten Testfall (Tabelle 5) wird in einer leeren Arbeitsfläche ein kleiner Graph zusammengestellt. Anschließend werden sowohl seine grafische Repräsentation als auch seine `AAF`-Repräsentation anhand der Nachbedingungen auf Korrektheit überprüft.

Unit-Test

Der in Listing 13 abgebildetete Testfall testet die Klassenmethode `convert:type:` der Klasse `AAFUtils`. Hierzu werden während des Tests einige Konvertierungen vorgenommen. Die Ergebnisse werden anschließend daraufhin überprüft, ob ihr Typ und ihr Wert korrekt sind. Durch die `should:-` bzw. `shouldnt:-` Anweisungen werden die erwarteten Ergebnisse festgelegt. Der Squeak-Testrunner[36] wertet aus, ob diese tatsächlich vorliegen und gibt dem Programmierer Rückmeldung über Erfolg oder Misserfolg des Tests.

34 Werkzeug der Squeak-Entwicklungsumgebung zum Inspizieren und Manipulieren von Objekten.

35 Werkzeug der Squeak-Programmierumgebung, das das direkte Ausführen von Codezeilen erlaubt.

36 Werkzeug der Squeak-Entwicklungsumgebung, welches Unit-Tests ausführt.

TESTFALL 1	
Vorbedingung:	• Es befinden sich außer Wurzel- und Senkenelement keine Elemente im Arbeitsbereich. • Der zugehörige AAF-Graph besteht nur aus Wurzel- und Senkenknoten. • Wurzel- und Senkenknoten haben keine Verbindungen.
Test:	• Ein normales Element wird von links aus dem Elementbereich in die Arbeitsfläche gezogen. • Das Element wird mit Wurzel- und Senkenelement verbunden.
Nachbedingung:	• Das Element befindet sich im Arbeitsbereich. • Das Element ist mit Wurzel- und Senkenelement verbunden. • Der zuhörige AAF-Graph besteht aus Wurzel- und Senkenknoten und einem normalen Knoten. • Der Typ des Agenten im normalen Knoten des AAF-Graphen passt zum Element. • Im AAF-Graphen hat der Wurzelknoten den normalen Knoten als Kindknoten. • Im AAF-Graphen hat der Senkenknoten den normalen Knoten als Elternknoten. • Im AAF-Graphen hat der normale Knoten den Wurzelknoten als Elternknoten und den Senkenknoten als Kindknoten.

Tabelle 5: Beispiel für einen manuellen Testfall

```
testConvert
    | temp |

    temp := AAFUtils convert: 'Hallo' type: 'String'.
    self should: [temp isKindOf: String].
    self should: [temp = 'Hallo'].

    temp := AAFUtils convert: '123' type: 'String'.
    self should: [temp isKindOf: String].
    self should: [temp = '123'].
    self shouldnt: [temp = 123].

    temp := AAFUtils convert: '123' type: 'Number'.
    self should: [temp isKindOf: Number].
    self should: [temp isKindOf: Integer].
    self should: [temp = 123].
    self shouldnt: [temp isKindOf: Float].
    self shouldnt: [temp = '123'].

    temp := AAFUtils convert: '-45.0' type: 'Number'.
    self should: [temp isKindOf: Number].
    self should: [temp isKindOf: Float].
    self should: [temp = -45.0].
    self shouldnt: [temp isKindOf: Integer].
    self shouldnt: [temp = '-45.0'].

    temp := AAFUtils convert: 'true' type: 'Boolean'.
    self should: [temp isKindOf: Boolean].
    self should: [temp = true].
    self shouldnt: [temp = 'true'].

    temp := AAFUtils convert: 'false' type: 'Boolean'.
    self should: [temp isKindOf: Boolean].
    self should: [temp = false].
    self shouldnt: [temp = 'false'].

    temp := AAFUtils convert: 'Banane' type: 'Boolean'.
    self should: [temp isKindOf: String].
    self should: [temp ='Banane'].
    self shouldnt: [temp isKindOf: Boolean].
    self shouldnt: [temp = true].
    self shouldnt: [temp = false].
```

Listing 13: Testen der Methode `convert:type:` durch einen `SUnit`-Test

5 ZUSAMMENFASSUNG UND AUSBLICK

5.1 ZUSAMMENFASSUNG

Ziel der vorliegenden Arbeit war es, einen technisch wenig versierten Nutzer dabei zu unterstützen, Automatiken für das im Projekt ATEO verwendete SAM-System zu erstellen. Hierfür wurde als Ergebnis der Arbeit ein GUI entworfen und implementiert.

Im Rahmen der Arbeit wurden zunächst die für die Umsetzung notwendigen technischen Mittel beleuchtet. Durch das Projekt vorgegeben waren die Verwendung von Squeak, Morphic und SAM, für die Speicherung der anfallenden Daten fiel die Wahl auf XML.

Anschließend wurde untersucht, wie ein solches GUI software-ergonomisch bestmöglich umgesetzt werden kann. Dazu wurden sowohl allgemeine Dialogprinzipien als auch konkrete Gestaltungshinweise gesammelt und ausgewählt. Ferner wurde erörtert, wie sich überprüfen lässt, ob software-ergonomische Ziele erreicht wurden, und wie die in der Informatik gängigen Vorgehensmodelle mit der Entwicklung nutzerfreundlicher Software in Einklang stehen.

Zuletzt wurde auf Basis der erlangten Erkenntnisse das GUI konzipiert, in Squeak realisiert und getestet. Hierbei kamen gängige Softwaretechniken zum Einsatz, so z. B. die Analyse von Anwendungsfällen und die Erstellung eines objektorientierten Analyse- bzw. Designmodells. Die Tests wurden teilweise automatisiert und teilweise manuell durchgeführt.

5.2 PROBLEME, IDEEN UND AUSBLICK

Während der Bearbeitung des in dieser Arbeit beschriebenen GUI-Projekts sind immer mehr Ideen entstanden, zum Teil für zusätzliche Funktionalität, zum Teil für alternative Realisierungen bestehender Funktionen. Leider konnten aufgrund der begrenzten Zeit viele dieser Ideen nicht mehr umgesetzt werden. Sie sollen im Folgenden beschrieben werden, damit sie nicht verloren gehen und möglicherweise in Nachfolgeprojekten aufgegriffen werden können. Außerdem soll auf Probleme eingegangen werden, die mit der aktuellen Implementierung noch bestehen und in zukünftigen Versionen behoben werden sollten.

5.2.1 *Fortführung der Automatiken-Entwicklung*

Die Arbeit an der Automatiken-Entwicklung wird fortgeführt. Vier neue Projektmitglieder implementieren derzeit konkrete Elementar-Automatiken. Hierbei kommt das von Michael Hasselmann (in Bearb.) entworfene AAF-Framework zum Einsatz. Im Einzelnen beschäftigt sich Nikolai Kosjar mit der Umsetzung sogenannter weicher Eingriffe. Hierunter werden visuelle und auditive Hinweise verstanden, die den MWB durch Automatiken gegeben werden. Die Arbeit von Aydan Seid wird den MWB Soll- und Ist-Anzeigen für Richtungen, Geschwindigkeiten und Joystick-Auslenkungen bereitstellen. Auch diese werden zu den weichen Eingriffen gezählt. Andreas Wickert implementiert die harten Eingriffe. Dies sind solche, die die Automatik ausführt, ohne dass die MWB es verhindern können. Beispiele sind die Rückführung zur Strecke bei Abweichung oder das Abbremsen des Objekts. Helmut Weidner-Kim beschäftigt sich damit, Hilfsfunktionen bereitzustellen, die die Elementar-Automatiken für ihre Aufgabe benötigen. Als ein wichtiges Beispiel sei die Bestimmung der Ideallinie genannt, auf der sich das Objekt im besten Fall befinden sollte.

Die entwickelten Elementar-Automatiken sollen dann unter Verwendung des in dieser Arbeit entstandenen GUI zu komplexeren Automatiken kombiniert und im SAM-System getestet werden.

5.2.2 *Anpassen von Einstellungen des Automatiken-GUI*

Derzeit werden Eigenschaften des GUI wie Größen, Positionen, Farben etc. verwaltet, indem ihre Werte von Klassenmethoden der Klasse `AAFGTConsts` zurückgegeben werden.[37] Änderungen der Erscheinungsweise verlangen also immer einen Eingriff in den Quellcode und damit gewisse Squeak- und Smalltalk-Kenntnisse.

Es gibt durchaus Einstellungen, die der Anwender im Sinne der Software-Ergonomie nicht verändern können sollte.[38] Andere hingegen sollte er völlig frei oder wenigstens in gewissem Maße an seine Vorlieben und Bedürfnisse anpassen können, so z. B. Schriftgrößen oder Standardspeicherorte. Diese dürfen daher nicht im Quellcode festgeschrieben sein, sondern müssen z. B. in Konfigurationsdateien abgelegt werden, aus denen das Programm sie ausliest. Auch sollte zum Ändern einer Einstellung kein direktes Editieren von Konfigurationsdateien nötig

37 Squeak kennt keine Konstanten, dieses Vorgehen kommt einer Konstante sehr nahe und erlaubt die zentralisierte Verwaltung von Eigenschaftswerten.

38 Es besteht z. B. keine Notwendigkeit dem Anwender das Setzen sehr ungünstiger Farbkombinationen zu ermöglichen.

sein, sondern das Automatiken-GUI sollte ein Optionen-Menü bieten, über welches die Werte änderbar sind. Vom Anwender vorgenommene Anpassungen müssen den alten Wert in der Konfigurationsdatei überschreiben, sodass das GUI beim nächsten Start in der individuell veränderten Weise erscheint.

5.2.3 *Lokalisierung*

Das Automatiken-GUI enthält zahlreiche sprachabhängige Beschriftungen, so z. B. Aufschriften auf Schaltflächen oder Hinweistexte in den sogenannten Balloon-Hints. Da eine öffentliche Bereitstellung der im Projekt ATEO entstandenen Software angedacht ist, sollte das GUI nicht an eine bestimmte Sprache gebunden sein. Vielmehr sollte die verwendete Sprache eingestellt werden können. Hierzu ist es notwendig, die Auswahl der konkreten Zeichenketten von einer zentralen Spracheinstellung abhängig zu machen.

Ein erster Schritt in diese Richtung wurde bereits gemacht. Das Automatiken-GUI enthält die aktuelle Sprachwahl in einer Variablen und abhängig von dieser liefern die entsprechenden Klassenmethoden von `AAFGTConsts` einen Text in der jeweiligen Sprache zurück. Allerdings ist zum Wechsel der Sprache ein Eingriff in den Programmcode notwendig. Dies erfordert wieder wenigstens grundlegende Programmierkenntnisse in Squeak. Diese Einstellung sollte in zukünftigen Versionen über eine Konfigurationsdatei oder ein Optionen-Menü innerhalb der Anwendung vorgenommen werden können.

Auch ist das Hinzufügen weiterer Sprachen derzeit nicht optimal gelöst. Es müssen sehr viele verschiedene Methoden anpasst werden, eine für jeden anzupassenden Text. Dieses Vorgehen birgt die Gefahr, einzelne Änderungen zu vergessen, was wiederum zu sprachlichen Inkonsistenzen im GUI führt, da beim Fehlen einer bestimmten Sprachversion eine Standardsprache (derzeit Englisch) verwendet wird. Für die leichtere Wart- und Erweiterbarkeit wäre es sinnvoller, die verschiedenen Zeichenketten jeder Sprachversion an einem gemeinsamen Ort abzulegen. Eine Möglichkeit wäre die Verwaltung in Lokalisierungsdateien.

5.2.4 *Individuelle Element-Paletten*

Derzeit stellt das Automatiken-GUI zwei Auswahlbereiche für Elemente zur Verfügung. Es ist durch den Programmcode festgelegt, welche Elemente in welchem dieser Bereiche verfügbar sind. Um ein leichteres Arbeiten mit dem GUI zu ermöglichen, wäre es denkbar, das Anlegen eigener Element-Paletten zuzulassen. Auf einer solchen Palette könnte der Nutzer die Elemente

nach eigenen Kriterien gruppieren, z. B. nach der Häufigkeit der Verwendung oder nach bestimmten Funktionen.

Diese individualisierten Paletten müssten vom Programm in einem geeigneten Format gespeichert werden, sodass sie bei der nächsten Programm-Verwendung wieder zur Verfügung stehen.

5.2.5 *Generische Dialoge*

In der jetzigen Form des Automatiken-GUI muss für jeden Agenten, der Eigenschaften über die von `AAFAgent` ererbten hinaus besitzt, ein Dialog implementiert werden, über welchen die Eigenschaftswerte betrachtet und editiert werden können. Dies ist dem Umstand geschuldet, dass sich das ursprünglich geplante generische Erstellen von Eigenschaften-Dialogen als zu komplex erwiesen hat, um im Rahmen dieser Arbeit implementiert werden zu können. Es wäre wünschenswert, dies bei einer Weiterentwicklung des GUI nachzuholen.

Eine Implementierung, welche die Eigenschaften-Dialoge generisch zu einem Agenten erstellt, müsste die zu verwendenden Bedienelemente aus den Datentypen der Agent-Eigenschaften ableiten. Da die Agenten Objekte des AAF sind, müssten wahrscheinlich auch in diesem Anpassungen vorgenommen werden. Es wäre eine Schnittstelle zu entwickeln, über die das Automatiken-GUI die erforderlichen Informationen abfragen könnte.

Eine weitere Möglichkeit wäre, in gewissem Maße eine Konfiguration der generisch erstellten Dialoge zuzulassen. So könnte es je nach Anwendungsfall und Wertebereich durchaus sinnvoll sein, eine numerische Eigenschaft in einem Fall als Schieberegler und in einem anderen Fall als Spin-Button darzustellen.

Es wäre zu diskutieren, ob diese Konfiguration im AAF oder im Automatiken-GUI vorgenommen werden sollte oder vielleicht als eigene Schicht dazwischen umgesetzt werden sollte.

5.2.6 *Gemeinsames Bearbeiten gleichartiger Elemente*

Das Automatiken-GUI lässt in seiner jetzigen Form das gleichzeitige Aktivieren bzw. Deaktivieren von Agenten für bestimmte Streckenabschnittstypen zu. Alle weiteren Einstellungen müssen für jeden Agenten separat vorgenommen werden.

Hier sollte in zukünftigen Programmversionen das gleichzeitige Bearbeiten von Eigenschaften ermöglicht werden, wenn alle markierten Agenten vom gleichen Typ sind, also die gleichen Eigenschaften und einen identischen Dialog besitzen. Sind nur Agenten ausgewählt, welche in einer Vererbungsbeziehung zueinander stehen, so sollten die gemeinsamen Eigenschaften zugleich bearbeitet werden können.

Weiter wäre darüber nachzudenken, ob und wie eine Erkennung gleichartiger Eigenschaften bei Agenten gänzlich verschiedener Typen möglich ist, um auch für diese ggf. ein gemeinsames Editieren zu ermöglichen.

5.2.7 *Kombination zum Markieren mehrerer Elemente*

Die derzeitige Kombination zum Markieren mehrerer Elemente, nämlich der Rechtsklick bei gedrückter [Alt]-Taste, ist nicht erwartungskonform. Leider verhindert Squeak die Verwendung der gängigen Vorgehensweise durch Linksklick bei gedrückter [Strg]-Taste, indem es diese Kombination abfängt und auf undurchsichtige Weise selbst verarbeitet.

Hier wäre für zukünftige Versionen zu untersuchen, wie dieses Abfangen genau funktioniert und ob es umgangen werden kann. Allerdings wäre zu diskutieren, ob es überhaupt umgangen werden sollte. Dadurch könnte zwar im Automatiken-GUI die übliche Kombination verwendet werden, aber innerhalb von Squeak käme es zu einer inkonsistenten Reaktion auf diese Eingabe.[39]

5.2.8 *Zurücknehmen von Änderungen*

Aufgrund der begrenzten Zeit wurde bislang darauf verzichtet, eine Funktion zum Zurücknehmen von Änderungen zu implementieren. Eine solche sollte in Zukunft hinzugefügt werden.

Eine Möglichkeit zur Umsetzung wäre das Erstellen einer Kopie des Graphen auf einem Stack vor jeder Änderung. Wünscht der Nutzer, dass ein Schritt rückgängig gemacht wird, so wird der jeweils oberste Graph vom Stack geholt und als aktueller gesetzt. Auf ähnliche Weise könnte mit einem weiteren Stack auch eine Wiederherstellen-Funktion realisiert werden, wie sie aus vielen Programmen bekannt ist. Allerdings wäre bei aller Einfachheit dieser Methode zu prüfen, ob sie zu speicherintensiv für eine Verwendung in Squeak ist.[40]

Eine alternative Möglichkeit wäre eine geeignete Protokollierung der Änderungen, aus der diese rückgängig gemacht werden können. Dieser Weg wäre wahrscheinlich schwieriger in der Umsetzung, dafür aber weniger speicherintensiv.

---

39 In Squeak wird die Kombination verwendet, um auf dem angeklickten Element ein Menü zu öffnen. Dieses Verhalten würde nun bei einem Klick innerhalb des Automatiken-GUI nicht mehr gezeigt.

40 Auch wenn es sich dabei jeweils um wenig Speicher handelt, hat die Erfahrung gezeigt, dass Squeak für solche Probleme anfälliger ist als andere Programmiersprachen.

5.2.9 *Einheitliche grafische Bedienelemente*

Squeak bietet einige grafische Bedienelemente wie Schaltflächen oder Menüs an, die verwendet werden können. Allerdings sind diese sowohl im Look&Feel als auch in der Benutzung im Quellcode alles andere als gleichartig. Viele gängige Bedienelemente sind überhaupt nicht vorhanden, sodass diese bei Bedarf erst erstellt werden müssen. Ein Beispiel hierfür sind Radio-Buttons.

Für die zukünftige Weiterentwicklung könnte es daher sinnvoll sein, als Unterprojekt einen Fundus von Bedienelementen zu schaffen, welcher innerhalb des GUI leicht verwendet werden kann. Alternativ könnte geprüft werden, inwieweit die Integration bereits vorhandener GUI-Frameworks über Projekte wie wxSqueak[41] oder SqueakGtk[42] möglich ist.

5.2.10 *Ausbau des Dialogs für visuelle Hinweise*

Derzeit ist noch kein Freitext für Hinweise möglich, sondern es werden Bilder als Hinweise eingeblendet. Daher und aufgrund des eingeschränkten Zeitrahmens beschränkt sich die in dieser Arbeit vorgestellte Umsetzung eines Dialogs auf die Platzierung dieser Hinweisbilder in der Streckenvorschau. Zukünftige Versionen sollten dahin gehend ausgebaut werden, dass frei formulierte Hinweise möglich sind. Für diese müssen sich dann auch weitere Eigenschaften wie Schrifttyp, -größe und -farbe einstellen lassen.

5.2.11 *Integration in den Versuchsleiter-Arbeitsplatz*

Derzeit werden die Automatiken in einer vom SAM-System prinzipiell unabhängigen Anwendung erstellt, abgespeichert und dann über eine Konfigurationsdatei in SAM integriert. Bei zukünftigen Entwicklungen sollte darüber nachgedacht werden, wie das Automatiken-GUI besser in das SAM-System integriert werden kann. Möglichkeiten wären der Aufruf des GUI vom Versuchsleiter-Arbeitsplatz aus sowie das Aktivieren einer Automatik für eine Untersuchung direkt im GUI.

5.2.12 *Tastatursteuerung*

Abgesehen von wenigen Ausnahmen bei Texteingaben ist das Automatiken-GUI derzeit ausschließlich auf Mausbedienung ausgelegt. Zukünftige Versionen sollten parallel die vollständige Tastatursteuerung als alternative Eingabeform ermöglichen.

41 http://www.wxsqueak.org
42 http://squeakgtk.pbworks.com

Die Möglichkeit zur Festlegung individueller Tastenkürzel wäre ebenfalls wünschenswert.

5.2.13 *Hilfen zur Bedienung*

Es liegt bereits eine kurze bebilderte Bedienungsanleitung zum Automatiken-GUI vor. Diese ist in dieser Arbeit in Anhang G zu finden. Auch die in die Programmoberfläche integrierten Balloon-Hints, die erscheinen, wenn mit der Maus über bestimmte Bereiche gefahren wird, bieten bereits einige Hinweise zur Benutzung.

Dennoch wäre es schön, in Zukunft darüber hinausgehende Hilfen bereitzustellen. Anbieten würde sich z. B. eine umfangreichere Online-Hilfe, in der der Benutzer nach bestimmten Stichwörtern oder auch Aufgaben suchen kann. Durch aufgabenorientierte Schritt-für-Schritt-Anleitungen oder Video-Tutorien könnte das Erlernen der Programmbenutzung erleichtert werden.

5.2.14 *Usability-Evaluation*

Leider war es durch den zeitlich begrenzten Rahmen dieser Arbeit nicht möglich, eine Usability-Evaluation nach einer der in Abschnitt 3.4 erläuterten Verfahren durchzuführen. Dies sollte, sofern möglich, nachgeholt werden, um bisher nicht erkannte Schwachstellen und Verbesserungsmöglichkeiten aufzuzeigen.

A

KLASSENDIAGRAMM: AAF

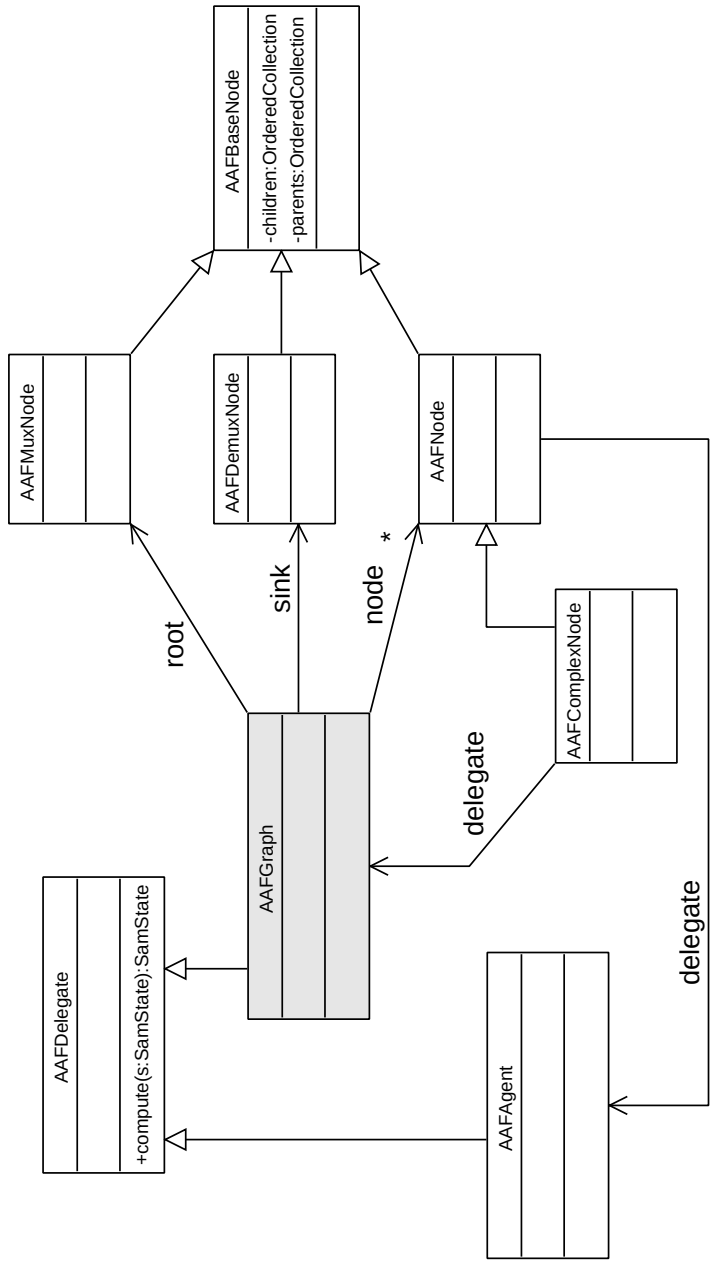


Abbildung 26: Klassendiagramm: AAF

B

KLASSENDIAGRAMME: AUTOMATIKEN-GUI

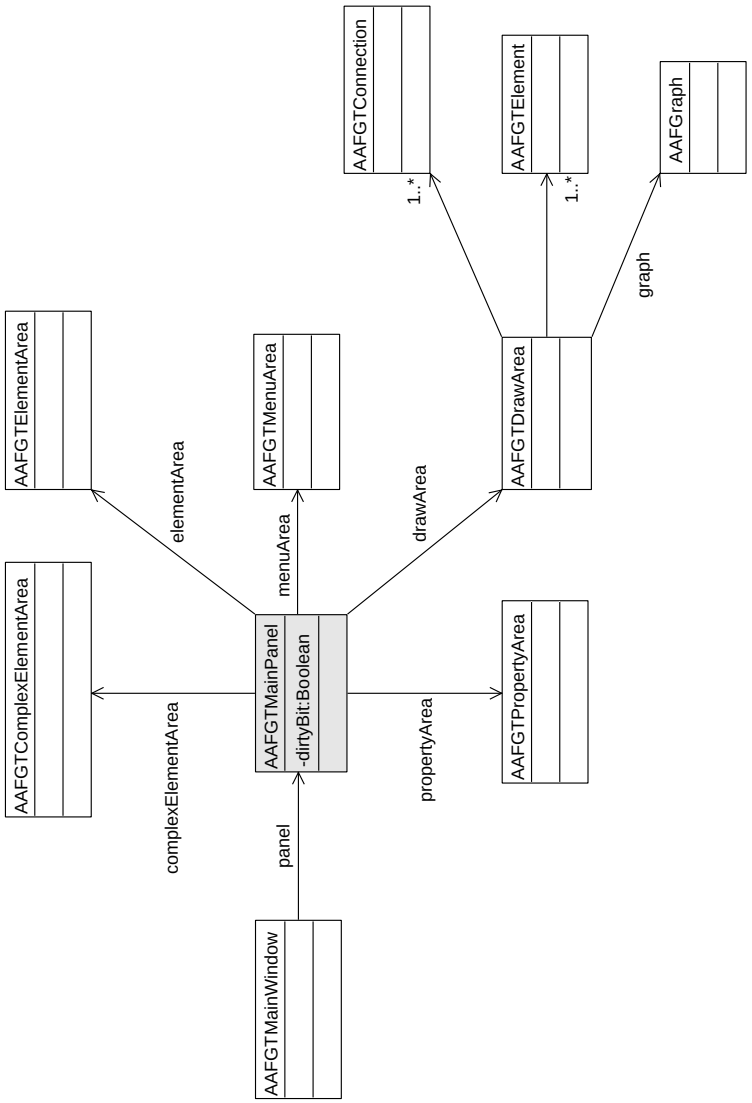


Abbildung 27: Hauptfenster: `AAFGTMainWindow`

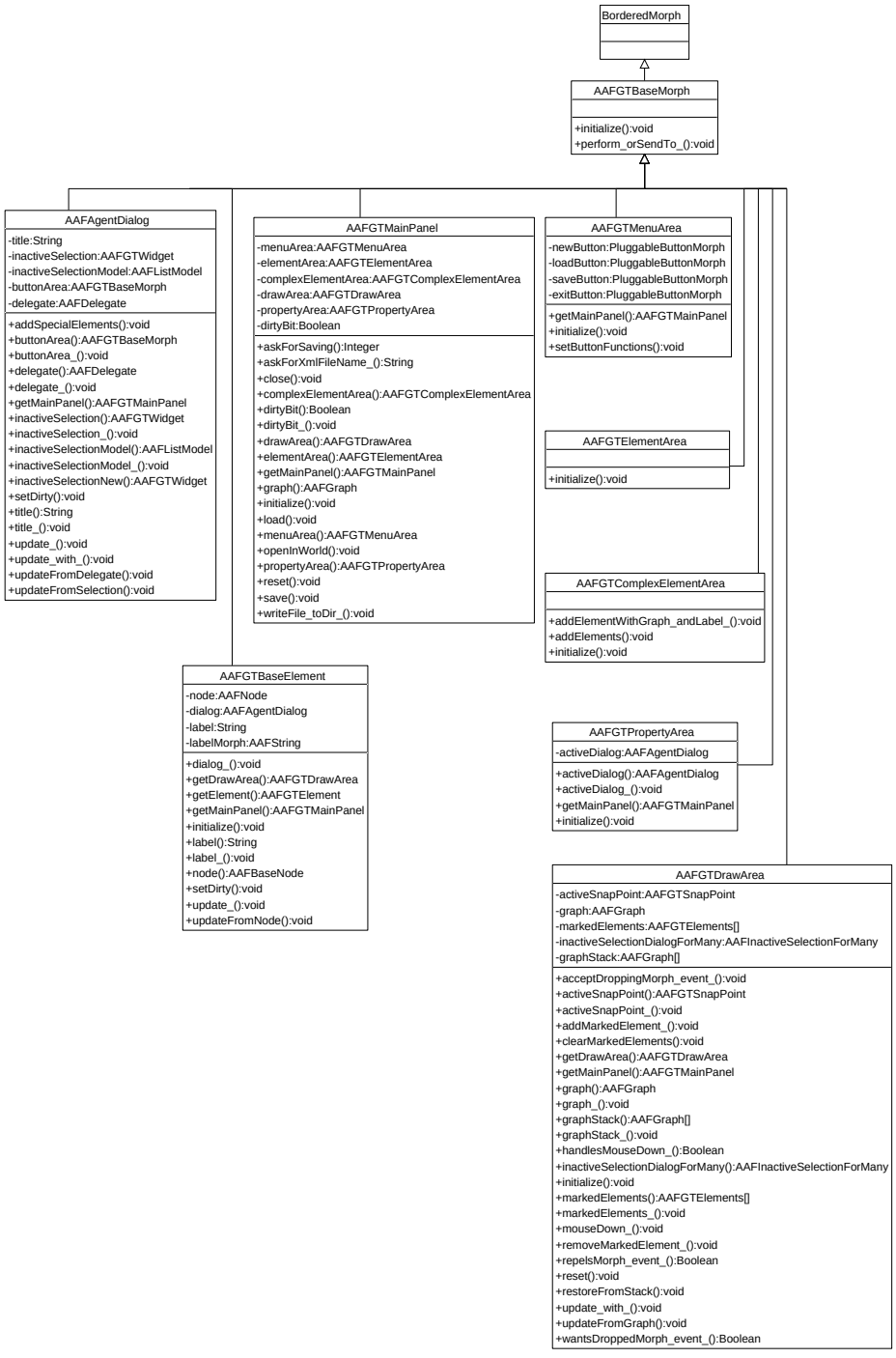


Abbildung 28: Basismorph: `AAFGTBaseMorph`

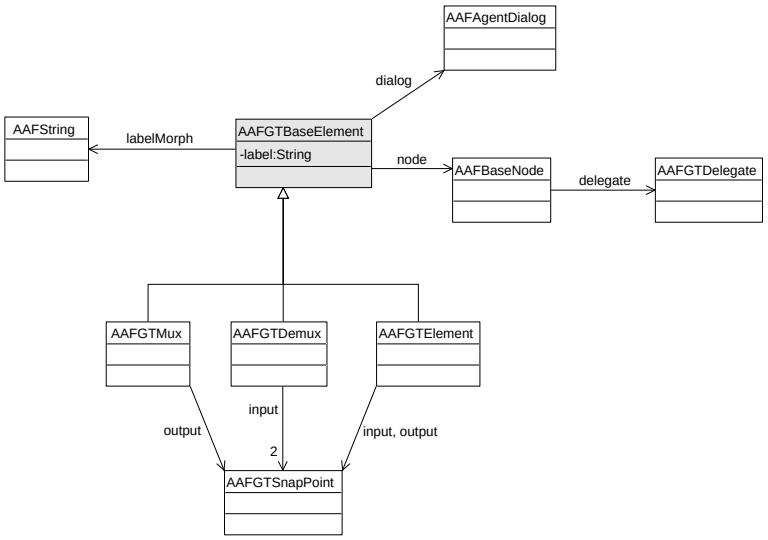


Abbildung 29: Elemente: `AAFGTBasisElement`

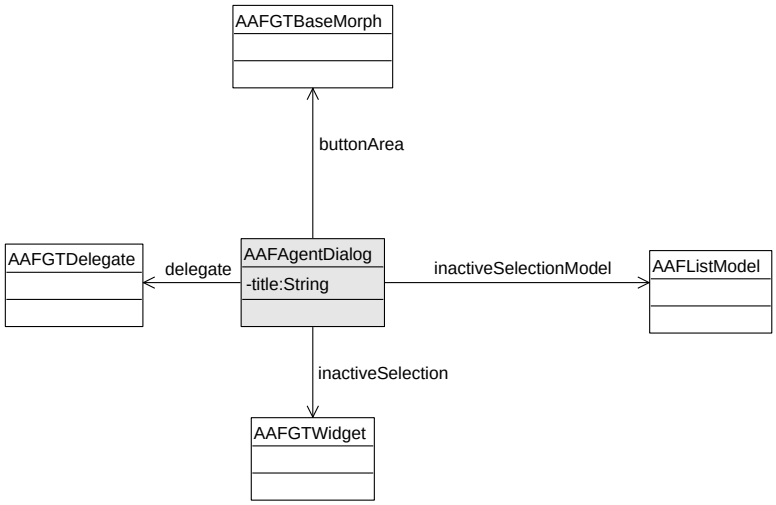


Abbildung 30: Dialoge: `AAFAgentDialog`

C

XML-SCHEMA FÜR DAS AUTOMATIKEN-SPEICHERFORMAT

```
<schema
  xmlns='http://www.w3.org/2000/10/XMLSchema'
  targetNamespace='http://www.w3.org/namespace/'
  xmlns:t='http://www.w3.org/namespace/'>

 <element name='graph'>
  <complexType>
   <sequence>
    <element ref='t:properties'/>
    <element ref='t:root'/>
    <choice minOccurs='0' maxOccurs='unbounded'>
     <element ref='t:node'/>
     <element ref='t:complexNode'/>
    </choice>
    <element ref='t:sink'/>
    <element ref='t:connection' minOccurs='0' maxOccurs='
    unbounded'/>
   </sequence>
  </complexType>
 </element>

 <element name='properties'>
  <complexType>
   <choice maxOccurs='unbounded'>
    <element ref='t:prop'/>
    <element ref='t:proplist'/>
   </choice>
  </complexType>
 </element>

 <element name='prop'>
  <complexType>
   <attribute name='key' type='string' use='required'/>
   <attribute name='value' type='string' use='required'/>
  </complexType>
 </element>

 <element name='proplist'>
  <complexType>
   <element ref='t:propval' maxOccurs='unbounded'/>
   <attribute name='key' type='string' use='required'/>
  </complexType>
 </element>

 <element name='propval'>
  <complexType>
   <attribute name='value' type='string' use='required'/>
  </complexType>
 </element>

 <element name='root'>
  <complexType>
```

```
52    <sequence>
53     <element ref='t:id'/>
54     <element ref='t:label'/>
55     <element ref='t:position'/>
56    </sequence>
57   </complexType>
58  </element>
59
60  <element name='complexNode'>
61   <complexType>
62    <sequence>
63     <element ref='t:id'/>
64     <element ref='t:label'/>
65     <element ref='t:position'/>
66     <element ref='t:graph'/>
67    </sequence>
68   </complexType>
69  </element>
70
71  <element name='node'>
72   <complexType>
73    <sequence>
74     <element ref='t:id'/>
75     <element ref='t:label'/>
76     <element ref='t:position'/>
77     <element ref='t:agent'/>
78    </sequence>
79   </complexType>
80  </element>
81
82  <element name='sink'>
83   <complexType>
84    <sequence>
85     <element ref='t:id'/>
86     <element ref='t:label'/>
87     <element ref='t:position'/>
88    </sequence>
89   </complexType>
90  </element>
91
92  <element name='id'>
93   <complexType>
94    <attribute name='value' type='string' use='required'/>
95   </complexType>
96  </element>
97
98  <element name='label'>
99   <complexType>
100   <attribute name='name' type='string' use='required'/>
101  </complexType>
102 </element>
103
104 <element name='position'>
105  <complexType>
106   <attribute name='x' type='string' use='required'/>
107   <attribute name='y' type='string' use='required'/>
108  </complexType>
109 </element>
110
111 <element name='agent'>
112  <complexType>
```

```
113     <sequence>
114      <element ref='t:properties'/>
115     </sequence>
116     <attribute name='type' type='string' use='required'/>
117    </complexType>
118   </element>
119
120   <element name='connection'>
121    <complexType>
122     <attribute name='parent' type='string' use='required'/>
123     <attribute name='child' type='string' use='required'/>
124    </complexType>
125   </element>
126 </schema>
```

D

DOKUMENTATION VON KLASSEN UND METHODEN

D.1 KLASSEN

KLASSE	BESCHREIBUNG
AAFAgentDialog	Basisklasse zur Erstellung von Eigenschaften-Dialogen.
AAFBreaksAgentDialog	Spezialisierter Eigenschaften-Dialog für Agenten von Typ `AAFBreaksAgent`.
AAFGraphDialog	Spezialisierter Eigenschaften-Dialog für `AAFDelegates` vom Typ `AAFGraph`.
AAFGTBaseElement	Basisklasse zur Erstellung von Elementen, wobei ein Element jeweils einen Graphenknoten grafisch repräsentiert.
AAFGTBaseMorph	Basismorph, auf dem alle weiteren rechteckigen Morphklassen des GUI aufbauen, definiert grundlegende gemeinsame Funktionalität.
AAFGTComplexElementArea	Bereich, in dem komplexe Elemente zur Auswahl bereitgestellt werden.
AAFGTConnection	Grafische Repräsentation der Kanten des Graphen, bildet die Verbindungen zwischen Knoten als Linien zwischen Elementen ab.
AAFGTConsts	Hilfsklasse zum Verwalten konstanter Werte (Farben, Größen, Position, Texte…

Tabelle 6: Klassen des Automatiken-GUI

KLASSE	BESCHREIBUNG
AAFGTDemux	Element, das die Senke des Graphen (Austrittspunkt aus der Automatik) grafisch repräsentiert.
AAFGTDialogUtils	Hilfsklasse zum Erstellen von passenden Eigenschaften-Dialogen für `AAFDelegate`-Objekte.
AAFGTDrawArea	Bereich, in dem die Automatiken (Graphen) im GUI aufgebaut werden.
AAFGTDropDownChoiceMorph	Morph zur Darstellung von Drop-Down-Menüs.
AAFGTElement	Elemente, durch die die Graphenknoten (Automatikenbestandteile) grafisch repräsentiert werden.
AAFGTElementArea	Bereich, in dem einfache Elemente zur Auswahl bereitgestellt werden.
AAFGTImageMorph	Angepasster `ImageMorph` zur Anzeige von Bildern.
AAFGTMainPanel	Container, in dem die verschiedenen Bereiche des GUI angeordnet werden.
AAFGTMainWindow	Systemfenster der Anwendung.
AAFGTMenuArea	Bereich, in dem die Schaltflächen angeordnet werden.
AAFGTMux	Element, das die Wurzel des Graphen (Eintrittspunkt in die Automatik) grafisch repräsentiert.
AAFGTPropertyArea	Bereich, in dem der Eigenschaften-Dialog eines markierten Elements eingeblendet wird.
AAFGTScrollableArea	Stellt für Bereiche, die größer als der Anzeigebereich sind, Scrollfunktionalität zur Verfügung.

Tabelle 6: Klassen des Automatiken-GUI

KLASSE	BESCHREIBUNG
AAFGTSnapPoint	Verbindungspunkt, implementiert die Funktionalität zum Verbinden von Elementen.
AAFGTWidget	Basisklasse zur Implementierung eigener GUI-Elemente.
AAFInactiveSelectionForMany	Eigenschaften-Dialog, über den mehrere Knoten zugleich als aktiv bzw. inaktiv für bestimmte Streckenabschnittstypen gesetzt werden können.
AAFMouseInputAgentDialog	Spezialisierter Eigenschaften-Dialog für Agenten von Typ `AAFMouseInputAgent`.
AAFString	Zeichenkettenklasse zur Verwendung innerhalb des GUI.
AAFTransformMorph	Realisiert die eigentliche Scrollfunktion für die Klasse `AAFGTScrollableArea`.
AAFUtils	Hilfsklasse, stellt einige Hilfsfunktionen für das GUI zur Verfügung.
AAFVisualHintsDemoAgent	Demo-Agent für den Dialog zum Einstellen visueller Hinweise.
AAFVisualHintsDemoAgent-Dialog	Spezialisierter Eigenschaften-Dialog für Agenten von Typ `AAFVisualHintsDemoAgent`.
AAFXMLParser	Parserklasse zum Einlesen der XML-Dateien, in denen die Automatiken (Graphen) gespeichert sind.

Tabelle 6: Klassen des Automatiken-GUI

D.2 METHODEN DER KLASSEN

D.2.1 *AAFAgentDialog*

METHODE	BESCHREIBUNG
addSpecialElements	Erzeugt einen allgemeinen Eigenschaften-Dialog, in dem keinerlei Einstellungen vorgenommen werden können. Muss ggf. von abgeleiteten Klassen überschrieben werden.
buttonArea	Liefert den Morph zurück, in dem die Bedienelemente für den Eigenschaften-Dialog angeordnet werden.
buttonArea:	Setzt einen Morph als den Morph, in dem die Bedienelemente für den Eigenschaften-Dialog angeordnet werden. Erwartet als Parameter ein Objekt vom Typ `AAFGTBaseMorph`.
delegate	Liefert das `AAFDelegate`-Objekt zurück, das dem Eigenschaften-Dialog zugeordnet ist.
delegate:	Nimmt einen Parameter vom Typ `AAFDelegate` und ordnet ihn dem Eigenschaften-Dialog zu.
inactiveSelection	Liefert den `AAFLazyListMorph` zurück, in dem angezeigt wird, für welche Streckenabschnittstypen das `AAFDelegate` (in-)aktiv ist.

Tabelle 7: Instanzmethoden der Klasse *AAFAgentDialog*

METHODE	BESCHREIBUNG
inactiveSelection:	Nimmt ein Objekt vom Typ `AAFLazyListMorph` als Parameter und setzt ihn als Liste, in der angezeigt wird, für welche Streckenabschnittstypen das `AAFDelegate` (in-)aktiv ist.
inactiveSelectionModel	Liefert das dem Eigenschaften-Dialog zugeordnete Listenmodell vom Typ `AAFListModel` zurück, in dem verwaltet wird, für welche Streckenabschnittstypen das `AAFDelegate` aktiv ist.
inactiveSelectionModel:	Nimmt ein Objekt vom Typ `AAFListModel` als Parameter und setzt es als das Listenmodell, in dem verwaltet wird, für welche Streckenabschnittstypen das `AAFDelegate` aktiv ist.
inactiveSelectionNew	Erzeugt das Widget zur Anzeige der (in-)aktiven Streckenabschnittstypen.
title	Liefert die Überschrift des Eigenschaften-Dialogs als `String`-Objekt zurück.
title:	Nimmt ein `String`-Objekt als Parameter und setzt es als Titel des Eigenschaften-Dialogs.
update:	Nimmt ein Squeak-Event und wertet es aus, um den Eigenschaften-Dialog zu aktualisieren.
update:with:	Nimmt ein Squeak-Event und wertet es aus, um den Eigenschaften-Dialog zu aktualisieren. Nimmt als zweites Argument ein weiteres Squeak-Objekt, welches mit dem Event gemeinsam übergeben wird.

Tabelle 7: Instanzmethoden der Klasse *AAFAgentDialog*

METHODE	BESCHREIBUNG
updateFromDelegate	Passt den Eigenschaften-Dialog an die aktuellen Werte des `AAFDelegate`-Objekts an.
updateFromSelection	Passt den Eigenschaften-Dialog an die aktuellen Werte der Auswahl (in-)aktiver Streckenabschnittstypen an.

Tabelle 7: Instanzmethoden der Klasse *AAFAgentDialog*

METHODE	BESCHREIBUNG
new	Verhindert die parameterlose Erzeugung von Objekten der Klasse. Erzwingt die Verwendung von `newForNode:`.
newForNode:	Nimmt ein Objekt vom Typ `AAFNode` als Parameter und erzeugt für diesen einen Eigenschaften-Dialog.

Tabelle 8: Klassenmethoden der Klasse *AAFAgentDialog*

D.2.2 *AAFBreaksAgentDialog*

METHODE	BESCHREIBUNG
addSpecialElements	Fügt die Bedienelemente zum Eigenschaften-Dialog hinzu, die einem `AAFBreaksAgent`-Objekt eigen sind.
updateFromDelegate	Aktualisiert die Werte im Eigenschaften-Dialog, die einem `AAFBreaksAgent`-Objekt eigen sind.

Tabelle 9: Instanzmethoden der Klasse *AAFBreaksAgentDialog*

D.2.3 *AAFGraphDialog*

METHODE	BESCHREIBUNG
addSpecialElements	Fügt die Bedienelemente zum Eigenschaften-Dialog hinzu, die einem `AAFGraph`-Objekt eigen sind.
showInnerGraph:	Verzweigt in einen Untergraphen. Nimmt den Graphen, in den verzweigt werden soll, als Parameter vom Typ `AAFGraph`.

Tabelle 10: Instanzmethoden der Klasse *AAFGraphDialog*

D.2.4 *AAFGTBaseElement*

METHODE	BESCHREIBUNG
dialog:	Ordnet einem Element einen Eigenschaften-Dialog vom Typ `AAFAgentDialog` zu.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
label	Liefert die Beschriftung eines Elements als `String`-Objekt zurück.
label:	Nimmt einen Parameter vom Typ `String` und setzt ihn als Beschriftung des Elements.
node	Liefert den Knoten vom Typ `AAFNode` zurück, der durch das Element repräsentiert wird.

Tabelle 11: Instanzmethoden der Klasse *AAFGTBaseElement*

METHODE	BESCHREIBUNG
update:	Nimmt ein Squeak-Event als Parameter und sorgt ggf. für Anpassung des Elements an Änderungen des zugehörigen Knotens.
updateFromNode	Fragt die Eigenschaften des repräsentierten Knotens ab und aktualisiert ggf. die Anzeige des Elements.

Tabelle 11: Instanzmethoden der Klasse *AAFGTBaseElement*

D.2.5 *AAFGTBaseMorph*

METHODE	BESCHREIBUNG
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
perform:orSendTo:	Überschreibt die gleichnamige Funktion der `Object`-Klasse, da deren Implementierung fehlerhaft ist.

Tabelle 12: Instanzmethoden der Klasse *AAFGTBaseMorph*

D.2.6 *AAFGTComplexElementArea*

METHODE	BESCHREIBUNG
addElementWithGraph: andLabel:	Nimmt als ersten Parameter ein Objekt vom Typ `AAFGraph`. Dieses wird der `AAFGTComplexElementArea` als Element hinzugefügt und mit dem `String`-Objekt beschriftet, das als zweiter Parameter übergeben wurde.

Tabelle 13: Instanzmethoden der Klasse *AAFGTComplexElementArea*

METHODE	BESCHREIBUNG
addElements	Fügt der aktuellen `AAFGTComplexElementArea` die Automatiken aus dem Standardspeicherverzeichnis als komplexe Elemente hinzu.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.

Tabelle 13: Instanzmethoden der Klasse *AAFGTComplexElementArea*

D.2.7 *AAFGTConnection*

METHODE	BESCHREIBUNG
childElement	Liefert das Kindelement vom Typ `AAFGTBaseElement` zurück, welches zu der Verbindung gehört.
childElement:	Nimmt als Parameter ein Objekt vom Typ `AAFGTBaseElement` und setzt es als Kindelement der Verbindung.
childNode	Liefert den Kindknoten vom Typ `AAFBaseNode` zurück, welcher zu der Verbindung gehört.
childNode:	Nimmt als Parameter ein Objekt vom Typ `AAFBaseNode` und setzt es als Kindknoten der Verbindung.
createContextMenu	Erzeugt das Kontextmenü für Objekte vom Typ `AAFGTConnection`.
disconnect	Löscht die Kante.

Tabelle 14: Instanzmethoden der Klasse *AAFGTConnection*

METHODE	BESCHREIBUNG
handlesMouseDown:	Nimmt ein Squeak-Event als Parameter. Liefert ein `Boolean`-Objekt zurück, über das angezeigt wird, ob der Empfänger das Event verarbeiten möchte.
mouseDown:	Nimmt ein Squeak-Event als Parameter und verarbeitet es.
parentElement	Liefert das Elternelement vom Typ `AAFGTBaseElement` zurück, welches zu der Verbindung gehört.
parentElement:	Nimmt als Parameter ein Objekt vom Typ `AAFGTBaseElement` und setzt es als Elternelement der Verbindung.
parentNode	Liefert den Elternknoten vom Typ `AAFBaseNode` zurück, welcher zu der Verbindung gehört.
parentNode:	Nimmt als Parameter ein Objekt vom Typ `AAFBaseNode` und setzt es als Elternknoten der Verbindung.
perform:orSendTo:	Überschreibt die gleichnamige Funktion der `Object`-Klasse, da deren Implementierung fehlerhaft ist.
update:	Nimmt ein Squeak-Event als Parameter und sorgt ggf. für Anpassung der Kante an Änderungen der zugehörigen Knoten oder Elemente.

Tabelle 14: Instanzmethoden der Klasse *AAFGTConnection*

METHODE	BESCHREIBUNG
from:to:color:width:	Nimmt als ersten Parameter einen Startpunkt vom Typ `Point` und als zweiten einen Endpunkt vom Typ `Point`. Erzeugt eine Verbindungslinie zwischen diesen beiden Punkten. Die Farbe der Linie wird durch den dritten Parameter vom `Color` bestimmt, die Breite vom vierten Parameter vom Typ `Integer`.
fromNode:toNode: withElement:andElement:	Erzeugt eine Kante, zu der ein Kind- und ein Elternelement sowie ein Kind- und ein Elternknoten gehören. Der Elternknoten wird als erstes Element übergeben, der Kindknoten als zweites, beide sind vom Typ `AAFBaseNode`. Der Elternknoten wird als dritter und der Kindknoten als vierter Parameter übergeben, beide sind vom Typ `AAFBaseNode`.

Tabelle 15: Klassenmethoden der Klasse *AAFGTConnection*

D.2.8 *AAFGTConsts*

METHODE	BESCHREIBUNG
activeSelectionColor	Liefert die Standardfarbe für die Anzeige eines Streckenabschnittstypen, für den die Automatik aktiviert ist, als Objekt vom Typ `Color`.
adjustment	Liefert die sprachabhängige Zeichenkette für *adjustment*.
backgroundColor	Liefert die Hintergrundfarbe des GUI-Systemfensters als Objekt vom Typ `Color`.

Tabelle 16: Klassenmethoden der Klasse *AAFGTConsts*

METHODE	BESCHREIBUNG
baseColor	Liefert die Grundfarbe für das GUI als Objekt vom Typ `Color`.
buttonColor	Liefert die Farbe für Schaltflächen als Objekt vom Typ `Color`.
chooseActiveSections	Liefert die sprachabhängige Zeichenkette für *choose active sections* (für Eigenschaften-Dialoge).
closeInnerGraph	Liefert die sprachabhängige Aufschrift der Schaltfläche zum Schließen eines inneren Graphen.
complexElementArea-BalloonHint	Liefert den sprachabhängigen Tooltipp für die `AAFGTComplexElementArea`.
complexElementAreaLabel	Liefert die sprachabhängige Beschriftung der `AAFGTComplexElementArea`.
complexElementArea-LayoutFrame	Liefert den Layoutrahmen für die Platzierung der `AAFGTComplexElementArea` als Objekt vom Typ `LayoutFrame`.
connectionLineWidth	Liefert die Breite einer Verbindungslinie.
drawAreaBalloonHint	Liefert den sprachabhängigen Tooltipp für die `AAFGTDrawArea`.
drawAreaLabel	Liefert die sprachabhängige Beschriftung der `AAFGTDrawArea`.
drawAreaLayoutFrame	Liefert den Layoutrahmen für die Platzierung der `AAFGTDrawArea` als Objekt vom Typ `LayoutFrame`.
drawAreaSize	Liefert die Größe für `AAFGTDrawArea` als Objekt vom Typ `Point`.

Tabelle 16: Klassenmethoden der Klasse *AAFGTConsts*

METHODE	BESCHREIBUNG
elementActiveColor	Liefert die Farbe für markierte Elemente als Objekt vom Typ `Color`.
elementAreaBalloonHint	Liefert den sprachabhängigen Tooltipp für die `AAFGTElementArea`.
elementAreaLabel	Liefert die sprachabhängige Beschriftung der `AAFGTElementArea`.
elementAreaLayoutFrame	Liefert den Layoutrahmen für die Platzierung der `AAFGTElementArea` als Objekt vom Typ `LayoutFrame`.
elementBalloonHint	Liefert den sprachabhängigen Tooltipp für Elemente.
elementInactiveColor	Liefert die Farbe für nicht markierte Elemente als Objekt vom Typ `Color`.
exitButtonBalloonHint	Liefert den sprachabhängigen Tooltipp für die *Verlassen*-Schaltfläche.
exitButtonLabel	Liefert die sprachabhängige Aufschrift der *Verlassen*-Schaltfläche.
inactiveSelectionColor	Liefert die Standardfarbe für die Anzeige eines Streckenabschnittstypen, für den die Automatik deaktiviert ist, als Objekt vom Typ `Color`.
language	Liefert die aktuell eingestellte Sprache des GUI.
lineColor	Liefert die Standardfarbe für Linien als Objekt vom Typ `Color`.
loadButtonBalloonHint	Liefert den sprachabhängigen Tooltipp für die *Öffnen*-Schaltfläche.
loadButtonLabel	Liefert die sprachabhängige Aufschrift der *Öffnen*-Schaltfläche.

Tabelle 16: Klassenmethoden der Klasse *AAFGTConsts*

METHODE	BESCHREIBUNG
lookAhead	Liefert die sprachabhängige Zeichenkette für *look ahead* (für Eigenschaften-Dialoge).
mainPanelSize	Liefert die Größe für `AAFGTMainPanel` als Objekt vom Typ `Point`.
mainWindowPosition	Liefert die Position des GUI in der Squeak-Welt als Objekt vom Typ `Point`.
mainWindowSize	Liefert die Größe des GUI-Hauptfensters.
menuAreaLayoutFrame	Liefert den Layoutrahmen für die Platzierung der `AAFGTMenuArea` als Objekt vom Typ `LayoutFrame`.
mixedSelectionColor	Liefert die Standardfarbe für die Anzeige eines Streckenabschnittstypen, der bei Auswahl mehrerer Elemente zugleich für einige aktiviert und einige deaktiviert ist. Rückgabewert ist vom Typ `Color`.
newButtonBalloonHint	Liefert den sprachabhängigen Tooltipp für die *Neu*-Schaltfläche.
newButtonLabel	Liefert die sprachabhängige Aufschrift der *Neu*-Schaltfläche.
noProps	Liefert die sprachabhängige Zeichenkette für *no properties available* (für Eigenschaften-Dialoge).
pointerOff	Liefert die sprachabhängige Zeichenkette für *pointer is off* (für Eigenschaften-Dialoge).
pointerOn	Liefert die sprachabhängige Zeichenkette für *pointer is on* (für Eigenschaften-Dialoge).
propertyAreaBalloonHint	Liefert den sprachabhängigen Tooltipp für die `AAFGTPropertyArea`.

Tabelle 16: Klassenmethoden der Klasse *AAFGTConsts*

METHODE	BESCHREIBUNG
propertyAreaLabel	Liefert die sprachabhängige Beschriftung der `AAFGTPropertyArea`.
propertyAreaLayoutFrame	Liefert den Layoutrahmen für die Platzierung der `AAFGTPropertyArea` als Objekt vom Typ `LayoutFrame`.
removeElementLabel	Liefert die sprachabhängige Zeichenkette für *remove element*.
removeLineLabel	Liefert die sprachabhängige Zeichenkette für *remove connection*.
rootBalloonHint	Liefert den sprachabhängigen Tooltipp für das Wurzelelement.
rootLabel	Liefert die sprachabhängige Beschriftung des Wurzelelements.
rootPosition	Liefert die Position des Wurzelelements im Arbeitsbereich als Objekt vom Typ `Point`.
saveButtonBalloonHint	Liefert den sprachabhängigen Tooltipp für die *Speichern*-Schaltfläche.
saveButtonLabel	Liefert die sprachabhängige Aufschrift der *Speichern*-Schaltfläche.
showInnerGraph	Liefert die sprachabhängige Aufschrift der Schaltfläche zum Verzweigen in einen inneren Graphen.
sinkBalloonHint	Liefert den sprachabhängigen Tooltipp für das Senkenelement.
sinkLabel	Liefert die sprachabhängige Beschriftung des Senkenelements.

Tabelle 16: Klassenmethoden der Klasse *AAFGTConsts*

METHODE	BESCHREIBUNG
sinkPosition	Liefert die Position des Senkenelements im Arbeitsbereich als Objekt vom Typ `Point`.
snapPointActiveColor	Liefert die Farbe für einen aktivierten Verbindungspunkt als Objekt vom Typ `Color`.
snapPointInactiveColor	Liefert die Farbe für einen inaktiven Verbindungspunkt als Objekt vom Typ `Color`.
theRealBackgroundColor	Liefert die Hintergrundfarbe des GUI als Objekt vom Typ `Color`.
threshold	Liefert die sprachabhängige Zeichenkette für *threshold* (für Eigenschaften-Dialoge).
trackTypes	Liefert die sprachabhängigen Bezeichnungen der Streckenabschnittstypen als Objekt vom Typ `OrderedCollection`.
xmlDefaultFileName	Liefert den Standarddateinamen zum Speichern einer Automatik.
xmlDir	Liefert den Pfad zum Standardspeicherverzeichnis für Automatiken.

Tabelle 16: Klassenmethoden der Klasse *AAFGTConsts*

D.2.9 *AAFGTDemux*

METHODE	BESCHREIBUNG
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
input	Liefert den Eingangsverbindungspunkt vom Typ `AAFGTSnapPoint` zurück.

Tabelle 17: Instanzmethoden der Klasse *AAFGTDemux*

METHODE	BESCHREIBUNG
mouseDown:	Nimmt ein Squeak-Event als Parameter und verarbeitet es.
node:	Nimmt ein Objekt vom Typ `AAFDemuxNode` und setzt es als den repräsentierten Knoten.
updateFromNode	Fragt die Eigenschaften des repräsentierten Knotens ab und aktualisiert ggf. die Anzeige des Elements.

Tabelle 17: Instanzmethoden der Klasse *AAFGTDemux*

D.2.10 *AAFGTDialogUtils*

METHODE	BESCHREIBUNG
createDialogFor:	Nimmt als Parameter ein Objekt vom Typ `AAFNode` und erzeugt einen Eigenschaften-Dialog vom Typ `AAFAgentDialog`, der zum im Knoten enthaltenen `AAFDelegate` passt. Liefert den Eigenschaften-Dialog zurück.

Tabelle 18: Klassenmethoden der Klasse *AAFGTDialogUtils*

D.2.11 *AAFGTDrawArea*

METHODE	BESCHREIBUNG
acceptDroppingMorph:event:	Nimmt als ersten Parameter ein Objekt vom Typ `Morph` und als zweiten ein Squeak-Event. Definiert Aktionen, die ausgeführt werden, nachdem das `Morph`-Objekt im Arbeitsbereich fallen gelassen und akzeptiert wurde.

Tabelle 19: Instanzmethoden der Klasse *AAFGTDrawArea*

METHODE	BESCHREIBUNG
activeSnapPoint	Liefert den aktuell aktivierten Verbindungspunkt als Objekt vom Typ `AAFGTSnapPoint`. Liefert `nil`, wenn kein Verbindungspunkt aktiviert ist.
activeSnapPoint:	Nimmt ein Objekt vom Typ `AAFGTSnapPoint` und setzt ihn als aktuell aktivierten Verbindungspunkt.
addMarkedElement:	Nimmt als Parameter ein Objekt vom Typ `AAFGTElement` und fügt es der Menge der markierten Elemente hinzu. Damit ist das Element markiert.
clearMarkedElements	Löscht die Menge der markierten Elemente. Damit ist kein Element mehr markiert.
graph	Gibt die aktuell angezeigte Automatik vom Typ `AAFGraph` zurück.
graph:	Nimmt einen Parameter vom Typ `AAFGraph` und setzt ihn als zu bearbeitende Automatik.
graphStack	Liefert den Stack vom Typ `OrderedCollection` zurück, auf dem die (Unter-)Graphen beim Verzweigen in komplexe Elemente verwaltet werden.
graphStack:	Nimmt einen Parameter vom Typ `OrderedCollection` und setzt ihn als Stack zur Verwaltung von (Unter-)Graphen beim Verzweigen in komplexe Elemente.

Tabelle 19: Instanzmethoden der Klasse *AAFGTDrawArea*

METHODE	BESCHREIBUNG
handlesMouseDown:	Nimmt ein Squeak-Event als Parameter. Liefert ein `Boolean`-Objekt zurück, über das angezeigt wird, ob der Empfänger das Event verarbeiten möchte.
inactiveSelectionDialog-ForMany	Liefert den Dialog zum (De-)-Aktivieren mehrerer Elemente gleichzeitig für bestimmte Streckenabschnittstypen zurück.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
markedElements	Liefert die Menge der markierten Elemente als `OrderedCollection` zurück.
markedElements:	Nimmt eine `OrderedCollection` als Parameter und setzt sie als Menge der markierten Elemente.
mouseDown:	Nimmt ein Squeak-Event als Parameter und verarbeitet es.
removeMarkedElement:	Nimmt als Parameter ein Objekt vom Typ `AAFGTElement` und entfernt es aus der Menge der markierten Elemente. Damit ist das Element nicht mehr markiert.
repelsMorph:event:	Negation von `wantsDroppedMorph:event:`
reset	Versetzt die Arbeitsfläche in den Anfangszustand.
restoreFromStack	Nimmt den obersten Graphen vom `graphStack` und setzt ihn als aktuelle Automatik.

Tabelle 19: Instanzmethoden der Klasse *AAFGTDrawArea*

METHODE	BESCHREIBUNG
update:with:	Nimmt ein Squeak-Event und wertet es aus, um die `AAFGTDrawArea` zu aktualisieren. Nimmt als zweites Argument ein weiteres Squeak-Objekt, welches mit dem Event gemeinsam übergeben wird.
updateFromGraph	Fragt die Eigenschaften des angezeigten Graphen ab und aktualisiert ggf. die Arbeitsfläche.
wantsDroppedMorph:event:	Nimmt als ersten Parameter ein `Morph`-Objekt und als zweiten ein Squeak-Event. Entscheidet, ob der Morph per Drag&Drop der Arbeitsfläche hinzugefügt werden darf.

Tabelle 19: Instanzmethoden der Klasse *AAFGTDrawArea*

D.2.12 *AAFGTDropDownChoiceMorph*

METHODE	BESCHREIBUNG
actionTarget	Liefert den Empfänger von Aktionen bei Klick als `Object`.
actionTarget:	Nimmt ein `Object` als Parameter. Setzt es als Empfänger von Aktionen bei Klick.
arguments:	Nimmt ein `OrderedCollection`-Objekt und setzt es als Argumentmenge für die Aktion bei Klick.
mouseDown:	Nimmt ein Squeak-Event als Parameter und verarbeitet es.

Tabelle 20: Instanzmethoden der Klasse *AAFGTDropDownChoiceMorph*

D.2.13 *AAFGTElement*

METHODE	BESCHREIBUNG
click:	Nimmt als Element ein Squeak-Event. Behandelt Klickereignisse.
clickOptionsForDrawArea:	Nimmt als Element ein Squeak-Event. Behandelt Klickereignisse, die im Arbeitsbereich stattgefunden haben.
clickOptionsForElementArea:	Nimmt als Element ein Squeak-Event. Behandelt Klickereignisse, die im Elementauswahlbereich stattgefunden haben.
cloneNode	Erzeugt eine 1:1-Kopie des durch das Element repräsentierten Knotens.
createContextMenu	Erzeugt das Kontextmenü für Elemente vom Typ `AAFGTElement`.
delete	Löscht das Element.
doubleClick:	Nimmt als Element ein Squeak-Event. Behandelt Doppelklickereignisse.
doubleClickOptions-ForDrawArea:	Nimmt als Element ein Squeak-Event. Behandelt Doppelklickereignisse, die im Arbeitsbereich stattgefunden haben.
doubleClickOptions-ForElementArea:	Nimmt als Element ein Squeak-Event. Behandelt Doppelklickereignisse, die im Elementauswahlbereich stattgefunden haben.
dragOptionsForDrawArea:	Nimmt als Element ein Squeak-Event. Behandelt Drag-Ereignisse, die im Arbeitsbereich stattgefunden haben.

Tabelle 21: Instanzmethoden der Klasse *AAFGTElement*

METHODE	BESCHREIBUNG
dragOptionsForElementArea:	Nimmt als Element ein Squeak-Event. Behandelt Drag-Ereignisse, die im Elementauswahlbereich stattgefunden haben.
handleMouseToEditLabel:	Nimmt als Element ein Squeak-Event. Setzt das Editieren eines Elementlabels um.
handleMouseToMarkElements:	Nimmt als Element ein Squeak-Event. Setzt das Markieren von Elementen um.
handleMouseToPickUp-Element:	Nimmt als Element ein Squeak-Event. Setzt das Aufnehmen von Elementen mit der Maus um.
handleMouseToPickUp-ElementCopy:	Nimmt als Element ein Squeak-Event. Setzt das Erzeugen und Aufnehmen einer Elementkopie mit der Maus um.
handleMouseToShowProps:	Nimmt als Element ein Squeak-Event. Setzt das Anzeigen der Eigenschaften bei Klick mit der Maus um.
handlesMouseDown:	Nimmt ein Squeak-Event als Parameter. Liefert ein `Boolean`-Objekt zurück, über das angezeigt wird, ob der Empfänger das Event verarbeiten möchte.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
input	Liefert den Eingangsverbindungspunkt vom Typ `AAFGTSnapPoint` zurück.

Tabelle 21: Instanzmethoden der Klasse *AAFGTElement*

METHODE	BESCHREIBUNG
justDroppedInto:event:	Nimmt als ersten Parameter einen `Morph` und als zweiten ein Squeak-Event. Definiert Aktionen, die ausgeführt werden sollen, wenn ein Element nach dem Verschieben in dem `Morph` fallen gelassen wird.
mouseDown:	Nimmt ein Squeak-Event als Parameter und verarbeitet es.
node:	Nimmt ein Objekt vom Typ `AAFNode` und setzt es als den repräsentierten Knoten.
output	Liefert den Ausgangsverbindungspunkt vom Typ `AAFGTSnapPoint` zurück.
rightClickOptions-ForDrawArea:	Nimmt als Element ein Squeak-Event. Behandelt Rechtsklickereignisse, die im Arbeitsbereich stattgefunden haben.
showProps	Blendet den Eigenschaften-Dialog des Elements in den Eigenschaftenbereich ein.
startDrag:	Nimmt als Element ein Squeak-Event. Behandelt den Start von Drag-Ereignissen.
updateFromNode	Fragt die Eigenschaften des repräsentierten Knotens ab und aktualisiert ggf. die Anzeige des Elements.
wantsToBeDroppedInto:	Nimmt als Parameter ein `Morph`-Objekt. Entscheidet, ob das Element in diesem `Morph` fallen gelassen werden kann.

Tabelle 21: Instanzmethoden der Klasse *AAFGTElement*

D.2.14 *AAFGTElementArea*

METHODE	BESCHREIBUNG
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.

Tabelle 22: Instanzmethoden der Klasse *AAFGTElementArea*

D.2.15 *AAFGTImageMorph*

METHODE	BESCHREIBUNG
handlesMouseDown:	Nimmt ein Squeak-Event als Parameter. Liefert ein `Boolean`-Objekt zurück, über das angezeigt wird, ob der Empfänger das Event verarbeiten möchte.
justDroppedInto:event:	Nimmt als ersten Parameter einen `Morph` und als zweiten ein Squeak-Event. Definiert Aktionen, die ausgeführt werden sollen, wenn ein Element nach dem Verschieben in dem `Morph` fallen gelassen wird.
mouseDown:	Nimmt ein Squeak-Event als Parameter und verarbeitet es.
startDrag:	Nimmt ein Squeak-Event als Parameter und verarbeitet es, wenn es sich um den Start einer Drag-Aktion handelt.
wantsToBeDroppedInto:	Nimmt als Parameter ein `Morph`-Objekt. Entscheidet, ob das Element in diesem `Morph` fallen gelassen werden kann.

Tabelle 23: Instanzmethoden der Klasse *AAFGTImageMorph*

METHODE	BESCHREIBUNG
newWithForm:	Nimmt als Parameter ein `Form`-Objekt und erzeugt einen passenden `AAFGTImageMorph`.

Tabelle 24: Klassenmethoden der Klasse *AAFGTImageMorph*

D.2.16 *AAFGTMainPanel*

METHODE	BESCHREIBUNG
askForSaving	Popup-Menu, welches fragt, ob Änderungen gespeichert werden sollen.
askForXmlFileName:	Fragt den Dateinamen zum Speichern einer Automatik ab. Nimmt einen Parameter vom Typ`String`, welcher als Dateiname vorgeschlagen wird.
close	Schließt das GUI.
complexElementArea	Liefert den Elementauswahlbereich für komplexe Elemente als Objekt vom Typ `AAFGTComplexElementArea` zurück.
dirtyBit	Liefert das Dirty Bit als Objekt vom Typ `Boolean` zurück.
dirtyBit:	Nimmt einen Parameter vom Typ `Boolean` und setzt das Dirty Bit auf diesen Wert.
drawArea	Liefert den Arbeitsbereich als Objekt vom Typ `AAFGTDrawArea` zurück.
elementArea	Liefert den Elementauswahlbereich für einfache Elemente als Objekt vom Typ `AAFGTElementArea` zurück.

Tabelle 25: Instanzmethoden der Klasse *AAFGTMainPanel*

METHODE	BESCHREIBUNG
graph	Gibt die aktuell angezeigte Automatik vom Typ `AAFGraph` zurück.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
load	Lädt eine Automatik.
menuArea	Liefert den Menübereich als Objekt vom Typ `AAFGTMenuArea` zurück.
openInWorld	Öffnet das Haupt-Panel in der aktuellen Squeak-Welt.
propertyArea	Liefert den Eigenschaftenbereich als Objekt vom Typ `AAFGTPropertyArea` zurück.
reset	Versetzt das GUI in den Anfangszustand zurück.
save	Speichert die aktuelle Automatik.
writeFile:toDir:	Nimmt als ersten Parameter einen Dateinamen und als zweiten einen Verzeichnisnamen. Beide Parameter sind vom Typ `String`. Schreibt die aktuelle Automatik in eine Datei mit dem angegebenen Namen im angegebenen Verzeichnis.

Tabelle 25: Instanzmethoden der Klasse *AAFGTMainPanel*

D.2.17 *AAFGTMainWindow*

METHODE	BESCHREIBUNG
addMorph:fullFrame:	Nimmt als ersten Parameter ein Objekt vom Typ `Morph` und als zweiten ein Objekt vom Typ `LayoutFrame`. Fügt das `Morph`-Objekt innerhalb des `LayoutFrame` dem Anwendungsfenster hinzu.
closeBoxHit	Schließt das GUI.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
openInWorld	Öffnet das Anwendungsfenster in der aktuellen Squeak-Welt.
panel	Liefert das Haupt-Panel als Objekt vom Typ `AAFGTMainPanel` zurück.

Tabelle 26: Instanzmethoden der Klasse *AAFGTMainWindow*

METHODE	BESCHREIBUNG
open	Öffnet das GUI in der aktuellen Squeak-Welt.

Tabelle 27: Klassenmethoden der Klasse *AAFGTMainWindow*

D.2.18 *AAFGTMenuArea*

METHODE	BESCHREIBUNG
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
setButtonFunctions	Ordnet den Schaltflächen im Menübereich Funktionen zu.

Tabelle 28: Instanzmethoden der Klasse *AAFGTMenuArea*

D.2.19 *AAFGTMux*

METHODE	BESCHREIBUNG
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
mouseDown:	Nimmt ein Squeak-Event als Parameter und verarbeitet es.
node:	Nimmt ein Objekt vom Typ `AAFMuxNode` und setzt es als den repräsentierten Knoten.
output	Liefert den Ausgangsverbindungspunkt vom Typ `AAFGTSnapPoint` zurück.
updateFromNode	Fragt die Eigenschaften des repräsentierten Knotens ab und aktualisiert ggf. die Anzeige des Elements.

Tabelle 29: Instanzmethoden der Klasse *AAFGTMux*

D.2.20 *AAFGTPropertyArea*

METHODE	BESCHREIBUNG
activeDialog	Liefert den aktuell eingeblendeten Eigenschaften-Dialog vom Typ `AAFAgentDialog` zurück.
activeDialog:	Nimmt als Parameter einen `AAFAgentDialog` und blendet ihn als aktuellen Dialog ein.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.

Tabelle 30: Instanzmethoden der Klasse *AAFGTPropertyArea*

D.2.21 *AAFGTScrollableArea*

METHODE	BESCHREIBUNG
addLabel	Zeigt den Inhalt der Instanzvariable `label` im Bereich an.
adjustPasteUpSize	Passt die Größe der zu scrollenden Fläche an.
doLayoutIn:	Layoutet die `AAFGTScrollableArea` innerhalb des als Parameter übergebenen `Rectangle`-Objekts.
hExtraScrollRange	Zusatzraum, der horizontal hinzugefügt wird.
hMargin	Breite des horizontalen Randes.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
label	Liefert die Beschriftung als `String` zurück.

Tabelle 31: Instanzmethoden der Klasse *AAFGTScrollableArea*

METHODE	BESCHREIBUNG
label:	Nimmt als Parameter einen `String` und setzt ihn als Beschriftung.
theArea	Liefert den Bereich vom Typ `AAFGTBaseMorph` zurück, der sich innerhalb des Scrollbereichs befindet.
theArea:	Nimmt einen Parameter vom Typ `AAFGTBaseMorph` und setzt ihn in den Scrollbereich.
vExtraScrollRange	Zusatzraum, der vertikal hinzugefügt wird.

Tabelle 31: Instanzmethoden der Klasse *AAFGTScrollableArea*

METHODE	BESCHREIBUNG
new:	Nimmt einen Parameter vom Typ `AAFGTBaseMorph` und erzeugt einen Scrollbereich für diesen.
new:label:	Nimmt einen ersten Parameter vom Typ `AAFGTBaseMorph` einen zweiten vom Typ `String`. Erzeugt einen Scrollbereich für den Bereich und beschriftet ihn mit dem `String`.

Tabelle 32: Klassenmethoden der Klasse *AAFGTScrollableArea*

D.2.22 *AAFGTSnapPoint*

METHODE	BESCHREIBUNG
handlesMouseDown:	Nimmt ein Squeak-Event als Parameter. Liefert ein `Boolean`-Objekt zurück, über das angezeigt wird, ob der Empfänger das Event verarbeiten möchte.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
isInput	Liefert ein `Boolean`-Objekt, welches anzeigt, ob es sich bei dem Verbindungspunkt um einen Eingang handelt.
isInput:	Nimmt ein `Boolean`-Objekt als Parameter. Legt fest, ob es sich bei dem Verbindungspunkt um einen Eingang handelt.
mouseDown:	Nimmt ein Squeak-Event als Parameter und verarbeitet es.
toggleActive	Schaltet den Zustand eines Verbindungspunkts zwischen aktiviert und deaktiviert um.

Tabelle 33: Instanzmethoden der Klasse *AAFGTSnapPoint*

METHODE	BESCHREIBUNG
newBounds:isInput:	Nimmt als ersten Parameter ein `Rectangle`-Objekt und als zweiten ein `Boolean`-Objekt. Erzeugt einen Verbindungspunkt, platziert ihn innerhalb des `Rectangle`-Objekts und setzt das `Boolean`-Objekt als Wert für `isInput`.

Tabelle 34: Klassenmethoden der Klasse *AAFGTSnapPoint*

D.2.23 *AAFGTWidget*

METHODE	BESCHREIBUNG
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.

Tabelle 35: Instanzmethoden der Klasse *AAFGTWidget*

D.2.24 *AAFInactiveSelectionForMany*

METHODE	BESCHREIBUNG
update:with:	Nimmt ein Squeak-Event und wertet es aus, um die Auswahl (in-)aktiver Streckenabschnittstypen zu aktualisieren, wenn mehrere Elemente zugleich ausgewählt sind. Nimmt als zweites Argument ein weiteres Squeak-Objekt, welches mit dem Event gemeinsam übergeben wird.

Tabelle 36: Instanzmethoden der Klasse *AAFInactiveSelectionForMany*

METHODE	BESCHREIBUNG
new	Erzeugt einen Dialog zum (De-)Aktivieren mehrerer Elemente gleichzeitig für bestimmte Streckenabschnittstypen.

Tabelle 37: Klassenmethoden der Klasse *AAFInactiveSelectionForMany*

D.2.25 *AAFMouseInputAgentDialog*

METHODE	BESCHREIBUNG
addSpecialElements	Fügt die Bedienelemente zum Eigenschaften-Dialog hinzu, die einem `AAFMouseInputAgent`-Objekt eigen sind.
updateFromDelegate	Aktualisiert die Werte im Eigenschaften-Dialog, die einem `AAFMouseInputAgent`-Objekt eigen sind.

Tabelle 38: Instanzmethoden der Klasse *AAFMouseInputAgentDialog*

D.2.26 *AAFString*

METHODE	BESCHREIBUNG
handlesMouseDown:	Nimmt ein Squeak-Event als Parameter. Liefert ein `Boolean`-Objekt zurück, über das angezeigt wird, ob der Empfänger das Event verarbeiten möchte.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
mouseDown:	Nimmt ein Squeak-Event als Parameter und verarbeitet es.

Tabelle 39: Instanzmethoden der Klasse *AAFString*

D.2.27 *AAFUtils*

METHODE	BESCHREIBUNG
applyDefaultButtonProperties:	Nimmt als Parameter einen `PluggableButtonMorph` und setzt ihn auf Standardeigenschaften.
applyDefaultToggleButton-Properties:	Nimmt als Parameter einen Togglebutton und setzt ihn auf Standardeigenschaften.
convert:type:	Nimmt als Parameter zwei `String`-Objekte. Wandelt das erste in ein Objekt von dem Typ um, der durch das zweite angegeben wurde.

Tabelle 40: Klassenmethoden der Klasse *AAFUtils*

D.2.28 *AAFVisualHintsDemoAgent*

METHODE	BESCHREIBUNG
compute:	Nimmt ein `SamState`-Objekt als Parameter und verarbeitet ihn. Liefert ein `SamState`-Objekt zurück.
getAllProps	Liefert alle Eigenschaften des Agenten als `Dictionary`.
hint	Liefert den Pfad zum gesetzten Hinweisbild als `String`.
hint:	Nimmt einen Parameter vom Typ `String`. Setzt ihn als Pfad zum Hinweisbild.
image1	Liefert den Pfad zum ersten gesetzten Streckenbild als `String`.
image1:	Nimmt einen Parameter vom Typ `String`. Setzt ihn als Pfad zum ersten Streckenbild.

Tabelle 41: Instanzmethoden der Klasse *AAFVisualHintsDemoAgent*

METHODE	BESCHREIBUNG
image2	Liefert den Pfad zum zweiten gesetzten Streckenbild als `String`.
image2:	Nimmt einen Parameter vom Typ `String`. Setzt ihn als Pfad zum zweiten Streckenbild.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
setAllProps	Nimmt als Parameter ein `Dictionary`. Setzt alle Eigenschaften des Agenten.
x	Liefert die x-Verschiebung des Hinweises im Streckenbild als `Integer`-Objekt.
x:	Nimmt ein `Integer`-Objekt als Parameter. Setzt es als x-Verschiebung des Hinweises im Streckenbild.
y	Liefert die y-Verschiebung des Hinweises im Streckenbild als `Integer`-Objekt.
y:	Nimmt ein `Integer`-Objekt als Parameter. Setzt es als y-Verschiebung des Hinweises im Streckenbild.

Tabelle 41: Instanzmethoden der Klasse *AAFVisualHintsDemoAgent*

METHODE	BESCHREIBUNG
hintsDict	Liefert die verfügbaren Hinweisbilder als `Dictionary` mit Bezeichnungen als Schlüssel und Dateipfaden als value.
hintsList	Liefert die Bezeichnungen der verfügbaren Hinweisbilder als `OrderedCollection`.

Tabelle 42: Klassenmethoden der Klasse *AAFVisualHintsDemoAgent*

METHODE	BESCHREIBUNG
imagesDict	Liefert die verfügbaren Streckenbilder als `Dictionary` mit Bezeichnungen als Schlüssel und Dateipfaden als value.
imagesList	Liefert die Bezeichnungen der verfügbaren Streckenbilder als `OrderedCollection`.
plaintextName	Liefert die Klartextbezeichnung des Agenten als `String`.
readyForUse	Liefert ein `Boolean`-Objekt, welches anzeigt, ob der Agent bereits einsatzfähig ist.

Tabelle 42: Klassenmethoden der Klasse *AAFVisualHintsDemoAgent*

D.2.29 *AAFVisualHintsDemoAgentDialog*

METHODE	BESCHREIBUNG
addSpecialElements	Fügt die Bedienelemente zum Eigenschaften-Dialog hinzu, die einem `AAFBreaksAgent`-Objekt eigen sind.
update:	Nimmt ein Squeak-Ereignis als Parameter. Aktualisiert die Werte im Eigenschaften-Dialog, die einem `AAFVisualHintsDemoAgentDialog`-Objekt eigen sind.

Tabelle 43: Instanzmethoden der Klasse *AAFVisualHintsDemoAgentDialog*

D.2.30 *AAFXMLParser*

METHODE	BESCHREIBUNG
characters:	Nimmt einen `String` mit ungeparsten Daten und verarbeitet ihn.
endDocument	Aktion, die ausgeführt wird, wenn das gelesene Dokument beendet ist.
endElement:	Nimmt als Parameter einen XML-Elementnamen. Wird ausgeführt, wenn das angegebene Element beendet ist.
initialize	Initialisiert ein Objekt der Klasse. Wird bei der Erzeugung mit `new` automatisch aufgerufen.
nodeMap	Liefert die Liste der erzeugten Knoten als `Dictionary` zurück.
pop	Liefert das oberste Element vom Stack zurück und entfernt es vom Stack.
push:	Nimmt ein `Object` und setzt es als oberstes des Stacks in der Instanzvariable `stack`.
stack	Liefert den Stack vom Typ `OrderedCollection` zurück.
startDocument	Aktion, die ausgeführt wird, wenn das zu lesende Dokument beginnt.
startElement:attributeList:	Nimmt als Parameter einen XML-Elementnamen und eine XML-Attributliste. Wird ausgeführt, wenn das angegebene Element startet.
top	Liefert das oberste Element vom Stack zurück, ohne es vom Stack zu entfernen.

Tabelle 44: Instanzmethoden der Klasse *AAFXMLParser*

D.3 TRAITS

TRAIT	BESCHREIBUNG
`GetMainPanel`	Liefert eine Referenz auf das Haupt-Panel. Liefert `nil` für Morphe, die sich außerhalb des Haupt-Panel befinden.
`GetDrawArea`	Liefert eine Referenz auf die Arbeitsfläche. Liefert `nil` für Morphe, die sich außerhalb der Arbeitsfläche befinden.
`GetElement`	Liefert eine Referenz auf das Element, zu dem ein Morph gehört. Liefert `nil` für Morphe, die sich nicht in einem Element befinden.
`SetDirty`	Markiert die Existenz noch nicht gespeicherter Änderungen am Graphen.

Tabelle 45: Traits des Automatiken-GUI

E

TESTS

E.1 UNIT-TESTS

Unit-Test 1

Name:	`testConvert`
Beschreibung:	Die Klassenmethode `convert:type:` der Klasse `AAFUtils` wird getestet. Hierzu werden einige Konvertierungen durchgeführt und das Ergebnis auf korrekten Typ und Inhalt geprüft.

```
1  testConvert
2      | temp |
3
4      temp := AAFUtils convert: 'Hallo' type: 'String'.
5      self should: [temp isKindOf: String].
6      self should: [temp = 'Hallo'].
7
8      temp := AAFUtils convert: '123' type: 'String'.
9      self should: [temp isKindOf: String].
10     self should: [temp = '123'].
11     self shouldnt: [temp = 123].
12
13     temp := AAFUtils convert: '123' type: 'Number'.
14     self should: [temp isKindOf: Number].
15     self should: [temp isKindOf: Integer].
16     self should: [temp = 123].
17     self shouldnt: [temp isKindOf: Float].
18     self shouldnt: [temp = '123'].
19
20     temp := AAFUtils convert: '-45.0' type: 'Number'.
21     self should: [temp isKindOf: Number].
22     self should: [temp isKindOf: Float].
23     self should: [temp = -45.0].
24     self shouldnt: [temp isKindOf: Integer].
25     self shouldnt: [temp = '-45.0'].
26
27     temp := AAFUtils convert: 'true' type: 'Boolean'.
28     self should: [temp isKindOf: Boolean].
29     self should: [temp = true].
30     self shouldnt: [temp = 'true'].
31
32     temp := AAFUtils convert: 'false' type: 'Boolean'.
33     self should: [temp isKindOf: Boolean].
34     self should: [temp = false].
35     self shouldnt: [temp = 'false'].
36
37     temp := AAFUtils convert: 'Banane' type: 'Boolean'.
38     self should: [temp isKindOf: String].
39     self should: [temp ='Banane'].
40     self shouldnt: [temp isKindOf: Boolean].
```

```
41      self shouldnt: [temp = true].
42      self shouldnt: [temp = false].
```

Unit-Test 2

Name: `testReadFromFile`

Beschreibung: Ein Graph wird aufgebaut, in eine Datei geschrieben und aus dieser wieder eingelesen. Anschließend wird überprüft, ob der Graph korrekt eingelesen wurde, indem Knoten, Kanten und `AAFDelegate`-Objekte geprüft werden. Dieser Testfall testet implizit mit, ob der Graph korrekt und vollständig in die XML-Datei geschrieben wurde, da sonst nach dem Einlesen der Graph in jedem Fall fehlerhaft sein müsste.

```
1  testReadFromFile
2      | g fs writer temp |
3
4      "generate graph"
5      g := self class generateGraph1.
6
7      "write graph to file"
8      fs := FileStream forceNewFileNamed: 'tests\
       testReadFromFile.aaf'.
9      writer := XMLWriter on: fs.
10     g printXMLOn: writer.
11     fs close.
12
13     "read graph from file"
14     g := nil.
15     fs := FileStream fileNamed: 'tests\testReadFromFile.aaf'.
16     g := (AAFXMLParser parseDocumentFrom: fs) document.
17     fs close.
18
19     "check graph for correctness"
20
21     "type of graph and number of nodes are ok"
22     self should: [g isKindOf: AAFGraph].
23     self should: [g nodes size = 3].
24
25     "types of nodes in node collection are ok"
26     self should: [(g nodes at: 1) class = AAFMuxNode].
27     self should: [(g nodes at: 2) class = AAFNode].
28     self should: [(g nodes at: 3) class = AAFComplexNode].
29
30     "types of root and sink are ok"
31     self should: [g root class = AAFMuxNode].
32     self should: [g sink class = AAFDemuxNode].
33
34     "connections of root are ok"
35     self should: [g root parents isEmpty].
36     self should: [g root children size = 2].
37     self should: [g root children contains: [:each | each
       class = AAFNode]].
```

```
38      self should: [g root children contains: [:each | each
        class = AAFComplexNode]].
39
40      "connections of sink are ok"
41      self should: [g sink parents size = 2].
42      self should: [g sink children isEmpty].
43      self should: [g sink parents contains: [:each | each
        class = AAFNode]].
44      self should: [g sink parents contains: [:each | each
        class = AAFComplexNode]].
45
46      "AAFNode in collection is ok"
47      temp := ((g nodes) select: [:each | each class = AAFNode
        ]) at: 1.
48      self should: [temp parents size = 1].
49      self should: [(temp parents at: 1) class = AAFMuxNode].
50      self should: [temp children size = 1].
51      self should: [(temp children at: 1) class = AAFDemuxNode
        ].
52      self should: [temp position = (220@450)].
53      self should: [temp delegate class = AAFBreaksAgent].
54      self should: [temp delegate isPointerVisible = false].
55
56      "connections of AAFComplexNode are ok"
57      temp := ((g nodes) select: [:each | each class =
        AAFComplexNode]) at: 1.
58      self should: [temp parents size = 1].
59      self should: [(temp parents at: 1) class = AAFMuxNode].
60      self should: [temp children size = 1].
61      self should: [(temp children at: 1) class = AAFDemuxNode
        ].
62      self should: [temp position = (700@450)].
63      self should: [temp delegate class = AAFGraph].
64
65      "check inner graph"
66      temp := temp delegate.
67      self should: [temp isKindOf: AAFGraph].
68      self should: [temp nodes size = 2].
69      self should: [(temp nodes at: 1) class = AAFMuxNode].
70      self should: [(temp nodes at: 2) class = AAFNode].
71      self should: [temp root class = AAFMuxNode].
72      self should: [temp sink class = AAFDemuxNode].
73      self should: [temp root parents isEmpty].
74      self should: [temp root children size = 1].
75      self should: [(temp root children at: 1) class = AAFNode
        ].
76      self should: [temp sink parents size = 1].
77      self should: [temp sink children isEmpty].
78      self should: [(temp sink parents at: 1) class = AAFNode].
79      "check AAFNode in inner graph"
80      temp := temp nodes at: 2.
81      self should: [temp parents size = 1].
82      self should: [(temp parents at: 1) class = AAFMuxNode].
83      self should: [temp children size = 1].
84      self should: [(temp children at: 1) class = AAFDemuxNode
        ].
85      self should: [temp position = (220@450)].
86      self should: [temp delegate class = AAFBreaksAgent].
87      self should: [temp delegate isPointerVisible = true].
```

E.2 PRÜFLISTE FÜR MANUELLE TESTS

Vorbedingung:	VB
Test:	T
Nachbedingung:	NB

TESTFALL 1	
VB	• Es befinden sich außer Wurzel- und Senkenelement keine Elemente im Arbeitsbereich. • Der zugehörige AAF-Graph besteht nur aus Wurzel- und Senkenknoten. • Wurzel- und Senkenknoten haben keine Verbindungen.
T	• Ein normales Element wird von links aus dem Elementbereich in die Arbeitsfläche gezogen. • Das Element wird mit Wurzel- und Senkenelement verbunden.
NB	• Das Element befindet sich im Arbeitsbereich. • Das Element ist mit Wurzel- und Senkenelement verbunden. • Der zuhörige AAF-Graph besteht aus Wurzel- und Senkenknoten und einem normalen Knoten. • Der Typ des Agenten im normalen Knoten des AAF-Graphen passt zum Element. • Im AAF-Graphen hat der Wurzelknoten den normalen Knoten als Kindknoten. • Im AAF-Graphen hat der Senkenknoten den normalen Knoten als Elternknoten. • Im AAF-Graphen hat der normale Knoten den Wurzelknoten als Elternknoten und den Senkenknoten als Kindknoten.

TESTFALL 2	
VB	–
T	• Ein Element wird aus dem Auswahlbereich für einfache Elemente in den Auswahlbereich für komplexe Elemente gezogen.
NB	• Das Element wurde dem Auswahlbereich für komplexe Elemente nicht hinzugefügt. • Das Element wurde verworfen und existiert nicht mehr.

TESTFALL 3	
VB	• NB aus Testfall 1.
T	• Das Element wird in der Arbeitsfläche verschoben.
NB	• Das Element wird an der neuen Position angezeigt. • Die Koordinaten des zugehörigen `AAF`-Knotens haben sich angepasst.

TESTFALL 4	
VB	• NB aus Testfall 1.
T	• Das Element wird aus dem Graphen gelöscht.
NB	• Das Element ist aus dem Graphen gelöscht. • Die Verbindungen zu Wurzel- und Senkenelement sind aus dem Graphen gelöscht. • Der zuhörige AAF-Graph besteht nur aus Wurzel- und Senkenknoten. • Wurzel- und Senkenknoten haben keine Verbindungen.

TESTFALL 5	
VB	• NB aus Testfall 1. • Es existiert keine gespeicherte Automatik mit dem Namen *test.aaf*. • Es existiert kein komplexes Element mit der Aufschrift *test.aaf* im Auswahlbereich für komplexe Elemente.
T	• Abspeichern als *test.aaf*.
NB	• Die Datei *test.aaf* wurde im Standardspeicherverzeichnis geschrieben. • Die Datei *test.aaf* enthält den zum Graphen passenden korrekten XML-Code. • Im Auswahlbereich für komplexe Elemente wird ein neues Element mit der Aufschrift *test.aaf* angezeigt.

TESTFALL 6	
VB	• Es befinden sich außer Wurzel- und Senkenelement keine Elemente im Arbeitsbereich. • Der zugehörige AAF-Graph besteht nur aus Wurzel- und Senkenknoten. • Wurzel- und Senkenknoten haben keine Verbindungen. • Es existieren keine ungespeicherten Veränderungen.
T	• Öffnen von *test.aaf*.
NB	• NB wie Testfall 1. • Das einfache Element trägt die Aufschrift, die es beim Abspeichern trug. • Diese Aufschrift ist auf AAF-Ebene gesetzt.

TESTFALL 7	
VB	• NB aus Testfall 1.
T	• Klick auf einen der Verbindungspunkte am normalen Element.
NB	• Der Status des Verbindungspunkt ist *nicht aktiviert*. • Der Verbindungspunkt ist nicht rot.

TESTFALL 8	
VB	• Es befinden sich außer Wurzel- und Senkenelement keine Elemente im Arbeitsbereich. • Der zugehörige AAF-Graph besteht nur aus Wurzel- und Senkenknoten. • Wurzel- und Senkenknoten haben keine Verbindungen.
T	• Zwei Elemente A und B aus den Auswahlbereichen in die Arbeitsfläche ziehen. • Ausgang von A anklicken und so aktivieren. • Ausgang von B anklicken.
NB	• Der Ausgang von A ist rot. • Der Ausgang von B ist grau. • Der Status des Ausgangsverbindungspunktes von A ist *aktiviert*. • Der Status des Ausgangsverbindungspunktes von B ist *nicht aktiviert*. • Zwischen den beiden Ausgängen wird keine Verbindung angezeigt. • Zwischen den zu A und B gehörenden AAF-Knoten besteht keine Verbindung.

TESTFALL 9	
VB	• Es befinden sich außer Wurzel- und Senkenelement keine Elemente im Arbeitsbereich. • Der zugehörige AAF-Graph besteht nur aus Wurzel- und Senkenknoten. • Wurzel- und Senkenknoten haben keine Verbindungen.
T	• Zwei Elemente A und B aus den Auswahlbereichen in die Arbeitsfläche ziehen. • Eingang von A anklicken und so aktivieren. • Eingang von B anklicken.
NB	• Der Eingang von A ist rot. • Der Eingang von B ist grau. • Der Status des Eingangsverbindungspunktes von A ist *aktiviert*. • Der Status des Eingangsverbindungspunktes von B ist *nicht aktiviert*. • Zwischen den beiden Eingängen wird keine Verbindung angezeigt. • Zwischen den zu A und B gehörenden AAF-Knoten besteht keine Verbindung.

TESTFALL 10	
VB	• Es befinden sich außer Wurzel- und Senkenelement keine Elemente im Arbeitsbereich. • Der zugehörige AAF-Graph besteht nur aus Wurzel- und Senkenknoten. • Wurzel- und Senkenknoten haben keine Verbindungen.
T	• Ein Element aus einem der Auswahlbereiche in die Arbeitsfläche ziehen. • Eingang des Elements anklicken und so aktivieren. • Ausgang desselben Elements anklicken.
NB	• Der Eingang ist rot. • Der Ausgang ist grau. • Der Status des Eingangsverbindungspunktes ist *aktiviert*. • Der Status des Ausgangsverbindungspunktes ist *nicht aktiviert*. • Zwischen Ein- und Ausgang wird keine Verbindung angezeigt. • Der zugehörige AAF-Knoten ist nicht mit sich selbst verbunden.

TESTFALL 11	
VB	• NB aus Testfall 1.
T	• Element doppelklicken. • Neuen Namen vergeben.
NB	• Für das Element wird der neue Name angezeigt. • Der neue Name wurde als Label auf AAF-Ebene gesetzt.

TESTFALL 12

VB	• Es befinden sich außer Wurzel- und Senkenelement keine Elemente im Arbeitsbereich. • Der zugehörige AAF-Graph besteht nur aus Wurzel- und Senkenknoten. • Wurzel- und Senkenknoten haben keine Verbindungen.
T	• Zwei Elemente A und B aus den Auswahlbereichen in den Arbeitsbereich ziehen. • Eingang von A anklicken. • Ausgang von B anklicken. • Ausgang von A anklicken. • Eingang von B anklicken.
NB	• Eingang und Ausgang von B werden grau angezeigt. • Die entsprechenden Verbindungspunkte haben den Status *nicht aktiviert*. • Eingang von A wird grau angezeigt. • Der entsprechende Verbindungspunkt hat den Status *nicht aktiviert*. • Ausgang von A wird rot angezeigt. • Der entsprechende Verbindungspunkt hat den Status *aktiviert*. • Zwischen dem Eingang von A und dem Ausgang von B wird eine Verbindung angezeigt. • Zwischen dem Ausgang von A und dem Eingang von B wird keine Verbindung angezeigt. • Die zu A und B gehörenden AAF-Knoten haben nur eine Verbindung, bei dieser ist A Kind- und B Elternknoten.

TESTFALL 13

VB	• NB aus Testfall 1.
T	• Element in der Arbeitsfläche verschieben. • *Neu*-Schaltfläche anklicken.
NB	• Der Hinweis auf ungespeicherte Änderungen wird angezeigt.

TESTFALL 14	
VB	• NB aus Testfall 1.
T	• Element in der Arbeitsfläche verschieben. • *Öffnen*-Schaltfläche anklicken.
NB	• Der Hinweis auf ungespeicherte Änderungen wird angezeigt.

TESTFALL 15	
VB	• NB aus Testfall 1.
T	• Element in der Arbeitsfläche verschieben. • *Verlassen*-Schaltfläche anklicken.
NB	• Der Hinweis auf ungespeicherte Änderungen wird angezeigt.

TESTFALL 16	
VB	• Es existiert ein komplexes Element.
T	• Das komplexe Element in die Arbeitsfläche ziehen. • Das komplexe Element mit der Maus anklicken und es so markieren. • Im Eigenschaftenbereich den Button mit Aufschrift *Inneren Graphen anzeigen* anklicken.
NB	• Der innere Graph wird angezeigt.

TESTFALL 17	
VB	• Es ist eine Automatik geladen, die mindestens ein komplexes Element enthält. • Es wird der Graph aus einem komplexen Element angezeigt.
T	• Abspeichern als *test2.aaf*.
NB	• Der abgespeicherte Graph ist der auf oberster Ebene.

TESTFALL 18	
VB	• Es befinden sich mindestens 2 Elemente A und B im Arbeitsbereich. • Element A ist für Kurven aktiviert. • Element B ist für Kurven und Gabelungen aktiviert.
T	• Beide Elemente markieren.
NB	• Im Eigenschaftenbereich wird für Kurve die Farbe für *aktiviert* und für Gabelung die Farbe für *gemischter Zustand* angezeigt. Die übrigen Listeneinträge sind als *inaktiviert* eingefärbt.

TESTFALL 19	
VB	• NB aus vorigem Testfall.
T	• In der Auswahlliste auf Gabelung klicken. • In der Auswahlliste auf Kurve klicken.
NB	• Für beide Elemente wird für Kurve *inaktiv* angezeigt. • Für beide Elemente wird für Gabelung *aktiv* angezeigt. • Diese Zustände sind in den zugehörigen Feldern auf `AAF`-Ebene korrekt abgebildet.

F

INTEGRIEREN NEUER AGENTEN IN DAS AUTOMATIKEN-GUI

In dieser Anleitung werden die einzelnen Schritte beschrieben, die durchgeführt werden müssen, um einen neuen AAF-Agenten im Automatiken-GUI als Element zur Verfügung zu stellen.

F.1 ERSTELLEN EINES PASSENDEN AGENTEN

Um einen AAF-Agenten im Automatiken-GUI integrieren zu können, muss er zunächst erstellt werden. Hierbei sind einige Bedingungen zu erfüllen, damit der fertig implementierte Agent ins Automatiken-GUI eingebunden und später in einer Automatik verwendet werden kann.

Ableiten von `AAFAgent`

Der Agent muss eine Ableitung von `AAFAgent` sein. Dabei muss die Ableitung nicht direkt sein, es ist ebenso möglich, über eine oder mehrere Zwischenklassen von `AAFAgent` zu erben.

```
1 AAFAgent subclass: #AAFBreaksAgent
2     instanceVariableNames: ''
3     classVariableNames: ''
4     poolDictionaries: ''
5     category: 'AAF-Agents'
```

Klartextname des Agenten

Damit das Automatiken-GUI das Element mit einer passenden Aufschrift versehen kann, muss der Klartextname des Agenten von der Klassenmethode `plaintextName` als Zeichenkette zurückgeliefert werden.

```
1 plaintextName
2     ↑'Breaks'.
```

Einsatzbereitschaft anzeigen

Ist der Agent fertig implementiert und prinzipiell als Teil einer Automatik einsatzfähig, so muss dies angezeigt werden. Für diesen Zweck haben AAF-Agenten eine Klassenmethode `readyForUse`, welche einen booleschen Wert zurückgibt. Die geerbte Standardimplementierung liefert den Wert `false`. Soll der

Agent für die Verwendung freigegeben werden, so muss der Programmierer diese Methode überschreiben und den Wert `true` zurückgeben, welcher Einsatzbereitschaft anzeigt.

```
readyForUse
    ↑true.
```

Für Details zu den Anforderungen, die ein AAF-Agent erfüllen muss, um als Teil eines AAF-Graphen einsetzbar zu sein, sei hier auf die Dokumentation des AAF verwiesen.

Sind die beiden bis hierher genannten Voraussetzungen erfüllt, so wird im Automatiken-GUI bereits ein Element für den Agenten erstellt und im Elementbereich bereitgestellt.

Besitzt der Agent keinerlei Eigenschaften, die über die geerbten hinausgehen, so ist die Integration in das Automatiken-GUI an dieser Stelle abgeschlossen und das erstellte Element kann verwendet werden. Besitzt er weitere Eigenschaften, so sind noch weitere Schritte nötig, damit diese gespeichert, geladen und angepasst werden können.

Abfrage und Setzen spezieller Eigenschaften

Besitzt der Agent Eigenschaften, die über die geerbten hinausgehen, so muss er die Instanzmethoden `getAllProps` und `setAllProps` erweitern. Diese bilden die Schnittstelle zum Auslesen und Setzen aller Eigenschaften, ohne die einzelnen Zugriffsmethoden kennen zu müssen. Benötigt wird diese Schnittstelle z. B. zum Speichern und Wiederherstellen von Automatiken.

`getAllProps` muss ein `Dictionary` zurückgeben, welches die Eigenschaften und ihre aktuellen Werte als Schlüssel-Wert-Paare enthält. Hierbei ist zu beachten, dass gleich zu Beginn die entsprechende Methode der Basisklasse aufgerufen werden muss, sodass auch die Eigenschaften enthalten sind, die geerbt wurden.

```
getAllProps
    | dict |
    dict := super getAllProps.

    (dict at: 'pointerVisible' put: (self isPointerVisible)).

    ↑dict.
```

Listing 14: `getAllProps` gibt alle Eigenschaften und ihre Werte als `Dictionary` mit Schlüssel-Wert-Paaren zurück.

`setAllProps` erhält ein `Dictionary` als Parameter. Dieses enthält alle Eigenschaften des Agenten sowie ihre zu setzenden Werte als Schlüssel-Wert-Paare. Die Methode `setAllProps` ist

```
AAFDelegate>>getAllProps
    | props dict |

    dict := Dictionary new.
    props := OrderedCollection new.

    (inactiveSelection isKindOf: Array)
        ifTrue: [
            (1 to: inactiveSelection size)
                do: [:c | props add: (inactiveSelection at: c
    )]
        ].

    dict at: 'inactiveValues' put: props.
    ↑dict.
```

Listing 15: Der Quelltext von `AAFDelegate»getAllProps` zeigt, dass es sich beim Rückgabewert um ein `Dictionary` handelt.

nun dafür zuständig, die Werte über die entsprechenden Zugriffsmethoden den Eigenschaften zuzuweisen. Auch hier ist zu beachten, dass ein Teil der Werte durch die `setAllProps`-Methode der Basisklasse verarbeitet werden muss, um die geerbten Eigenschaften zu setzen.

```
setAllProps: aPropertyDictionary

    (aPropertyDictionary size >= 2)
    ifTrue: [
        super setAllProps: aPropertyDictionary.
        self pointerVisible: (AAFUtils convert: (
    aPropertyDictionary at: 'pointerVisible') type: 'Boolean'
    ).
    ].
```

Listing 16: Die Methode `setAllProps` nimmt alle Eigenschaftenwerte als `Dictionary` und ordnet sie den einzelnen Eigenschaften zu.

An dieser Stelle ist der Agent so weit, dass er tatsächlich im Automatiken-GUI verwendet werden kann, auch Speichern und Laden sind möglich. Allerdings kann er bis hierher nur mit seinen Standardeigenschaften verwendet werden, welche sinnvoll vorbelegt sein sollten, da es noch keine Möglichkeit zum Editieren gibt.

F.2 ERSTELLEN DES EIGENSCHAFTEN-DIALOGS

Damit die Eigenschaften des Agenten angezeigt und editiert werden können, ist es notwendig, für den Agenten einen passenden Dialog zu erstellen. Die hierfür durchzuführenden Schritte werden im Folgenden erläutert.

Ableiten von `AAFAgentDialog`

Der Eigenschaften-Dialog für den Agenten muss eine Ableitung von `AAFAgentDialog` sein. Auch hier spielt es wieder keine Rolle, ob die Ableitung direkt oder über Zwischenklassen erfolgt.

```
AAFAgentDialog subclass: #AAFBreaksAgentDialog
    instanceVariableNames: 'pointerVisibleButton'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'AAFGT-Dialog'
```

Listing 17: Dialoge müssen Ableitungen von `AAFAgentDialog` sein.

Benennung der Dialogklasse

Damit die Dialogklasse vom Automatiken-GUI gefunden und verwendet werden kann, muss sie den gleichen Namen tragen wie der zugehörige Agent, erweitert durch die Zeichenkette „`Dialog`“. Heißt also die Agenten-Klasse beispielsweise `AAFBreaksAgent`, so heißt die zugehörige Dialogklasse `AAFBreaksAgentDialog`.

Hinzufügen der Bedienelemente für die speziellen Eigenschaften

Um Eigenschaften anzeigen und editieren zu können, die der Agent über die geerbten hinaus besitzt, muss die Methode `addSpecialElements` entsprechend erweitert werden. In dieser Klasse werden die entsprechenden Bedienelemente erzeugt und anschließend dem dafür vorgesehenen Bereich `buttonArea` zugefügt. Zuletzt werden die angezeigten Werte aktualisiert, indem sie vom Agenten abgefragt werden.

```
addSpecialElements
    pointerVisibleButton := AAFWidgetUtils
                                toggleButtonNew: delegate
                                ...

    ...

    self buttonArea
        addMorph: pointerVisibleButton
        ...

    self updateFromDelegate.
```

Listing 18: Um spezielle Eigenschaften eines Dialogs anzeigen zu können, muss die Methode `addSpecialElements` implementiert werden.

Die Erstellung der Bedienelemente kann nicht allgemein beschrieben werden, da sie stark davon abhängig ist, um welchen Datentyp es sich handelt und welche Art von Bedienelement zum Einsatz kommen soll. Häufig müssen sogar die Bedienelemente selbst erst implementiert werden, da Squeak keine passenden zur Verfügung stellt. In jedem Fall muss das Bedienelement jedoch bei einer Eingabe die Werte direkt im Agenten ändern. Außerdem ist zu beachten, dass das Bedienelement sich als Beobachter beim Agenten registrieren muss, da es nur so die angezeigten Werte aktualisiert, nachdem es sie verändert hat oder sie von anderer Stelle aus verändert wurden[43].

Aktualisieren der Anzeige bei Änderungen am Agenten

Damit der Dialog sich im Falle von Änderungen am Agenten korrekt aktualisiert, muss nun noch die Methode `updateFromDelegate` implementiert werden. Hierfür müssen die aktuellen Werte vom Agenten abgefragt und den entsprechenden Anzeigeelementen zugewiesen werden. Wichtig ist auch hier wieder der Aufruf der `updateFromDelegate`-Methode der Basisklasse, da nur so die Aktualisierung der Eigenschaften erfolgen kann, die der Agent geerbt hat.

```
1  updateFromDelegate
2      super updateFromDelegate.
3
4      ...
5
6      (delegate isPointerVisible)
7          ifTrue: [pointerVisibleButton  turnOn.]
8          ifFalse: [pointerVisibleButton turnOff.].
9
10     ...
```

Listing 19: Damit sich die Anzeige der Eigenschaftenwerte bei Änderungen am Agenten korrekt aktualisiert, muss die Methode `updateFromDelegate` implementiert werden.

Nun ist der Agent vollständig einsatzbereit und kann nicht nur gespeichert und geladen, sondern über den Dialog auch in seinen Eigenschaften angepasst werden.

Hinweis: im Automatiken-GUI-Quelltext sind in der Kategorie `AAFGT-Dialog` bereits einige Dialoge für Agenten implementiert. Diese können als Beispiele für konkrete Implementierungen herangezogen werden.

43 Stichwort *Model-View-Controller*: der Agent stellt das Model dar, View und Controller sind in dem Morph vereint, der das Bedienelement implementiert. Änderungen bei Nutzereingaben werden nur am Model vorgenommen, die Änderungen im View resultieren aus einer Aktualisierung vom Model her.

G

BEDIENUNGSANLEITUNG

G.1 STARTEN DES AUTOMATIKEN-GUI

Das Automatiken-GUI wird gestartet, indem in einem Workspace-Fenster in Squeak Folgendes ausgeführt wird:

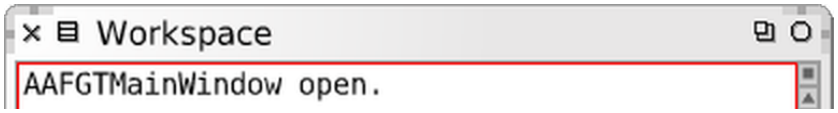


Abbildung 31: Starten des GUI

G.2 AUFBAU EINER AUTOMATIK

G.2.1 *Elemente hinzufügen*

Elemente werden zur Automatik hinzugefügt, indem sie aus einem der beiden Elementbereiche auf der linken Seite in die Arbeitsfläche im rechten Bereich der Anwendung gezogen und dort fallen gelassen werden (Abbildung 32).

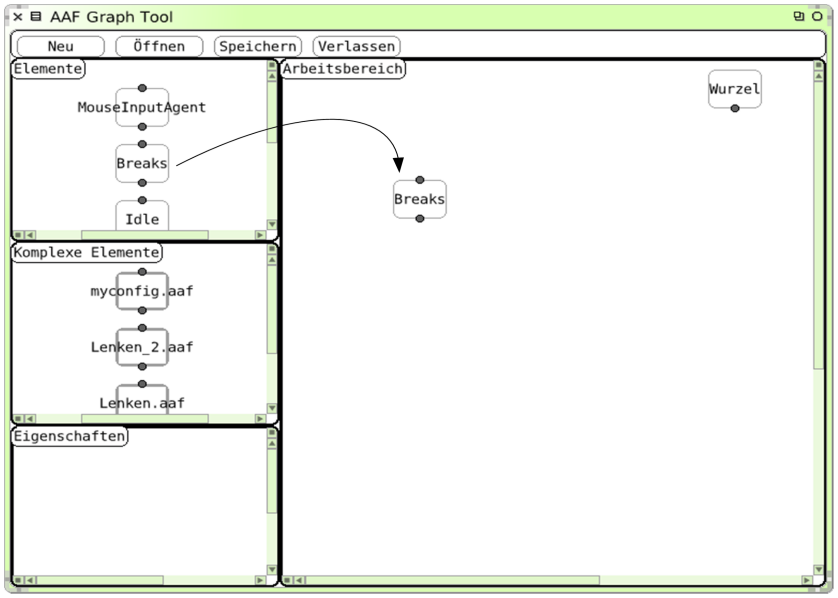


Abbildung 32: Zum Hinzufügen eines Elementes zur Automatik wird es aus einem der Elementbereiche auf die Arbeitsfläche gezogen.

G.2.2 *Verbinden von Elementen*

Zum Verbinden von Elementen wird folgendermaßen vorgegangen:

1. Verbindungspunkt an dem einen der zu verbindenden Elemente anklicken. Er ist nun aktiviert und erscheint rot (s. Abbildung 33).

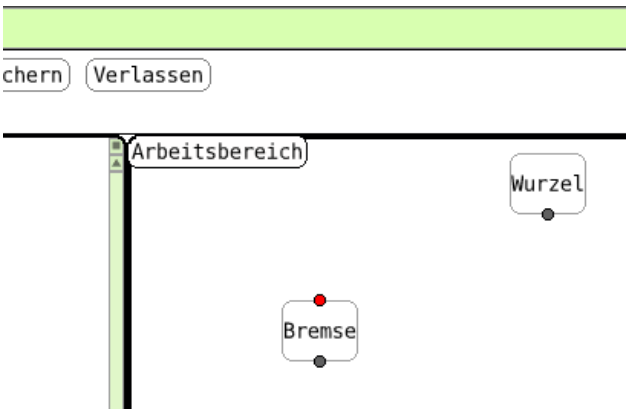


Abbildung 33: Der erste Verbindungspunkt ist aktiviert.

2. Anschließend den Verbindungspunkt anklicken, mit dem die Verbindung hergestellt werden soll. Die Verbindungslinie wird gezogen (s. Abbildung 34).

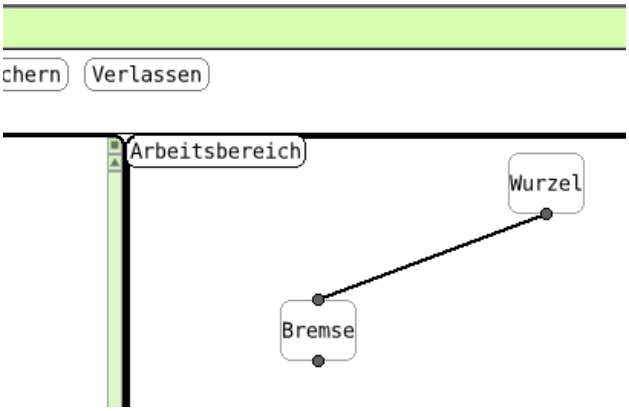


Abbildung 34: Die Verbindung zwischen zwei Elementen wurde hergestellt.

Soll ein bereits aktivierter und rot angezeigter Verbindungspunkt doch nicht verbunden werden, so wird er durch erneutes Anklicken deaktiviert.

Beim Ziehen von Verbindungen ist zu beachten, dass nur jeweils ein Eingang mit einem Ausgang verbunden werden kann. Dabei ist es gleichgültig, ob die Verbindung vom Eingang zum

Ausgang oder umgekehrt gesetzt wird. Eingang und Ausgang desselben Elementes können nicht verbunden werden.

Jedes Element mit Ausnahme von Wurzel und Senke kann nur eine eingehende und eine ausgehende Verbindung herstellen.

Die Wurzel kann beliebig viele ausgehende, die Senke beliebig viele eingehende Verbindungen herstellen.

G.2.3 *Benennung eines Elementes ändern*

Um den Namen eines Elementes zu ändern, wird ein Doppelklick auf das Element ausgeführt. In das erscheinende Texteingabefeld kann nun ein neuer Name eingegeben werden (s. Abbildung 35).

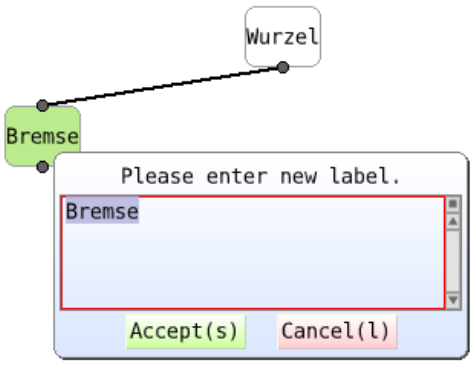


Abbildung 35: Nach einem Doppelklick auf ein Element kann ein neuer Name vergeben werden.

G.2.4 *Löschen einer Verbindung*

Um eine existierende Verbindung zwischen zwei Elementen wieder zu löschen, wird die Verbindung mit der rechten Maustaste angeklickt. Im erscheinenden Kontextmenü wird nun der Punkt `remove line` ausgewählt (s. Abbildung 36).

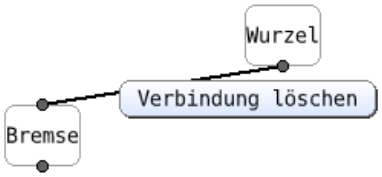


Abbildung 36: Über das Kontextmenü können Verbindungen wieder gelöscht werden.

G.2.5 *Löschen eines Elements*

Soll ein Element aus der Automatik entfernt werden, so wird es mit der rechten Maustaste angeklickt. Im nun erscheinenden Kontextmenü wird der Punkt `remove` gewählt (s. Abbildung 37). Ist das Element mit anderen verbunden, so werden die Verbindungen ebenfalls gelöscht.

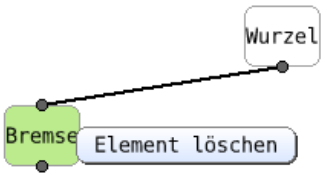


Abbildung 37: Über das Kontextmenü können Elemente wieder gelöscht werden.

G.2.6 *Bearbeiten von Elementeigenschaften*

Um Eigenschaften eines Elementes zu bearbeiten, wird das Element per Mausklick markiert. Die zugehörigen Eigenschaften erscheinen nun im Eigenschaftenbereich unten auf der linken Seite und können bearbeitet werden (s. Abbildung 38).

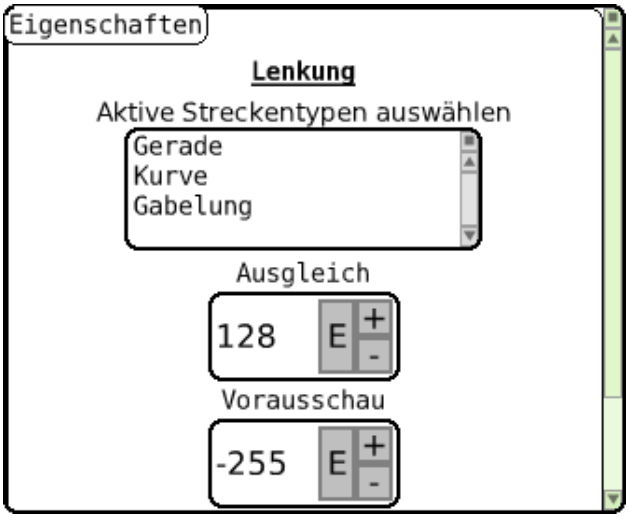


Abbildung 38: Im Eigenschaftenbereich können die Eigenschaften eines markierten Elements bearbeitet werden.

G.2.7 *Bearbeiten der (In-)Aktivitätseigenschaft für mehrere Elemente*

Eine besondere Eigenschaft bildet die Liste der Streckenabschnittstypen, für die ein Element aktiv sein soll. Diese Eigen-

schaft ist allen Elementen gemein und kann daher für mehrere gleichzeitig bearbeitet werden.

Um mehrere Elemente zu markieren, werden sie nacheinander bei gedrückter [Alt]-Taste mit der rechten Maustaste[44] angeklickt. Ein solcher Klick auf ein bereits markiertes Element entfernt die Markierung wieder.

Ist ein Streckenabschnittstyp für alle gerade markierten Elemente aktiviert, so ist er grün unterlegt. Abschnittstypen, die für alle nicht aktiviert sind, erscheinen vor einem weißen Hintergrund. Ist ein Abschnittstyp für einige der markierten Elemente aktiviert und für andere nicht, so erscheint er in einem helleren Grünton (s. Abbildung 39).

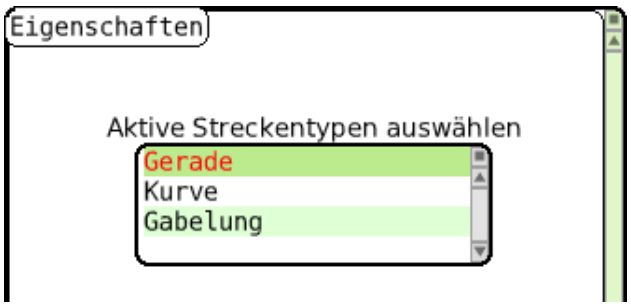


Abbildung 39: Die Streckenabschnittstypen, für die ein Element aktiv sein soll, können für mehrere Elemente zugleich angepasst werden.

G.2.8 *In einen Untergraphen verzweigen*

Ist ein Element angeklickt, welches seine Funktion über einen Untergraphen realisiert, so wird im Eigenschaftenbereich eine Schaltfläche angezeigt, über die in den Untergraphen verzweigt werden kann (s. Abbildung 40). Dieser erscheint dann anstelle des bis dahin sichtbaren Graphen im Arbeitsbereich und kann auf die gleiche Weise bearbeitet werden. Elemente, die einen Untergraphen erhalten, sind am dickeren Rahmen zu erkennen.

Soll die Bearbeitung des Untergraphen beendet werden, so schließt ihn die Schaltfläche mit der Aufschrift `Inneren Graphen schließen` und stellt den vorherigen Graphen wieder her.

Achtung: Wird ein Graph gespeichert, während ein Untergraph in der Arbeitsfläche angezeigt wird, so wird dennoch der Graph auf oberster Ebene gespeichert, zu dem der Untergraph gehört.

44 Diese Bedienung ist leider etwas unschön, da sie von den Konventionen anderer Programme abweicht. Da Squeak jedoch die übliche Kombination von [Strg]-Taste und Klick selbst abfängt und verarbeitet, musste auf diese Alternative ausgewichen werden.

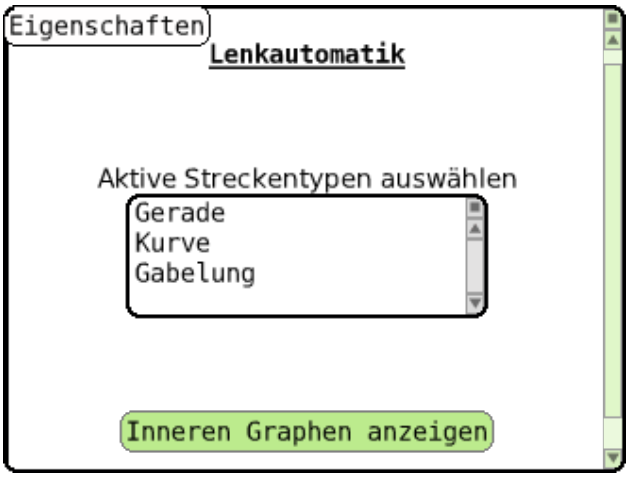


Abbildung 40: Die Automatikfunktion eines Elements kann durch einen Untergraphen realisiert sein. Dieser lässt sich anzeigen und bearbeiten.

G.3 EINBINDEN BESTEHENDER AUTOMATIKEN ALS ELEMENTE

Bereits bestehende Automatiken können als Elemente neuer Automatiken verwendet werden. Dazu stellt das Automatiken-GUI alle Automatiken, die im Standardverzeichnis für gespeicherte Automatiken liegen und lauffähig sind, im Auswahlbereich für komplexe Elemente zur Verfügung (s. Abbildung 41). Sie können genauso wie normale Elemente auch verwendet werden.

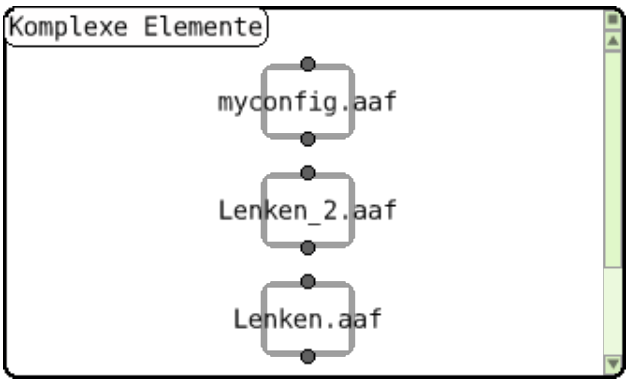


Abbildung 41: Existierende Automatiken stehen als komplexe Elemente zur Verwendung in neuen Automatiken zur Verfügung.

G.4 SPEICHERN EINER AUTOMATIK

Um eine Automatik in eine Datei abzuspeichern, wird in der Menüleiste (s. Abbildung 42) die Schaltfläche `Speichern` ausge-

wählt. Anschließend erscheint ein Texteingabe-Dialog, über den der Name der Datei angegeben wird. Dabei ist zu beachten, dass die Speicherung automatisch in das Standardverzeichnis für Automatiken-Dateien erfolgt, es ist kein Verzeichnis auszuwählen. Existiert der gewählte Dateiname bereits, so erhält der Benutzer eine entsprechende Warnung und hat die Möglichkeit, einen anderen Namen anzugeben.

Abbildung 42: Über die Menüleiste sind die Funktionen `Neu`, `Laden`, `Speichern` und `Verlassen` zugänglich.

G.5 LADEN EINER AUTOMATIK

Um eine früher in eine Datei abgespeicherte Automatik zur erneuten Bearbeitung zu laden, wird die Schaltfläche `Öffnen` in der Menüleiste (Abbildung 42) ausgewählt. Es erscheint ein Auswahldialog, der alle im Standardverzeichnis vorhandenen Automatik zur Auswahl stellt (s. Abbildung 43).

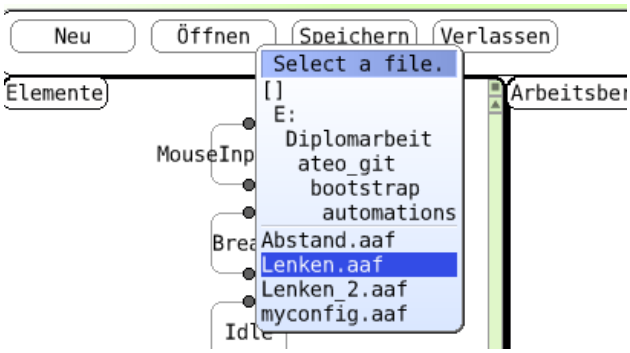


Abbildung 43: Beim Laden von Automatiken werden die verfügbaren zur Auswahl gestellt.

G.6 NEUE AUTOMATIK BEGINNEN

Soll eine neue Automatik begonnen werden, so ist die Schaltfläche `Neu` in der Menüleiste (Abbildung 42) zu verwenden. Der Arbeitsbereich wird geleert und die Bearbeitung kann von Neuem beginnen. Befinden sich in der aktuellen Automatik noch ungesicherte Änderungen, so erhält der Benutzer beim Klick auf `Neu` eine entsprechende Warnung sowie die Möglichkeit, sie noch abzuspeichern.

G.7 VERLASSEN DES AUTOMATIKEN-GUI

Das Automatiken-GUI kann alternativ über die `Verlassen`-Schaltfläche in der Menüleiste (Abbildung 42) oder das Kreuz in der Titelleiste verlassen werden. In beiden Fällen erhält der Nutzer eine Warnung, falls noch nicht gespeicherte Änderungen vorliegen, sowie die Möglichkeit, sie noch abzuspeichern.

H

INHALT DER DVD

Gemeinsam mit dieser Diplomarbeit wurde eine DVD eingereicht, auf der sich diese Arbeit in elektronischer Fassung befindet. Dort sind auch ein lauffähiges Squeak-Image mit dem Automatiken-GUI sowie der während der Arbeit entstandene Quellcode zu finden.

Die Daten auf der DVD befinden sich auf dem zum Abgabezeitpunkt aktuellen Stand. Da das Projekt fortgeführt wird, können bereits neuere Versionen des Quellcodes und des Squeak-Images vorliegen. Diese liegen in einem Git-Repository und können über http://www.assembla.com/spaces/ATEO/ bezogen werden.

INHALT DER DVD

Readme.txt	Beschreibung des DVD-Inhalts.
Diplomarbeit.pdf	Eine elektronische Fassung der Diplomarbeit.
Squeak/	In diesem Verzeichnis befindet sich ein Squeak-Image, in dem das Automatiken-GUI ausgeführt werden kann. Im Verzeichnis die Datei `Squeak.exe` ausführen, dann den Ausdruck `AAFGTMainWindow open` im bereits offenen Workspace mit [Strg+D] evaluieren.
Code/	In diesem Verzeichnis befindet sich der im Rahmen der Arbeit entstandene Squeak-Quellcode. Die Dateien sind durch Anwendung der im Projekt entwickelten `ATEOFileOut`-Klasse exportiert worden und können in ein Squeak-Image importiert werden.

LITERATURVERZEICHNIS

[Balzert 2005] Balzert, Heide: *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. 2. Auflage. Heidelberg : Elsevier Spektrum Akad. Verl., 2005 (Lehrbücher der Informatik). – ISBN 3827411629

[Balzert 2001] Balzert, Helmut: *Lehrbuch der Software-Technik: Software-Entwicklung*. 2. Aufl. Heidelberg : Spektrum Akad. Verl., 2001 (Lehrbücher der Informatik). – ISBN 3827404800

[Balzert und Priemer 2008] Balzert, Helmut ; Priemer, Jürgen: *Java 6: Anwendungen programmieren: Von der GUI-Programmierung bis zur Datenbank-Anbindung*. Herdecke : W3L-Verl., 2008 (IT lernen). – ISBN 3937137092

[BGI 650 2007] VBG: Verwaltungs-Berufsgenossenschaft: *Bildschirm- und Büroarbeitsplätze: Leitfaden für die Gestaltung*. September 2007. – Berufsgenossenschaftliche Information, BGI 650

[Black et al. 2009] Black, Andrew P. ; Ducasse, Stéphane ; Nierstrasz, Oscar ; Pollet, Damien: *Squeak by Example*. Vs. of 2009-09-29, rev. ed. with corr. Bern : Square Bracket Ass., 2009. – ISBN 978–3–9523341–0–2

[Bothe et al. 2009] Bothe, Klaus ; Hildebrandt, Michael ; Niestroj, Nicolas: *ATEO-System Komponente: SAMs: Verhaltensspezifikation*. Version: 2009. `https://www2.informatik.hu-berlin.de/swt/lehre/PR_MTI_0910/restricted/exercises/Aufgabe1-WS0910.pdf`, Abruf: 02.09.2010

[Dahm 2006] Dahm, Markus: *Grundlagen der Mensch-Computer-Interaktion*. München : Pearson Studium, 2006 (Informatik). – ISBN 3827371759

[DIN 9241-10 1996] Norm DIN EN ISO 9241-10 1996. *Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten - Teil 10: Grundsätze der Dialoggestaltung*. – DIN 9241-10

[DIN 9241-11 1998] Norm DIN EN ISO 9241-11 1998. *Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten - Teil 11: Anforderungen an die Gebrauchstauglichkeit - Leitsätze*. – DIN 9241-11

[Dzida et al. 1978] Dzida, Wolfgang ; Herda, Siegfried ; Itzfeldt, Wolf D.: User-perceived quality of interactive systems.

In: *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, IEEE Press, 1978, S. 188–195

[Endsley et al. 2003] ENDSLEY, Mica R. ; BOLTÉ, Betty ; JONES, Debra G.: *Designing for Situation Awareness: An Approach to User-Centered Design*. London : Taylor & Francis, 2003. – ISBN 0748409661

[Grechenig 2010] GRECHENIG, Thomas: *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. München : Pearson-Verl., 2010 (Pearson Studium). – ISBN 9783868940077

[Hasselmann in Bearb.] HASSELMANN, Michael: *Erweiterung einer Softwarekomponente zur Systemprozessüberwachung und -kontrolle in einer psychologischen Versuchsumgebung*. in Bearb.. – Unveröffentlichte Diplomarbeit, in Bearbeitung

[Herczeg 1994] HERCZEG, Michael: *Software-Ergonomie: Grundlagen der Mensch-Computer-Kommunikation*. 1. Aufl. Bonn : Addison-Wesley, 1994. – ISBN 3893196153

[Herczeg 2005] HERCZEG, Michael: *Software-Ergonomie: Grundlagen der Mensch-Computer-Kommunikation*. 2., vollst. überarb. Aufl. München : Oldenbourg, 2005 (Lehrbücher Interaktive Medien). – ISBN 3486250523

[Hildebrandt 2009] HILDEBRANDT, Michael: *Reengineering der Architektur der SAM-Komponente des ATEO-Systems und deren prototypische Implementation in Java*. Berlin, Humboldt-Universität zu Berlin, Diplomarbeit, 2009

[Hunt 1997] HUNT, John: *Smalltalk and Object Orientation: An introduction*. London : Springer, 1997. – ISBN 3540761152

[Ingalls et al. 1997] INGALLS, Dan ; KAEHLER, Ted ; MALONEY, John ; WALLACE, Scott ; KAY, Alan: Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In: *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM, 1997. – ISBN 0–89791–908–4, S. 318–326

[Kesselring 2009] KESSELRING, Kai: *Entwicklung einer Softwarekomponente zur Systemprozessüberwachung und -führung in einer psychologischen Versuchsumgebung*. Berlin, Humboldt-Universität zu Berlin, Diplomarbeit, 2009

[Krinner und Gross 2005] KRINNER, Cordula ; GROSS, Barbara-Ulrike ; STEFFENS, Christiane (Hrsg.): *Arbeitsteilung zwischen Entwicklern und Operateuren von Mensch-Maschine-Systemen - eine neue Perspektive auf Funktionsteilung in Mensch-Maschine-Systemen*. Berlin, 2005

[Leonhard in Bearb.] LEONHARD, Christian: *Fenster zum Prozess: Weiterentwicklung eines Operateursarbeitsplatzes im Projekt Arbeitsteilung Entwickler Operateur (ATEO)*. in Bearb.. – Unveröffentlichte Diplomarbeit, in Bearbeitung

[Lewis 1995] LEWIS, Simon: *The Art and Science of Smalltalk: [an introduction to object-oriented programming using VisualWorks]*. London : Prentice Hall, 1995 (Hewlett-Packard professional books). – ISBN 0133713458

[Maloney 2002] MALONEY, John: An Introduction to Morphic: The Squeak User Interface Framework. In: GUZDIAL, Mark (Hrsg.) ; ROSE, Kim (Hrsg.): *Squeak: OpenPersonal Computing and Multimedia*. Prentice Hall, 2002

[McLaughlin et al. 2008] MCLAUGHLIN, Brett D. ; POLLICE, Gary ; WEST, David ; SCHULTEN, Lars: *Objektorientierte Analyse und Design von Kopf bis Fuß: [ein Buch zum Mitmachen und Verstehen]*. 1. Aufl., 1., korrigierter Nachdr. Beijing : O'Reilly, 2008. – ISBN 978–3–89721–495–8

[Rätzmann 2003] RÄTZMANN, Manfred: *Software-Testing: [Tests, Verfahren, Werkzeuge ; die Praxis des Rapid Application Testings ; agiles Qualitätsmanagement]*. 1. Aufl. Bonn : Galileo Press, 2003 (Galileo computing). – ISBN 9783898422710

[Sarodnick und Brau 2006] SARODNICK, Florian ; BRAU, Henning: *Methoden der Usability Evaluation: Wissenschaftliche Grundlagen und praktische Anwendung*. 1. Aufl. Bern : Huber, 2006 (Huber). – ISBN 3456842007

[Schwarz 2009] SCHWARZ, Hermann: *Fenster zum Prozess: ein Operateursarbeitsplatz zur Überwachung und Kontrolle von kooperativem Tracking*. Berlin, Humboldt-Universität zu Berlin, Diplomarbeit, 2009

[Sharp 1997] SHARP, Alec: *Smalltalk by Example: The Developer's Guide*. New York : McGraw-Hill, 1997 (Computing McGraw-Hill). – ISBN 0079130364

[Shneiderman und Plaisant 2010] SHNEIDERMAN, Ben ; PLAISANT, Catherine: *Designing the user interface: Strategies for effective human-computer interaction*. 5. ed., internat. ed. Upper Saddle River, NJ : Addison-Wesley/Pearson, 2010. – ISBN 0321601483

[Sommerville 2007] SOMMERVILLE, Ian: *Software Engineering*. 8., aktualisierte Aufl. München : Pearson Studium, 2007 (it informatik). – ISBN 9783827372574

[Vonhoegen 2005] VONHOEGEN, Helmut: *Einstieg in XML: [für Entwickler und XML-Einsteiger ; Grundlagen, Praxis, Referenzen*

; XML-Schema, DTD, XSLT, XPath, DOM, SAX, SOAP ; neu: Office und XML]. 3., erw. Aufl. Bonn : Galileo Press, 2005 (Galileo computing). – ISBN 3898426300

[Wandke 2003] WANDKE, Hartmut: *Forschungsschwerpunkt ATEO*. 2003

[Wandke und Nachtwei 2008] WANDKE, Hartmut ; NACHTWEI, Jens: The different human factor in automation: the developer behind vs. the operator in action. In: WAARD, D. de (Hrsg.) ; FLEMISCH, F. O. (Hrsg.) ; LORENZ, B. (Hrsg.) ; OBERHEID, H. (Hrsg.) ; BROOKHUIS, K. A. (Hrsg.): *Human Factors for Assistance and Automation*. Maastricht, the Netherlands : Shaker Publishing, 2008. – ISBN 978–90–423–0350–8, S. 493–502

[Wessel 2002] WESSEL, Ivo: *GUI-Design: Richtlinien zur Gestaltung ergonomischer Windows-Applikationen*. 2., aktualisierte und bearb. Aufl. München : Hanser, 2002. – ISBN 3446219617

ERKLÄRUNG

SELBSTSTÄNDIGKEITSERKLÄRUNG

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Berlin, den 13. Oktober 2010

Esther Fuhrmann

EINVERSTÄNDNISERKLÄRUNG

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der für das Institut für Informatik zuständigen Zweigbibliothek der Humboldt-Universität zu Berlin bereitgestellt werden darf.

Berlin, den 13. Oktober 2010

Esther Fuhrmann