Bachelorarbeit

Alexej Tietz

Entwurf und Realisierung einer
Anwendungssuite für einen Eye-Tracker

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexej Tietz

Entwurf und Realisierung einer Anwendungssuite für einen Eye-Tracker

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Olaf Zukunft
Zweitgutachter : Prof. Dr. Bettina Buth

Abgegeben am 28. März 2011

Alexej Tietz

Thema der Bachelorarbeit

Entwurf und Realisierung einer Anwendungssuite für einen Eye-Tracker

Stichworte

Eye-Tracker, Eye-Tracking, Erstellung von Plug-Ins, 2D-Rendering in Java, Laden von Klassen zur Laufzeit in Java

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Erstellung einer Anwendungssuite für einen Eye-Tracker, die aus einer API und einigen Modulen besteht. Die API und die Module sind in Java entwickelt worden. Bei der API können zur Laufzeit verschiedene Programm-Module geladen und entladen werden. Diese werden als Plug-Ins angesehen und sind mit den menschlichen Augen über die API steuerbar.

Alexej Tietz

Title of the paper

Design and Realization an Application Suite for an Eye-Tracker

Keywords

Eye-Tracker, Eye-Tracking, Creation of Plug-Ins, 2D-Rendering in Java, the loading of classes in Java at runtime

Abstract

This paper describes the creation of an application suite for an eye-tracker, which has an API and some API-modules. The API and the modules are developed in Java. By the API could be loaded and unloaded different program modules at runtime, the program modules are designed as plug-ins. The plug-ins are controllable over the API with human’s eyes.

Inhaltsverzeichnis

1. Einleitung 1

1.1. Motivation 1
1.2. Gliederung der Arbeit 2

2. Grundlagen/Definitionen 3

2.1. Eye-Tracking 3
2.2. 2D in Java 6

3. Konzept der API 9

3.1. API – „Multi Eye“ 9
3.1.1. Anforderungen 9
3.1.2. Benutzungs-/Anwendungsfälle 10
3.1.3. Zustandsdiagramm 11
3.2. Connector 11
3.2.1. Anforderungen 12
3.2.2. Benutzungs-/Anwendungsfälle 13
3.2.3. Zustandsdiagramm 14
3.3. Plug-In 14
3.3.1. Anforderungen 15
3.3.2. Benutzungs-/Anwendungsfälle 15
3.3.3. Zustandsdiagramm 16
3.4. Utility-Klassen 17

4. API-Umsetzung/Implementierung 19

4.1. Schnittstellen zu einem Connector, einem Plug-In und der API . . . 19
4.2. Selbst-Adaptivität der API 22
4.3. Verläufe im Lebenszyklus der API 24
4.4. Implementierung eines UDP-Connectors 26
4.5. Kopplungen zwischen den Komponenten der Anwendungssuite . . . 29
4.5.1. Assoziationen zwischen API und Connector 30
4.5.2. Assoziationen zwischen API und Plug-In 30
4.5.3. Assoziationen zwischen Plug-In und Connector 30
4.6. Testmöglichkeiten 33
4.7. Erstellung eines Beispiel-Plug-Ins – „Hallo Welt!“. 36
4.8. Problematik 40
4.9. Test von dem UDP-Connector 42

5. Plug-Ins **45**
5.1. Grundlegender Aufbau eines 2D-Plug-Ins 45
5.2. Grundlegendes multifunktionales Objekt der 2D-Welt 47
5.3. Aktivierungsmanager 54
5.4. Plug-In: „Sky Defendor“ (2D-Spiel) 56
5.4.1. Erweiterung des 2D-Objekts „*AnimatableWorldObject*“ mit der Klasse „*SDDefendor*“ 56
5.4.2. Importe, Klassenvariablen und Konstruktor der Klasse „*MEPlugInSkyDefendor*“ 58
5.4.3. Hilfsmethoden der Klasse „*MEPlugInSkyDefendor*“ 66
5.4.4. Initialisierung des Menüs und der 2D-Welt 67
5.4.5. Datagetterthread und Animationsthread 68
5.4.5.1. Datagetterthread 68
5.4.5.2. Animationsthread 69
5.4.5.2.1. Methode „*gameUpdate*“ 70
5.4.5.2.2. Methode „*gameRender*“ 71
5.4.5.2.3. Methode „*paintScreen*“ 72
5.5. Plug-In: „Simple Eye View” (Textdokument-Reader) 72
5.5.1. Klasse „*SEVDoc*“ – Umhüllung für die SEVD-Textdokumente . . . 73
5.5.2. Importe und Klassenvariablen der Klasse „*MEPlugInSimpleEyeView*“ . . 75
5.5.3. Initialisierung der Views 77
5.5.4. Animationsthread 78
5.5.4.1. Methode „*gameUpdate*“ 78
5.5.4.2. Methode „*gameRender*“ 79

6. Zusammenfassung und Ausblick **80**
6.1. Bewertung 80
6.2. Zusammenfassung 80
6.3. Ausblick 81

Anhang: Ergebnisse der praktischen Arbeit auf CD-ROM **82**

Literaturverzeichnis **83**

Abbildungsverzeichnis **85**

Tabellenverzeichnis **87**

Versicherung über Selbstständigkeit **88**

1. Einleitung

Die Interaktion zwischen Mensch und Computer verbessert sich mit jedem neuen Tag und damit steigert sich auch die Komplexität der Kommunikation. Es werden neue Software und Hardware entwickelt, um sich das Leben zu erleichtern. Vor 20 Jahren brauchte man einen bestimmten Wissensstand, um eine einzige e-Mail zu versenden. Heutzutage sind die Programme so entwickelt, dass sie gewissermaßen erraten, was man von ihnen verlangt.
Es wird nach neuen Kommunikationsschnittstellen gesucht, z. B. bei der Spielekonsole Nintendo *Wii* des Herstellers Nintendo erkennen die Sensoren im Kontroller die Bewegungsrichtung des Spielers, dies ist nur dann möglich, wenn der Kontroller bewegt wird. Was ist mit Leuten, die beispielsweise querschnittsgelähmt sind oder keine Hände haben? Wie kann man die Kommunikation mit dem Computer auch für diese Personengruppe gestalten?

1.1. Motivation

Eine weitere Möglichkeit der Kommunikation bietet der Eye-Tracker. Der Eye-Tracker ist ein elektronisches Gerät, mit dessen Hilfe sich anhand des Blickwinkels und der Position der Pupillen einer Person feststellen lässt, welche Punkte innerhalb des Erfassungsbereichs (z. B. eines Computermonitors) die Augen des Menschen anvisieren.

Bei dieser Arbeit wird versucht die Kommunikation: ***Augen->Eye-Tracker->Computer*** zu gestalten und mit einigen Programmstücken zu testen.
Da von einer Schnittstelle die Rede ist, ist die Entwicklung einer API notwendig. Diese API soll die Rolle einer Schnittstelle erfüllen, die die Anpassungsfähigkeit bezüglich der Verwendung eines Eye-Trackers anbieten kann. Die Anpassungsfähigkeit ist hier in dem Sinne, dass man verschiedene Eye-Tracker verwenden kann, ohne dass man irgendwelche Änderungen an der API vornehmen muss. Um solche Ergebnisse erreichen zu können, muss die API eine bestimmte Modularität anbieten können, so dass es anpassbare Komponenten geben kann.
Die Komponenten sollten die Rolle eines Umwandlers bzw. Adapters annehmen und anpassungsfähig sein.

Die erwähnten Testprogramme sollen ebenfalls modular konzipiert werden. Sie sind im Umfang sehr komplex und werden daher als Plug-Ins angeboten. In diesem Fall bietet jedes Plug-In die Funktionalität eines vollständigen Programms, das zur Laufzeit geladen und entladen werden kann.

Die Komponente zur Verbindung mit einem Eye-Tracker mit der Funktionalität eines Adapters, ein modular-aufgebautes Plug-In mit der Funktionalität eines ganzen Programms und eine API, bei der all die erwähnten Komponenten verwendet werden – das alles könnte man in einem Rahmenwerk (engl. Framework) unterbringen, das bestimmte Funktionalitäten anbietet und auch fordert.

1.2. Gliederung der Arbeit

Diese Arbeit besteht im Wesentlichen aus vier Abschnitten, die hier kurz erläutert werden.

Kapitel 2 befasst sich mit grundlegenden Begriffen und Definitionen. In diesem Abschnitt vorkommende Begriffe werden in weiteren Kapiteln verwendet und können in diesem Kapitel nachgeschlagen werden.
Die Begriffe sind in zwei verschiedene Bereiche unterteilt: Eye-Tracking und 2D in Java.

Kapitel 3 beschreibt die Planung der API. Am Anfang werden die funktionalen und die nichtfunktionalen Anforderungen seitens Software Engineering aufgezeigt. Damit die Benutzbarkeit der API nicht gleich umständlich wird, werden grundlegende Anwendungsfälle im Auge behalten.

Kapitel 4 enthält die Umsetzung des Konzeptes der API und deren Komponenten. Dieser Abschnitt besteht zu einem großen Teil aus dem Javacode. Einige Tricks und Tipps der Realisierung werden hier offen gelegt. Vor allem wird die Kopplung zwischen der API und deren Komponenten gezeigt. Stellen, wo sich die Problematik entfaltet, werden unter die Lupe genommen und deren Lösung präsentiert.

Kapitel 5 zeigt, wie die Plug-Ins realisiert sind und was die Kommunikation mit dem Computer über einen Eye-Tracker problematisch macht. Zusätzlich werden einige Konzepte aus 2D-Welt erläutert und die grundsätzliche Vorgehensweise geschildert.

Kapitel 6 ist ein Abschlusskapitel, das eine Zusammenfassung dieser Arbeit, eine Bewertung der gewonnenen Erfahrungen und einen Ausblick auf mögliche weitere Erweiterungen und Verbesserungen enthält.

2. Grundlagen/Definitionen

In diesem Abschnitt werden die grundlegenden Begriffe und Definitionen erläutert, mit den man in weiteren Kapiteln konfrontiert wird.

Die zu erläuternden Begriffe und Definitionen sind in zwei verschiedene Gruppen eingeteilt, damit dem Leser die Zuordnung leichter fällt, und keine Verwechselungen zu Stande kommen.

Die Gruppeneinteilung sieht wie folgt aus:

- ***Eye-Tracking*** – Begriffe, die im Zusammenhang mit einem Eye-Tracker stehen. Es wird etwas über die Vorbereitung des Eye-Trackers vor der Benutzung erzählt und was man beim Aufbau einer Kommunikation mit einem Eye-Tracker berücksichtigen sollte.

- ***2D in Java*** – einige Spezialitäten der 2D-Handhabung in Java und ein paar Allgemeinheiten mit deren Bedeutung.

2.1. Eye-Tracking

Ein Eye-Tracker ist ein Gerät, das für Messungen der Augenposition verwendet wird (vgl. Lischka 2008a). Es gibt verschiedene Arten/Modifikationen (siehe Abbildung 2.1) des Geräts. Eine Einsicht in Techniken des Eye-Trackings bietet Kumar in seiner Doktorarbeit (Kumar 2007a). Im Grunde genommen ist ein Eye-Tracker nichts anderes als eine intelligente digitale Kamera. Es gibt nur einen einzigen Unterschied, dass ein Eye-Tracker nicht digitale Bilder zurück liefert, sondern digitalen Strom aus X-Y-Koordinaten & Co (Pupillengröße, linke und rechte Augenposition, etc.). Der Prozess der Verwendung des Eye-Trackers wird als Eye-Tracking bezeichnet. Bei diesem Prozess wird die Position der Augen aus einem Bild periodisch mit einer bestimmen Wiederholfrequenz ermittelt. Aus den ermittelten Daten wird die Blickposition berechnet und an eine Applikation zur Auswertung übermittelt.

Abbildung 2.1: Tobii Glasses, X60 & X120, T60 & T120 Eye-Trackers (Tobii 2010a)

Es gibt viele verschiedene Methoden des Eye-Trackings. Die am meisten verbreitete ist Cornea-Reflex, bei der die Augen von Kameras aufgezeichnet und gleichzeitig mit einem schwachen Infrarot-Licht beleuchtet werden. Durch das Beleuchten der Augen entsteht eine Reflektion auf der Hornhaut (Cornea) (siehe Abbildung 2.2), die zur Feststellung der Blickrichtung verwendet wird (vgl. Ray 2010a).

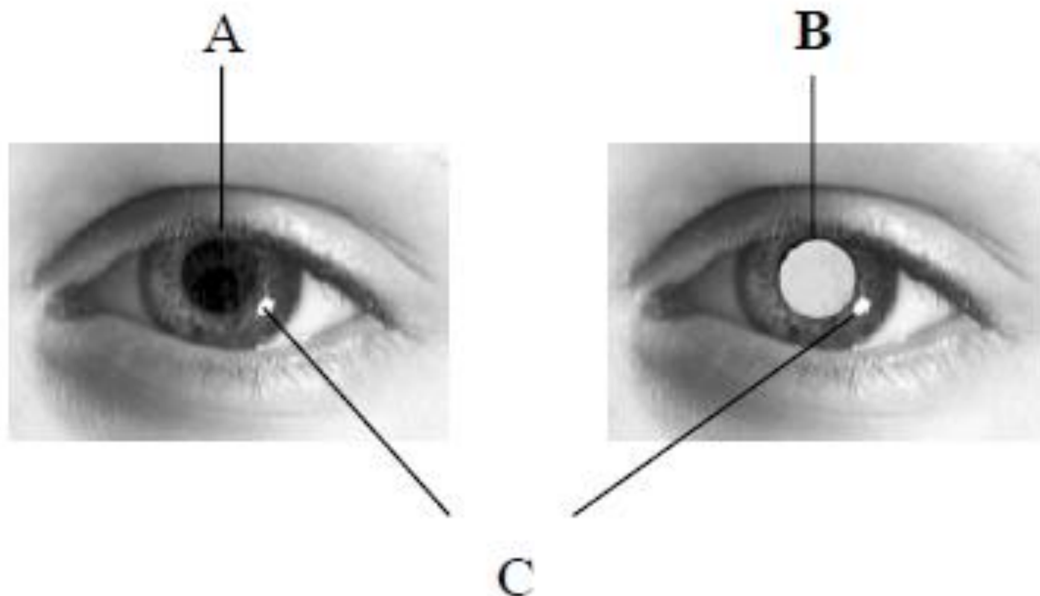


Abbildung 2.2: Beispielbilder für „dunkle Pupille" (A) und „infrarot-licht beleuchtete Pupille" (B) sowie der Infrarot-Reflektionen auf der Hornhaut (C) (Milekic 2003a)

Bei dem Eye-Tracking existieren zwei bedeutsame Zustände, in denen sich die Augen befinden können:

- ***Fixation*** – als Fixation bezeichnet man einen Anhaltspunkt der Augen beim Betrachten, also eine kurzweilige Anvisierung eines Ziels/Teils.

- ***Sakkaden*** – Bei Sakkaden handelt es sich um kurzzeitige Sprünge von einer Fixation zur nächsten. Man kann auch von einem Blickwechsel sprechen. Während einer Sakkade erfolgt keine Informationsaufnahme. Die Augenbewegungen erfolgen dabei sehr schnell und ruckartig (Ray 2010a).

Vor der Verwendung benötigt ein Eye-Tracker eine Art „Gerät-Grundzustand“. Das Gerät wird durch eine Kalibrierung in diesen Zustand versetzt. Unter einer Kalibrierung wird eine Operation verstanden, die unter folgenden Bedingungen:

- eine Beziehung zwischen der Anzahl der gemessenen Werte und darin vorkommenden Messunsicherheiten bereitstellt, ermöglicht durch Messstandards und korrespondierenden Anzeichen mit Assoziierung der Messunsicherheiten.
- diese Information nutzt, um eine Beziehung für das Erlangen eines Messergebnisses von einem Hinweis zu etablieren.

ANMERKUNG 1: *Eine Kalibrierung kann durch eine Erklärung zum Ausdruck gebracht werden. z. B. durch eine Kalibrierungsfunktion, ein Kalibrier-Diagramm, eine Kalibrier-Kurve oder Kalibrierungstabelle. In einigen Fällen kann es aus einer additiven oder multiplikativen Korrektur der Angabe mit zugehöriger Messunsicherheit bestehen.*

ANMERKUNG 2: *Kalibrierung sollte nicht mit der Anpassung des Messsystems verwechselt werden, die oft fälschlicherweise als Selbstkalibrierung oder Überprüfung der Kalibrierung genannt wird* (vgl. VIM 2008a).

Im Falle eines Eye-Trackers werden die Grenzen des betrachteten Objekts (z. B. Computerbildschirm) und die Abweichung zwischen der erwarteten und der tatsächlichen Position der Augen ermittelt.

Da ein Eye-Tracker ein selbstständiges Gerät ist, darf man bei der Verbindung mit ihm einen wichtigen Aspekt nicht aus den Augen verlieren. Dieser Aspekt ist die Übermittlung der Daten von einem Eye-Tracker und zwar die Byte-Reihenfolge, in der ein Eye-Tracker redet (kommuniziert). Eine Byte-Reihenfolge beschreibt, wie die Daten physikalisch auf einem Datenträger gespeichert oder übertragen werden, also wie man die Bytes zu interpretieren hat. Es gibt zwei gängige Formate (siehe Abbildung 2.3):

- ***Little Endian*** – bei Little Endian liegt das kleine Byte auf der kleinen Adresse. Die Byte-Reihenfolge wird z. B. von Intel 80x86 benutzt.
- ***Big Endian*** – bei Big Endian liegt das kleine Byte auf der große Adresse. Die Byte-Reihenfolge wird z. B. von Motorola 680x0 verwendet. (vgl. Schäfers 2009a)

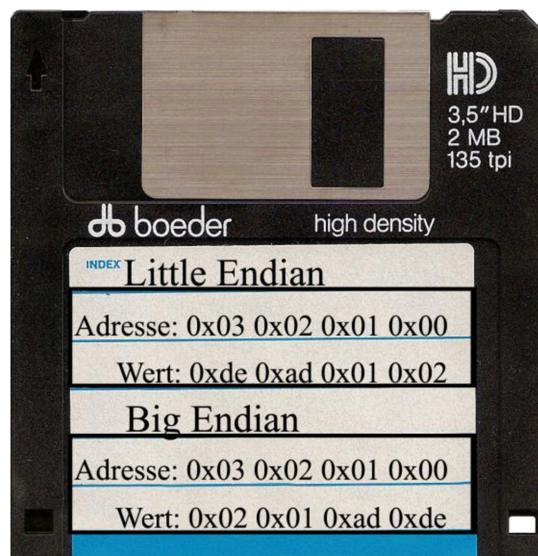


Abbildung 2.3: Byte-Reihenfolge: Little Endian und Big Endian (32Bit-Wert:0xdead0102)

2.2. 2D in Java

Wenn man aus einer API (engl. **A**pplication **P**rogramming **I**nterface) etwas mehr als eine Anwendungsschnittstelle machen will, sodass sie mächtiger und nutzvoller wird, bietet man zusätzlich Utility-Klassen (Hilfsklassen) an, um den Umgang mit der Anwendungsschnittstelle für Programmierer zu erleichtern. Durch diese Erweiterung versucht man ein Framework zu erschaffen.
Die API, die in dieser Arbeit beschrieben wird, wird auch solche Hilfsklassen anbieten. Einige der Utility-Klassen werden Manipulationen von digitalen Bildern vornehmen.

In diesem Teilabschnitt kommen ein paar grundlegenden Begriffen bezüglich der 2D-Handhabung in Java.

Eine Generierung von Bildern aus zweidimensionalen Objekten heißt **2D-Rendering**. Die Java 2D API ™ bietet ein einheitliches Rendering-Modell in verschiedenen Arten von Geräten. Auf der Anwendungsebene ist der Rendering-Prozess der gleiche, unabhängig davon, ob das Zielgerät ein Bildschirm oder ein Drucker ist. Wenn eine Komponente angezeigt werden muss, wird ihre Mal- oder Update-Methode automatisch mit dem entsprechenden Grafikkontext aufgerufen.
Die Java 2D API enthält eine Graphics2D-Klasse, die der Graphics-Klasse den Zugriff auf die verbesserte Grafik-und Rendering-Funktionen liefert. Zu diesen Funktionen zählen:

- Rendering der Umrisse von geometrischen Primitiven (**Draw**-Methode).
- Rendering von geometrischen Primitiven durch das Ausfüllen des Interieurs mit Farbe oder einem Muster (**Fill**-Methode).
- Rendering eines beliebigen Text-Strings (**drawString**-Methode).
- Rendering eines angegebenen Bildes (**DrawImage**-Methode) (vgl. Oracle 2010a).

Es gibt eine sehr nutzvolle Klasse „***BufferedImage***“, die für eine Repräsentierung von digitalen Bildern genutzt wird. Die Klasse ist eine Unterklasse von der Klasse „*Image*“ (Bild) mit einem zugänglichen Puffer der Bilddaten. Eine BufferedImage-Instanz wird von einem ***ColorModel*** (Mapping der Bilddaten auf ein Farbmodell) und einem ***Raster*** (Pixel - Bilddaten und ihre Interpretation) der Bilddaten umfasst. Alle BufferedImag*e*-Objekte haben in der linken oberen Ecke den Koordinatenursprung und benutzen das ARGB-Farbmodell (**A**lpha Channel-**R**ot-**G**rün-**B**lau) (vgl. Oracle 2010b). Die Benutzung der Klasse „*BufferedImage*“ ist vorteilhaft im Vergleich zu „*Image*“, da Bildmanipulationen (Skalieren, Drehen, etc.) über Methoden möglich sind.

Fade-in (Einblendung) und ***Fade-out*** (Ausblendung) Effekte können leicht erreicht werden, wenn man ein BufferedImage verwendet. Das ist nur deswegen möglich, weil ein BufferedImage einen so genannten ***Alpha Channel*** besitzt. Um besser zu verstehen, was ein Alpha Channel ist, wird hier das Kleinste eines digitalen Bildes unter die Lupe genommen und zwar ein Pixel und seine farbliche Repräsentation.
Die Klasse ***„Color“*** (Farbe) benutzt den sRGB (**S**tandard **R**ot **G**rün **B**lau) Farbraum. Jede Farbe hat einen indirekten Alphawert im Alphakanal (engl. Alpha Chanel) (vgl. Oracle 2010c). Dieser Wert gibt die Transparenz dieser Farbe/des Pixels an. Der Alphakanal kann einen Wert von 0.0 bis 1.0 oder 0 bis 255 (Sache der Interpretation) annehmen (siehe Abbildung 2.4).
Bedeutung des Werts:

- 0.0 oder 0 → das Farbpixel ist völlig transparent
- 1.0 oder 255 → das Farbpixel hat keine Transparenz

Abbildung 2.4: Alphawert des Marmorsteins

Eine weitere Modifikation eines BufferedImage ist durch **Affine Transformation** (engl. Affine Transform) möglich. Die Klasse „*AffineTransform*“ stellt eine 2D-Affine Transformation zur Verfügung, führt eine lineare Abbildung von 2D-Koordinaten zu andere 2D-Koordinaten durch und bewahrt die "Geradlinigkeit" und "Parallelität" der Linien.
Durch eine Affine Transformation kann man verschiedene grafische Effekte erreichen, sowie Rotation, Translation, Skalierung und Spreizung (vgl. Oracle 2010d).

Um einfache und komplexe Umrisse eines 2D-Objektes in Java 2D zu zeichnen oder mit einer Farbe zu füllen, kann man Polygone verwenden. Die Klasse ***„Polygon“*** kapselt die Beschreibung eines geschlossenen zweidimensionalen Koordinatensystems innerhalb eines Raumes. Dieses Koordinatensystem ist durch eine beliebige Anzahl von Strecken begrenzt, von denen jede eine Seite des Polygons ist. Intern besteht ein Polygon aus einer Liste von X-Y-Koordinatenpaaren, wobei jedes Paar einen Eckpunkt des Polygons bestimmt. Zwei aufeinanderfolgende Paare bestimmen die Endpunkte einer Linie, sodass diese zusammen eine Seite des Polygons bilden. Die ersten und die letzten X-Y-Punkte werden ebenfalls durch eine Linie verbunden, die das Polygon abschließt (vgl. Oracle 2010e).

Ein sehr wichtiges Konzept der 2D-Programmierung und besonders bei der 2D-Spieleprogrammierung ist die **Animationsschleife** (engl. Animation Loop)(siehe Abbildung 2.5).

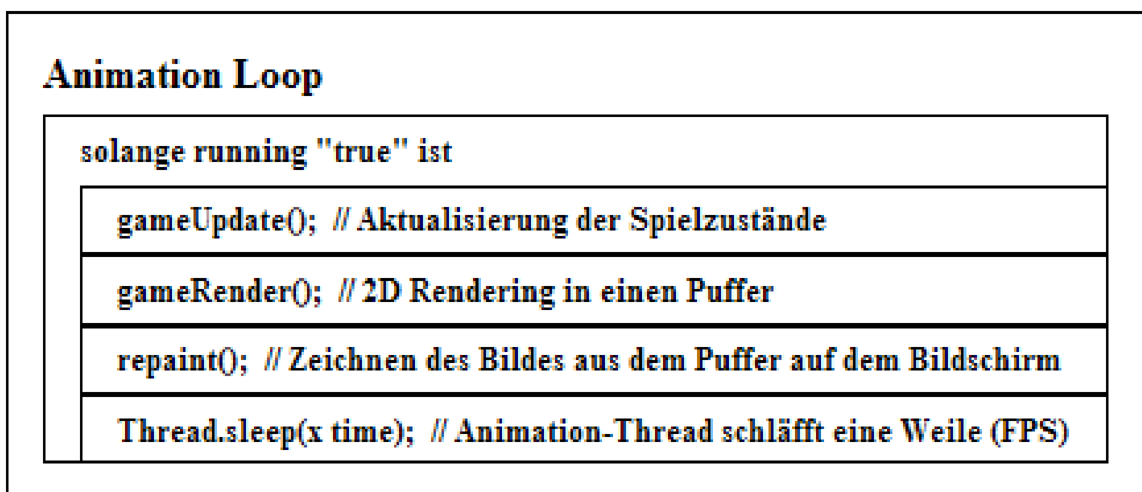


Abbildung 2.5: Animationsschleife-Struktogramm nach Nassi-Shneiderman

Wie man aus dem Strukturgramm entnehmen kann, besteht die Animationsschleife aus vier Abschnitten:

- ***gameUpdate*** – bei dieser Methode werden die internen Zustände der Spielobjekte aktualisiert. Es wird zusätzlich auf Benutzerinteraktionen reagiert und die interaktionsbezogenen Änderungen bezüglich der Spielwelt werden durchgeführt. z. B. Spielfigur ist gesprungen oder Spielfigur ist am Finish angekommen.

- ***gameRender*** – bei dieser Phase werden die Zustände der Spielobjekte und der Welt in einer 2D-Repräsentation generiert, welche in einen Puffer geschrieben wird. Durch das Schreiben in einen Puffer vermeidet man das Flackern des Bildschirmes.

- ***repaint*** – hier wird die im vorherigen Schritt gerenderte 2D-Repräsentation auf den Bildschirm übertragen.

- ***Thread.sleep*** – mit dieser Methode will man zwei Sachen erreichen. Als erstes den Thread schlaffen legen, bei dem die Animationsschleife läuft, um die CPU was anderes machen zu lassen (schließlich hat man nicht nur ein Programm am Laufen). Und zuletzt will man eine bestimmte Bildrate erreichen, sodass eine feste Bildwiederholfrequenz (engl. FPS – **F**rame **P**er **S**econd) zu Stande kommt (vgl. Davison 2004a).

Im nächsten Kapitel (Kapitel 3) wird die Planung der API Schritt für Schritt durchgangen. Die Planung wird seitens Software Engineering angefasst. Es werden dadurch entstandene Dokumente repräsentiert. Das Kapitel wird mit einigen Gedanken bezüglich der Utility-Klassen abgeschlossen.

3. Konzept der API

Dieses Kapitel enthält die Konzipierung der API, des Verbindungsmoduls (engl. Connector - Konnektor) und des Plug-Ins. Zusätzlich werden ein paar Gedanken bezüglich Hilfsklassen (Utility-Klassen) geschildert. Der Aufbau der Planung ist für jede Komponente annähernd gleich.
Zuerst werden die Anforderungen (engl. Requirements) aufgelistet, danach kommen die Benutzungs-/Anwendungsfälle mit einem Bezug auf die Requirements. Zum Schluss wird mit einem Zustandsdiagramm der Zustandsverlauf präsentiert.

Die Struktur des Kapitels sieht so aus:
- ***API – „Multi Eye“***
- ***Connector***
- ***Plug-In***
- ***Utility-Klassen***

3.1. API – „Multi Eye“

Was könnte man von einer API erwarten? Wie soll sie benutzt werden? Was muss sie anbieten? Wie sollten die Plug-Ins und die Connectors bei der API angewandt werden? Welche Bildschirmmodi soll die API anbieten?
Viele Aspekte und die Teilaspekte muss man im Auge behalten, um eine durchdachte API zu konzipieren und die Feststellung von zielführenden Anforderungen ist für den Erfolg unabdingbar.

3.1.1. Anforderungen

Es ist von einer API zu erwarten, dass sie bestimmte funktionale und nichtfunktionale Anforderungen erfüllt.
Die folgenden Anforderungen sind in fachliche und softwaretechnische Anforderungen unterteilt. Die fachlichen beziehen sich auf einen Eye-Tracker und die softwaretechnischen auf eine softwaretechnische Realisierung.

3.1.1.1. Die API soll über eine grafische Benutzeroberfläche bedient werden *(softwaretechnisch).*
3.1.1.2. Sie muss eine Unterstützung für die Connectors bieten, die spezifisch für jede Gruppe von Eye-Trackern definiert werden *(fachlich/softwaretechnisch).*
3.1.1.3. Die Konfiguration der Connectors muss aus der API heraus möglich sein *(softwaretechnisch).*

3.1.1.4. Die Konfiguration der Connectors wird mit einer Maus und einer Tastatur durchgeführt *(softwaretechnisch)*.
3.1.1.5. Bei der API muss das Laden von Plug-Ins und Connectors möglich sein und zwar während der Ausführungszeit *(softwaretechnisch)*.
3.1.1.6. Die API soll den Vollbildmodus und den Fenstermodus unterstützen *(softwaretechnisch)*.
3.1.1.7. Die Plug-Ins sollen mit den menschlichen Augen über einen Eye-Tracker gesteuert werden *(fachlich/softwaretechnisch)*.
3.1.1.8. Die API muss eine Möglichkeit bieten, dass ein Plug-In sich selbst beenden kann *(softwaretechnisch)*.
3.1.1.9. Das Auswählen von zuladenden vorhandenen Plug-Ins und Connectors muss möglich sein *(softwaretechnisch)*.
3.1.1.10. Die Konfigurationen, die in der API vorgenommen werden können, müssen in einer übersichtlichen Form vorliegen *(softwaretechnisch)*.
3.1.1.11. Die Plug-Ins und Connectors können der API eigene Zustände mitteilen *(softwaretechnisch)*.
3.1.1.12. Die Plug-Ins werden aus der API heraus gestartet *(softwaretechnisch)*.
3.1.1.13. Die API und ein geladenes Plug-In können durch den Benutzer beendet werden *(softwaretechnisch)*.

3.1.2. Benutzungs-/Anwendungsfälle

Aus Anforderung 3.1.1.1 ist zu entnehmen, dass die API über eine GUI (engl. **G**raphical **U**ser **I**nterface – grafische Benutzerschnittstelle) bedient werden soll. Daraus folgernd werden alle Interaktionen auf den GUI-Widgets (grafische Fensterkomponenten) ausgeführt. Die Benutzungs-/Anwendungsfälle beziehen sich auf die Funktionen, die hinter der grafischen Schnittstelle stecken und durch ein Use Case Diagramm (siehe Abbildung 3.1) dargestellt werden, das aus den Anforderungen aus dem Abschnitt 3.1.1 entstanden ist.

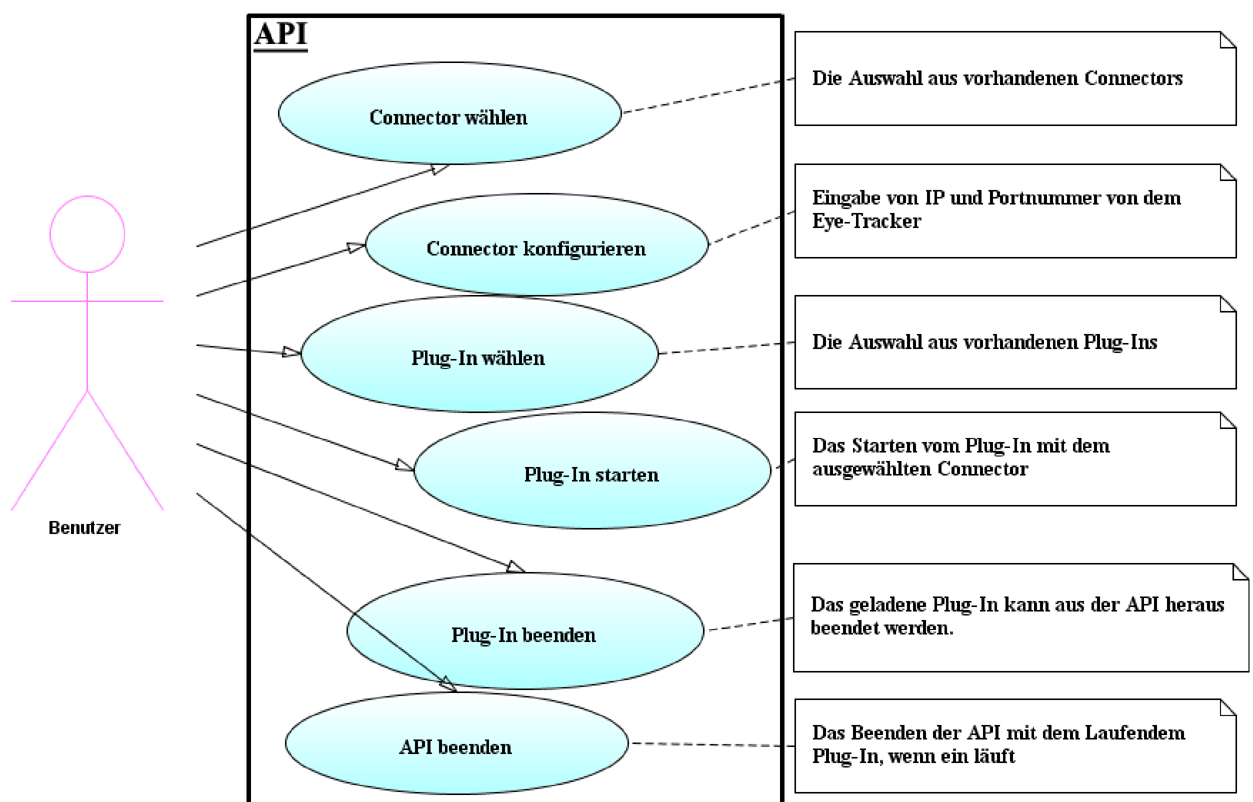


Abbildung 3.1: Use Case Diagramm der API

Das Diagramm zeigt die Interaktionen, die ein Akteur (Benutzer) an der API durchführen kann. Es muss berücksichtigt werden, dass ein ausgewähltes Plug-In nur dann erfolgreich gestartet werden kann, wenn ein richtiger Connector ausgewählt und richtig konfiguriert ist.
Die folgende Tabelle (siehe Tabelle 3.1) zeigt die Beziehung zwischen Anforderungen und Akteurinteraktionen.

Akteurinteraktionen	Anforderung
Connector wählen	3.1.1.9
Connector konfigurieren	3.1.1.3
Plug-In wählen	3.1.1.9
Plug-In starten	3.1.1.12
Plug-In beenden	3.1.1.13
API beenden	3.1.1.13

Tabelle 3.1: Bezug der Interaktionen auf Anforderungen der API

3.1.3. Zustandsdiagramm

Einige der wichtigsten Funktionen und Verläufe der API können dem Use Case Diagramm entnommen werden, aber den genaueren Ablauf und das Verhalten der API kann man am besten anhand eines Zustandsdiagramms (siehe Abbildung 3.2) sehen.

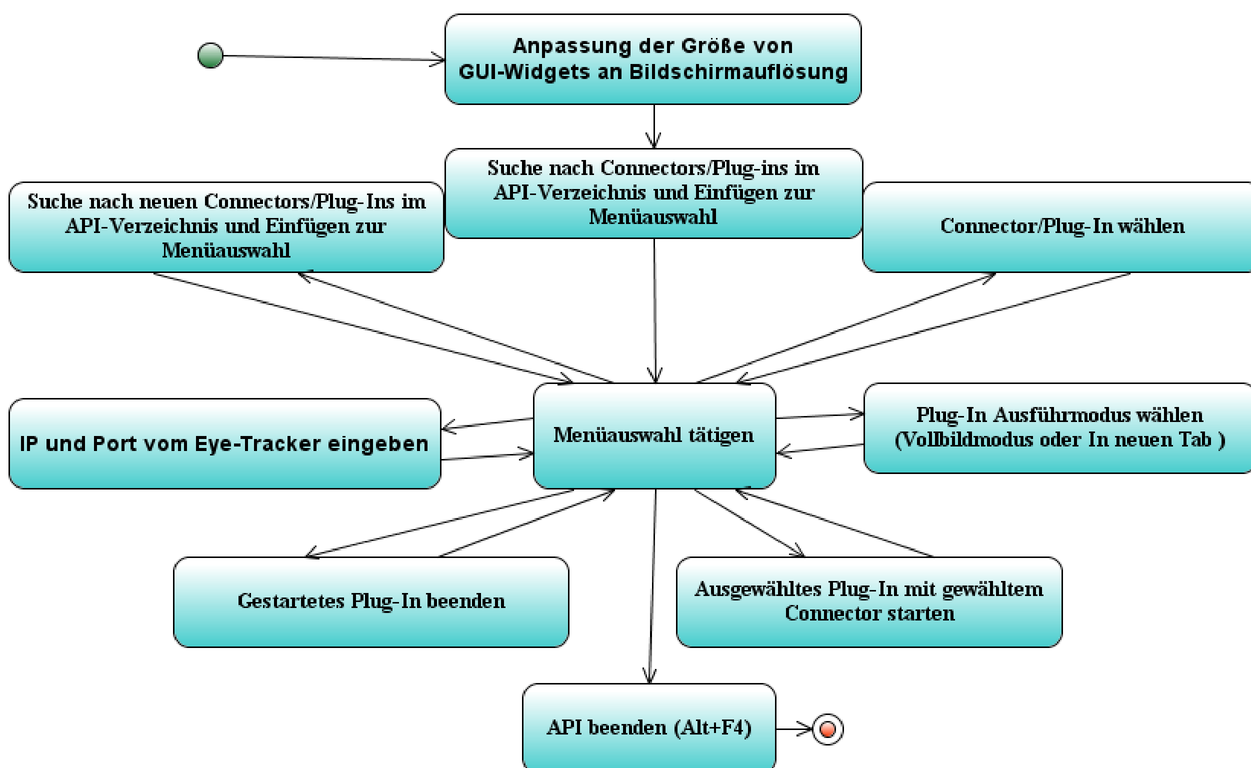


Abbildung 3.2: Zustandsdiagramm der API

3.2. Connector

Grundsätzlich ist ein Connector für eine Verbindung mit einer vorhandenen Quelle zuständig. In den meisten Fällen bietet ein Connector auch zusätzliche Funktionen an.

Bevor es zum nächsten Abschnitt übergegangen wird, kommen hier einige technische Spezifikationen (siehe Tabelle 3.2) und Abmessungen (siehe Abbildung 3.3) eines Eye-Trackers (Tobii X120 Series Eye Trackers), für das ein Connector entwickelt wird. Das Gerät wird mit 60Hz betrieben, deswegen werden nur die Eigenschafften für 60Hz-Btrieb aufgelistet.

Genauigkeit	typisch 0.5 Grad
Abtrift	typisch 0.1 Grad
Räumliche Auflösung	typisch 0.2 Grad
Kopfbewegungsfehler	typisch 0.2 Grad
Kopfbewegungsgrenzen (Breite x Höhe)	44 x 22 cm in 70 cm
Tracking Abstand	50-80 cm
Maximaler Blickwinkel	35 Grad
Höchste Kopf-Bewegungsgeschwindigkeit	25 cm/s
Latenz	maximal 33 ms
Blink-Tracking Erholung	maximal 17 ms
Zeit für Tracking Erholung	typisch 300 ms
Gewicht (ohne Gehäuse)	~ 3 kg / 7 lbs
Eye-Tracking-Technik	hellen und dunklen Pupille Tracking (beides)
Eye-Tracking-Server	eingebettet
Verbindungsanschlüsse	LAN, Power

Tabelle 3.2: Tobii X120 Series Eye Trackers – Technische Spezifikationen (Tobii 2010b)

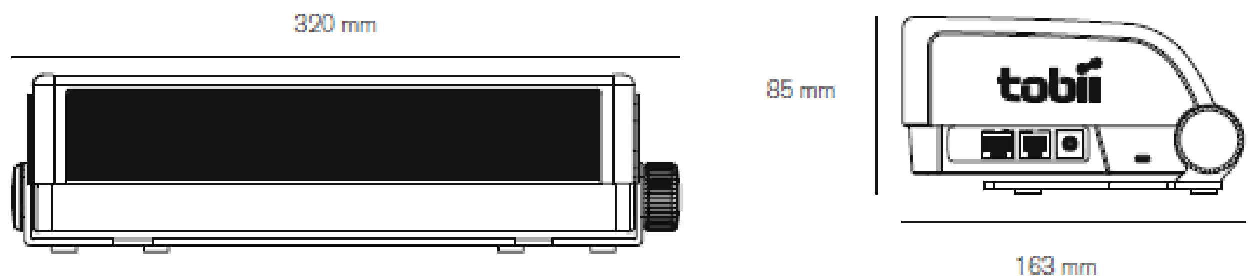


Abbildung 3.3: Tobii X120 Series Eye Trackers – Abmessungen (Tobii 2010b)

3.2.1. Anforderungen

In diesem Abschnitt werden einige Anforderungen aufgelistet, die von einem Connector zu erwarten sind. Die nächstkommenden Requirements sind in keinem Fall die Richtlinien für Entwickler eines Connectors, da eine Verbindung zu einem Eye-Tracker auf verschiedene Weise aufgebaut werden kann (TCP-IP, UDP-IP, über eine serielle Schnittstelle u. a.). Einige der Anforderung sind Spezialanforderungen, die nur für den Tobii X120 Series Eye Tracker gelten sollten.
Die folgenden Anforderungen sind wiederum in fachliche bzw. softwaretechnische Anforderungen unterteilt. Die fachlichen beziehen sich auf einen Eye-Tracker und die softwaretechnischen auf eine softwaretechnische Realisierung.

3.2.1.1. Der Connector muss eine UDP-Verbindung mit einem Eye-Tracker aufbauen können *(fachlich).*
3.2.1.2. Es muss die Möglichkeit bestehen, den geladenen Connector zu konfigurieren *(softwaretechnisch).*
3.2.1.3. Der Connector wird in einem der vorhandenen kompatiblen Plug-Ins eingesetzt *(softwaretechnisch).*

3.1.1.14. Die Schnittstelle zu einem Plug-In muss dynamisch angeboten werden, sodass ein Plug-In die meiste Intelligenz bezüglich der Connector-Verwendung besitzt *(softwaretechnisch).*
3.2.1.4. Connector muss die Daten vom Eye-Tracker periodisch empfangen können *(fachlich).*
3.2.1.5. Es muss möglich sein, auf die Verlaufsdaten zu zugreifen, die während der Verwendung des Connectors vom Eye-Tracker empfangen worden sind *(softwaretechnisch).*
3.2.1.6. Beim Beenden eines Connectors muss jede bestehende Verbindung geschlossen werden, um die Verbindungskonflikte zu vermeiden *(softwaretechnisch).*

3.2.2. Benutzungs-/Anwendungsfälle

Ein Connector ist eine Komponente, die nicht direkt vom Benutzer verwendet wird, sondern sie wird durch die API erstellt und konfiguriert. Anschließend wird sie bei einem ausgewählten Plug-In für das Empfangen der Daten von einem Eye-Tracker zuständig sein.
Aus diesem Grund tauchen in dem Use Case Diagramm (siehe Abbildung 3.4), das die Benutzungs-/Anwendungsfälle repräsentiert, zwei Akteure.

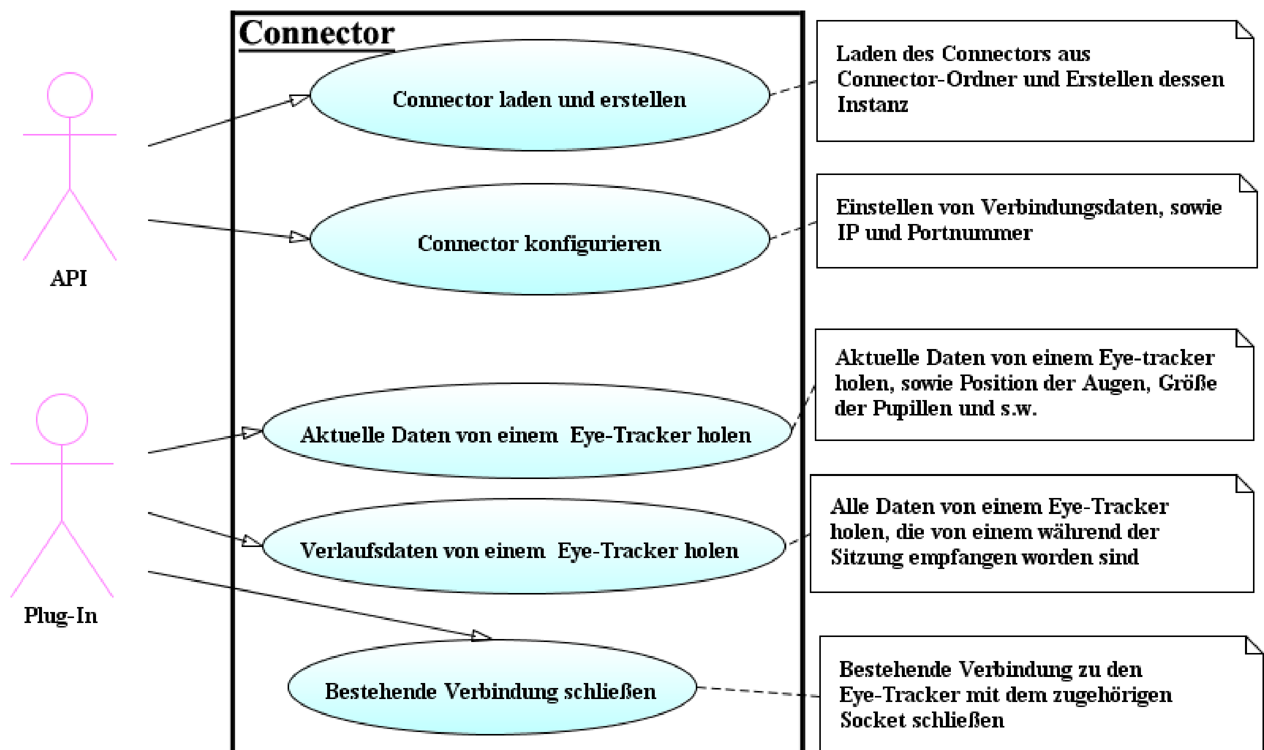


Abbildung 3.4: Use Case Diagramm eines Connectors

Demnächst kommt eine Tabelle (siehe Tabelle 3.3), in der die Beziehung zwischen Anforderungen und Interaktionen der Akteure gezeigt wird.

Interaktionen der Akteure	Anforderung
Connector konfigurieren	3.2.1.2
Aktuelle Daten von einem Eye-Tracker holen	3.2.1.5
Verlaufsdaten von einem Eye-Tracker holen	3.2.1.6
Bestehende Verbindung schließen	3.2.1.7

Tabelle 3.3: Bezug der Interaktionen auf Anforderungen eines Connectors

3.2.3. Zustandsdiagramm

Die Funktionalität eines Connectors kann dem folgenden Zustandsdiagramm entnommen werden. In dem Diagramm sind einige Aspekte gar nicht behandelt worden. Es enthält nicht, wie ein Connector eine Verbindung mit einem Eye-Tracker aufbaut, wie ein Connector Daten von einem Eye-Tracker empfängt und was er mit den empfangenen Daten tut. Diese Aspekte sind nur aus dem Grund nicht angesprochen, da es so viele Möglichkeiten für eine Verbindung mit einem Eye-Tracker bestehen und die Daten von einem Eye-Tracker auf verschiedene Weise einem Plug-In zur Verfügung gestellt werden können. Deswegen ist bei dem Diagramm nur auf die Kernfunktionalität eines Eye-Trackers eingegangen worden, die jeder Connector in der API anbieten muss.
Man muss berücksichtigen, dass die Daten von einem Eye-Tracker empfangen werden können, wenn auch die Verbindungsdaten eingestellt sind.

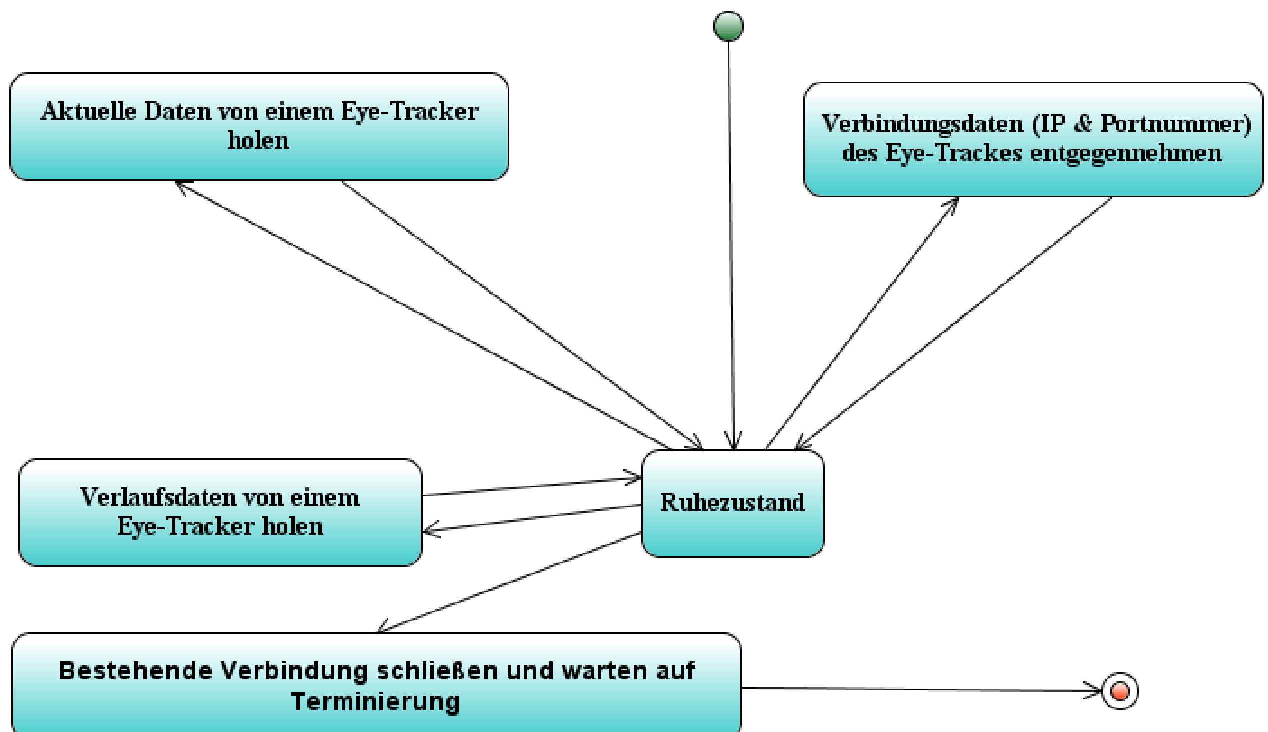


Abbildung 3.5: Zustandsdiagramm eines Connectors

3.3. Plug-In

Ein Plug-In ist eine Erweiterung eines Programms bzw. Frameworks. Allerdings ist dies bei der API-"Multi Eye anders: das Plug-In soll das Programm sein. Die API soll nur das Programm laden und mit dem Connector verkoppeln, dabei ist die ganze Logik und die Macht allein bei einem Plug-In.
Ein sehr sauberes Spiel für einen Framework-Entwickler würde man sagen, wenn irgendwas schief läuft, ist der Andere schuld. Es gibt aber einige Aspekte, die die Entwicklung des Frameworks doch nicht so leicht machen. Die Aspekte werden im nächstkommenden Kapitel (Kapitel 4) gründlich unter die Lupe genommen.

3.3.1. Anforderungen

Da ein Plug-In bei der API – „Multi Eye“ die Funktionalität fast jedes beliebigen Programms enthalten kann, sodass ein Plug-In-Entwickler viel Freiheit hätte, wird es schwierig irgendwelche softwaretechnischen Anforderungen zu schreiben. Aber bei der Entwicklung eines Frameworks müssen einige Rahmen für eine beteiligte Komponente gegeben sein, sonst ist allein nur der Name „Rahmenwerk“ widersprüchlich.
In diesem Fall ist es weder möglich die Kernfunktionalität und das genauere Verhalten zu schildern, obwohl man ein allgemeines Verhalten mit ein paar Anforderungen festlegen könnte.

3.3.1.1. Ein Plug-In wird mit den menschlichen Augen gesteuert.
3.3.1.2. Ein Plug-In muss ein Connector verwenden, um die Daten von einem Eye-Tracker zu empfangen.
3.3.1.3. Es muss eine Möglichkeit bestehen, ein Plug-In zu pausieren.
3.3.1.4. Ein Plug-In soll sich an eine übergebene Auflösung bei dem Start anpassen können.
3.3.1.5. Es muss eine Möglichkeit geben, ein Plug-In zu beenden.

3.3.2. Benutzungs-/Anwendungsfälle

Ein Plug-In wird als ein Endprodukt betrachtet, das von einem Eye-Tracker gesteuert wird, genauer gesagt wird es mit den menschlichen Augen angesteuert und ein Eye-Tracker ist nur das Mittel dazu. Das Produkt hat grundlegende Funktionen, die von mehreren Quellen aus benutzt werden können, deswegen tauchen auch mehrere Akteure in dem Use Case Diagramm (siehe Abbildung 3.6) auf, das die Benutzungs-/Anwendungsfälle schildert.

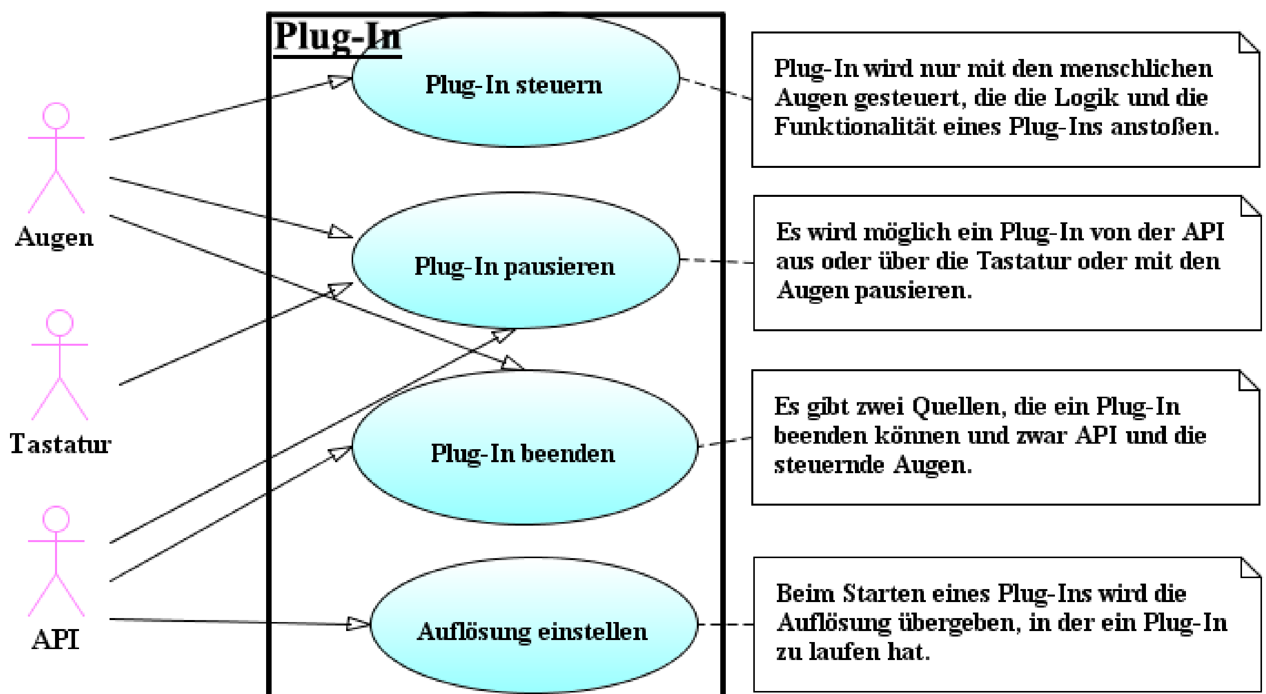


Abbildung 3.6: Use Case Diagramm eines Plug-Ins

In dem Diagramm wird gezeigt, dass ein Plug-In drei Akteure haben könnte, die Interaktionen auslösen können. Wie schon erwähnt, es ist nur das Grundlegende aufgezeigt, das ein Plug-In unterstützen muss. Nur ein Punkt ist hier abstrakt. Unter „Plug-In steuern“ könnte Vieles realisiert werden, da es gar nicht gesagt ist, wie und auf welche Weise es geschehen sollte.

Demnächst wird in einer Tabelle (siehe Tabelle 3.4) die Beziehung zwischen Anforderungen und Interaktionen der Akteure aufgestellt.

Interaktionen der Akteure	**Anforderung**
Plug-In steuern	3.3.1.1
Plug-In pausieren	3.3.1.3
Plug-In beenden	3.3.1.5
Auflösung einstellen	3.3.1.4

Tabelle 3.4: Bezug der Interaktionen auf Anforderungen eines Plug-Ins

3.3.3. Zustandsdiagramm

Im folgenden Zustandsdiagramm (siehe Abbildung 3.7) ist ein möglicher Zustandsverlauf eines Plug-Ins mit einer grundsätzlichen Funktionalität dargestellt. Das Diagramm soll nur ein Wegweiser für einen Plug-In-Entwickler und in keinem Fall ein strikter Lösungsweg sein.
Zudem muss auch hinzugefügt werden, dass einige Zustände eines Plug-Ins durchlebt werden müssen. Es sind folgende:

- Die Ausführungsauflösung muss vor dem Starten des Plug-Ins übernommen werden.
- Die Daten von einem Eye-Tracker müssen mit Hilfe eines Connectors empfangen werden.

Der nächstkommende Zustandsverlauf wäre empfehlenswert für eine Entwicklung eines Plug-Ins in 2D. Nur müsste man berücksichtigen, dass man die Daten von einem Eye-Tracker besser nicht in einer Animationsschleife (Kapitel 2 Abschnitt 2) holt, sondern parallel/nebenläufig empfängt, da man dadurch das 2D-Rendering beeinträchtigen könnte. In Folge dessen könnte sein, dass man ein schwankendes Verhalten bezüglich der Bildrate hat.

Auch bei der Entwicklung eines Plug-Ins mit den Standard-GUI-Widgets wäre es nicht verkehrt, wenn man das Verhalten aus dem Zustandsdiagramm entnimmt. Das Verhalten muss dann in einem zu der grafischen Oberfläche parallel laufenden Thread realisiert werden, der die Kommunikation mit einem Eye-Tracker aufrecht erhält.

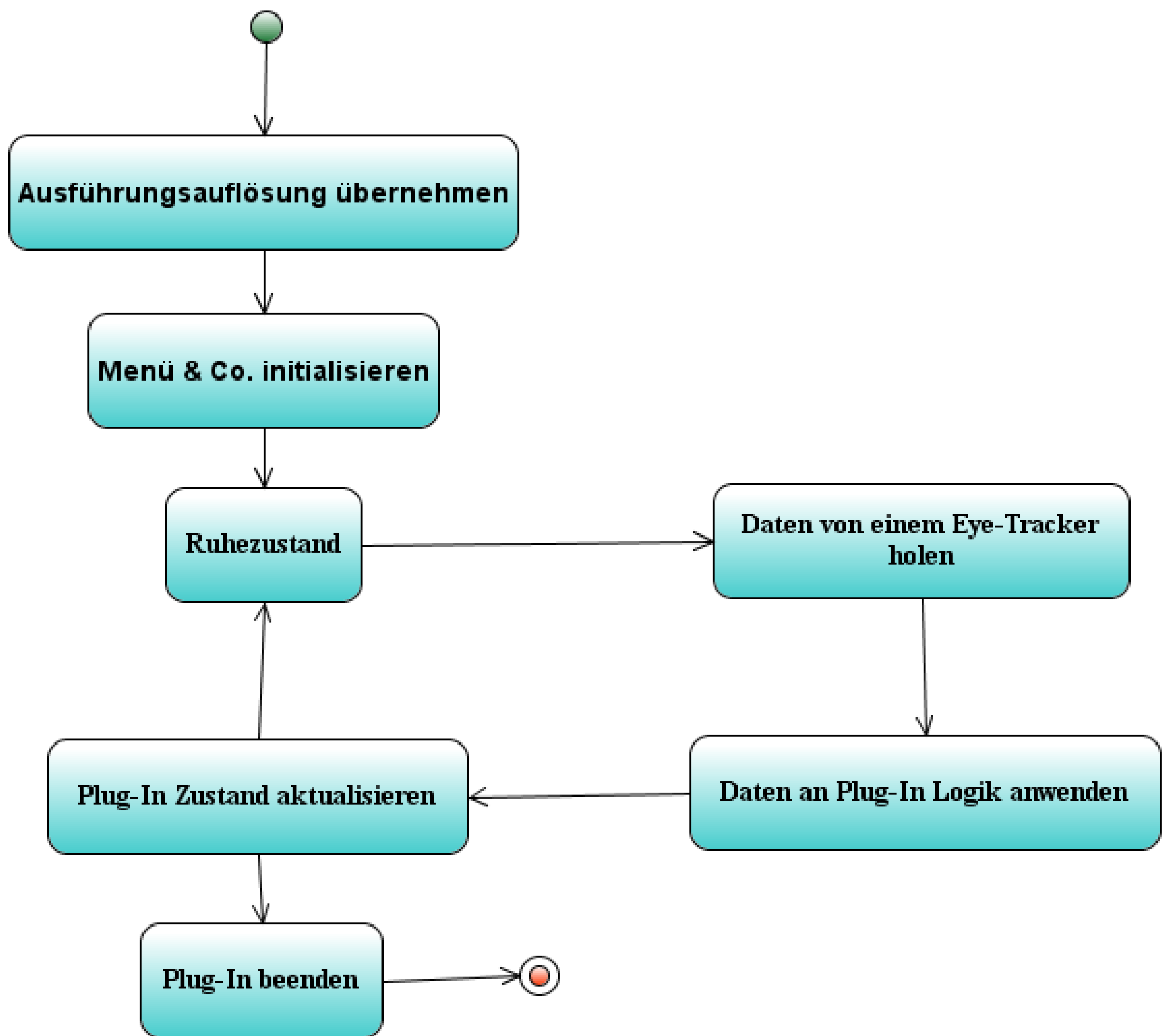


Abbildung 3.7: Zustandsdiagramm eines Plug-Ins

3.4. Utility-Klassen

Die Utility-Klassen sind Hilfsklassen, die die Entwicklung von Software einem Entwickler erleichtern. Viele Utility-Klassen werden als statische Klassen zur Verfügung gestellt. Sie müssen daher nicht instanziiert und können direkt über einen Bibliotheksaufruf benutzt werden.
Oft entstehen die Utility-Klassen während der Entwicklung eines Produkts, wenn man die häufig vorkommenden Codeabschnitte zu optimieren oder zu verbessern versucht. Die Klassen können eine mathematische oder technische Lösung einer Teilaufgabe anbieten.

Die Hilfsklassen bei der Anwendungssuite unterteilen sich in folgende Bereiche:

- ***Erstellung und Modifikation von BufferedImage-Objekten*** – hierzu gehören grundlegende Operationen, die man an einem BufferedImage durchführen kann. Als Erstes werden Methoden angeboten, mit den man BufferedImage-Objekte erstellen/umkonvertieren kann. Zum Nächsten werden die Bildmanipulationsoperationen angeboten und zwar Bildskalierung, Bildrotation und Bildsättigung.

- ***Laden und Erstellen von Klasseninstanzen*** – hier werden ein paar Methoden angeboten, mit den man das Laden und Erstellen von Klasseninstanzen aus einem lokalen Dateisystem durch die Verwendung eines Klassenladers vornehmen kann, egal ob die Klassen in einem JAR-Paket (**J**ava **Ar**chive) verpackt sind oder nicht. Die Klasse ist sehr wichtig für die Plug-Ins und Connectors, da sie mit Hilfe der Klasse geladen werden.

- ***Konverter für Byte-Reihenfolgen*** – bei der Klasse werden die Methoden zum Umkonvertieren von einigen Primitiven, sowie Integer und Double, aus Little Endian in Big Endian Byte-Reihenfolgen angeboten. Wenn man es andersrum umkonvertieren will, also aus Big Endian in Little Endian umwandeln, kann man dieselben Methoden verwenden, da die Umwandlung symmetrisch ist.

- ***Berechnung von Skalierungsfaktoren und Skalierung von Objekten*** – als Erstes wird eine Klasse für die Berechnung der Skalierungsfaktoren angeboten, mit der man die Skalierungsfaktoren für die Höhe, die Breite und den Mittelwert eines Objekts berechnen kann. Mit Hilfe der Skalierungsfaktoren ist es möglich die Größe der Dimensionen neu zu berechnen. Es werden zusätzlich Methoden zur Skalierung von Standard-GUI-Widgets aus Swing-Framework zur Verwendung gestellt.

Im nächstkommenden Kapitel (Kapitel 4) wird vor allem die API – „Multi Eye“ im Detail präsentiert. Es werden die Schnittstellen zur API, zum Connector und zum Plug-In gezeigt. Nachdem alle Schnittstellen offen gelegt sind, werden die Kopplungen zwischen den Komponenten und der API vorgelegt. Anhand eines Beispiels wird vorgeführt, wie man ein Plug-In erstellen kann. Nachfolgend kommt eine Problematik, die nicht sofort erkennbar ist. Das Kapitel wird mit einem Teilabschnitt über das Testen des UDP-Connectors abgeschlossen.

4. API-Umsetzung/Implementierung

In diesem Kapitel werden die Schnittstellen zu einem Connector, einem Plug-In und der API aufgezeigt. Zusätzlich wird die Verwendung der Schnittstellen erläutert. Die Selbst-Adaptivität und einige Verläufe im Lebenszyklus der API werden im Detail präsentiert. Die Darstellung eines Plug-ins und die Möglichkeiten, die in Verbindung mit dem gewählten Weg der Realisierung stehen, werden gründlich durchleuchtet. Die Implementierung eines UDP-Connectors wird von der dynamischen Seite vorgestellt.
Die Kopplungen zwischen all den Komponenten der Anwendungssuite werden nicht nur für die Kommunikation mit einem Eye-Tracker verwendet, sondern sie können für das Testen von Connectors und Plug-Ins missbraucht werden, was auch Sinn und Zweck von einigen Kopplungen ist.
Es wird ein Beispiel-Plug-In erstellt, das ohne Verwendung eines Connectors vorgestellt wird, um einen knappen Einblick in die Plug-In-Erstellung zu geben. Zusätzlich werden die Testmöglichkeiten der API an einem Beispiel präsentiert.
Danach wird eine Problematik, die während der Realisierung des UDP-Connectors entstanden ist, erläutert und ihre Lösung vorgeführt.
Zum Schluss wird der entwickelte UDP-Connector mit Hilfe von *„JUnit Testing Framework“* getestet.

Der Aufbau des Kapitels ist folgender:

- ***Schnittstellen zu einem Connector, einem Plug-In und der API***
- ***Selbst-Adaptivität der API***
- ***Verläufe im Lebenszyklus der API***
- ***Implementierung eines UDP-Connectors***
- ***Kopplungen zwischen den Komponenten der Anwendungssuite***
- ***Testmöglichkeiten***
- ***Erstellung eines Beispiel-Plug-Ins – „Hallo Welt!“***
- ***Problematik***
- ***Test von dem UDP-Connector***

4.1. Schnittstellen zu einem Connector, einem Plug-In und der API

Die Schnittstelle (siehe Abbildung 4.1) zu einem Connector hat Methoden zum Einstellen der Verbindungdaten zu einem Eye-Tracker, zum Setzen der Kopplungen zur API, zum Empfangen der Daten von einem Eye-Tracker, zum Zugriff auf die Verlaufsdaten von einem benutzten

Eye-Tracker, zum Empfangen des Namens des benutzten Connectors und zum Schließen des benutzten Connectors (wobei eine bestehende Verbindung zu einem Eye-Tracker geschlossen werden muss).

```
package multieye.connectors;

/**
 *
 * @author Alexej Tietz
 */
public interface MEConnector {
    void setConnectionProperties(String ip, int port);

    byte[] getEyeTrackerData(int dataLength) throws Exception;

    byte[][] getHistoryEyeTrackerData();

    void setParentME(multieye.MultiEye me);

    void close();

    String getTitel();
}
```

Abbildung 4.1: Interface des Connectors – „*MEConnector*“

Demnächst kommt die Schnittstelle (siehe Abbildung 4.2) zu der API – „Multi Eye“. Die Schnittstelle hat nur zwei Methoden, die der Informationsmitteilung an die API (z. B. „Plug-In XYZ wurde erfolgreich gestartet!“) und einer sauberen Beendigung eines Plug-Ins dienen (wenn ein Plug-In sich selbst beendet, müssen auch die angelegten Ressourcen des Plug-Ins entsorgt werden).

```
package multieye;

import multieye.plugins.MEPlugIn;

/**
 *
 * @author Alexej Tietz
 */
public interface MultiEye {
    void setInfo(String info);

    void killMe(MEPlugIn plugIn);
}
```

Abbildung 4.2: Interface der API – „*Multi Eye*"

Als Letztes wird die Schnittstelle (siehe Abbildung 4.3) zu einem Plug-In präsentiert, die viel komplexer als die anderen Schnittstellen ist.

```
package multieye.plugins;

import java.awt.Dimension;
import java.awt.image.BufferedImage;
import javax.swing.JFrame;
import multieye.connectors.MEConnector;

/**
 *
 * @author Alexej Tietz
 */
public interface MEPlugIn {
    void start();

    void stop();

    void setPaused(boolean pause);

    void setEnabled(boolean enabled);

    void setConnector(MEConnector c);

    void setParentME(multieye.MultiEye me);

    void setParentJFrame(JFrame jm);

    void setPlugInSize(int x, int y);

    void setPlugInSize(Dimension size);

    void resizeComponents(Dimension newDimension);

    BufferedImage getProgIcon();

    String getTitel();

    Dimension getDevelopedDimension();
}
```

Abbildung 4.3: Interface des Plug-Ins – „*MEPlugIn*"

Das Interface ist aus dem Grund komplex, da ein Plug-In die Funktionalität eines ganzen Programms enthalten kann und auch Vieles können muss. Die Schnittstelle enthält als Erstes die Methoden zum Manipulieren eines Plug-Ins, mit den es gestartet, gestoppt, pausiert oder deaktiviert werden kann. Als Zweites hat das Interface die Methoden zum Konfigurieren und Setzen von Kopplungen eines Plug-Ins. Und Zuletzt kommen Methoden, die einige grundlegende Informationen über ein Plug-In zurückgeben, sowie Entwicklungsauflösung, den Namen oder ein Icon-Bild eines Plug-Ins.

4.2. Selbst-Adaptivität der API

Unter Selbst-Adaptivität der API ist in diesem Fall zu verstehen, dass die Software sich selbst an die Umgebung anpasst, in der sie läuft. Da die Software in Java entwickelt wird, ist die Selbst-Adaptivität schon teilweise durch die Java VM (**Java V**irtual **M**achine) gegeben, sodass die Software sich selbst an das Betriebssystem anpasst. Man geht dann weiter und hat ein Problem. Die Software kann auf einem Laptop oder auf einem PC ausgeführt werden und die Problematik ist bei der Auflösung des Bildschirms des Geräts. Das Problem ist bei der API – „Multi Eye“ auf folgende Weise gelöst. Zuerst wird die Standard-Grafikkarte ermittelt. Durch sie erfährt man die aktuelle Auflösung des Bildschirms. Dann muss das Softwarefenster an die Auflösung angepasst werden. Es scheint so als wäre das Problem gelöst, ist es aber nicht. Im Applikationsfenster hat man gewöhnlich verschiedene GUI-Widgets, die als Nächstes an die Auflösung skaliert werden müssen.
Demnächst kommt ein Beispielcode (siehe Abbildung 4.4), mit dem man die Problematik in den Griff bekommt.
Die notwendigen Klassen sind in dem Beispiel importiert. Zusätzlich sind zwei Utility-Klassen aus der Anwendungssuite verwendet, die am Ende des dritten Kapitels kurz beschrieben wurden. Die Hilfsklassen erledigen das Berechnen des Skalierungsfaktors und das Skalieren der Swing-GUI-Widgets eines Applikationsfensters.

Damit der Code leichter zu verstehen wäre, werden in Folge die Klassenvariablen erläutert und was hinter den steckt:

- ***GraphicsEnvironment env*** – die Klasse, von der die Variable ist, repräsentiert eine grafische Umgebung. In dem Beispiel referenziert sie die lokale grafische Umgebung.

- ***GraphicsDevice defaultGraphicsDevice*** – die Klasse der Variable repräsentiert ein grafisches Gerät, das ein Bildschirm ober ein Drucker sein kann. In dem Beispiel steht die Variable für einen Hauptbildschirm.

- ***DisplayMode currentDisplayMode*** – die Klasse, von der die Variable ist, repräsentiert einen Bildschirmmodus, in dem ein Bildschirm arbeiten kann. In dem Beispiel enthält die Variable den aktuellen Modus, in dem ein Bildschirm zur Ausführungszeit arbeitet.

- ***Dimension developedDimension = new Dimension(1024, 768)*** – die Klasse der Variable repräsentiert eine Dimension. In dem Beispiel steht die Variable für eine Auflösung, in der die Software entwickelt wurde. Die Entwicklungsauflösung hat die Breite von 1024 Pixeln und die Höhe von 768 Pixeln.

Die Selbst-Adaptivität findet bei der Erstellung eines Objektes der API – „Multi Eye“ statt. Das kann man daran erkennen, dass die Methoden „*initGraphicsEnvironment*“ und „*resizeComponents*“ in dem Standard-Konstruktor ausgeführt werden.

```java
package multieye;

import java.awt.Dimension;
import java.awt.GraphicsDevice;
import java.awt.GraphicsEnvironment;
import multieye.utility.CompResizer;
import multieye.utility.ResizeFactor;

/**
 *
 * @author Alexej Tietz
 */
public class MultiEyeGUI extends javax.swing.JFrame implements MultiEye {
    private GraphicsEnvironment env;
    private GraphicsDevice defaultGraphicsDevice;
    private DisplayMode currentDisplayMode;
    private Dimension developedDimension = new Dimension(1024, 768);

    public MultiEyeGUI() {
        initGraphicsEnvironment();
        resizeComponents(getSize());
    }

    void initGraphicsEnvironment() {
        env = GraphicsEnvironment.getLocalGraphicsEnvironment();
        defaultGraphicsDevice = env.getDefaultScreenDevice();
        currentDisplayMode = defaultGraphicsDevice.getDisplayMode();
        setSize(currentDisplayMode.getWidth(), currentDisplayMode.getHeight());
    }

    void resizeComponents(Dimension newDimension) {
        ResizeFactor rsf = new ResizeFactor(developedDimension, newDimension);
        CompResizer.resizeFont(jTabbedPane, rsf.getFactor());
        for (Component c : jPanelConnection.getComponents()) {
            CompResizer.resizeFont(c, rsf.getFactor());
            CompResizer.resizeSurface(c, rsf.getFactorWidth(),
                    rsf.getFactorHeight());
        }
        CompResizer.resizeFont(jTextAreaLog, rsf.getFactor());
    }
}
```

Abbildung 4.4: Beispielcode der Selbst-Adaptivität an eine gegebene Auflösung

Die Methode „*initGraphicsEnvironment*“ initialisiert die Klassenvariablen: „*env*“, „*defaultGraphicsDevice*“ und „*currentDisplayMode*“, indem sie die lokale Eigenschaften des Betriebssystem abfragt. Nachdem die Variablen initialisiert sind, wird einem Applikationsfenster die aktuelle Auflösung übergeben, an die es sich anpassen muss. Die Auflösung erfährt man aus dem aktuellen Arbeitsmodus des Bildschirmes.

Wenn die aktuelle Auflösung des Bildschirmes bekannt ist, kann man mit der Methode „*resizeComponents*" die grafischen Komponenten eines Applikationsfensters an die Aufführungsdimension skalieren. Zuerst wird ein Skalierungsfaktor berechnet, das in der Variable „*rsf*" abgespeichert wird. Bei dem Skalieren von den GUI-Widgets darf man Eins nicht vergessen, dass viele grafische Komponenten eine Beschriftung oder einen Textfeld haben können, sodass die Eigenschaft auch geändert werden muss. Und zum Schluss kommt das Wichtigste: manche GUI-Widgets sowie „*JTextArea*" sind mit anderen Komponenten sowie „*JScrollPane*" umhüllt. In so einem Fall muss man die Schrift-Eigenschaft bei der Komponente ändern, die diese Eigenschaft hat und nicht bei der Umhüllungskomponente. Die Umhüllungskomponenten werden als Eltern-Komponenten angesehen und wenn man in einer „*for*"-Schleife alle GUI-Widgets von „*JFrame*" oder „*JPanel*" durchgeht, bekommt man die Eltern-Komponenten und nicht die Komponenten, die umhüllt werden.

4.3. Verläufe im Lebenszyklus der API

In diesem Teilabschnitt kommt ein Verlauf, der das Starten eines Plug-Ins im Fullscreen-Modus beschreibt. Der Verlauf besteht aus mehreren Phasen:

- Laden und Konfigurieren eines Connectors
- Laden und Konfigurieren eines Plug-Ins
- Konfigurieren eines Applikationsfensters für den Fullscreen-Modus
- Einfügen zu einem Applikationsfenster eines Plug-Ins
- Starten eines Plug-Ins

Um den nächstkommenden Code anschaulicher zu präsentieren, werden zwei erste Phasen allgemein gezeigt und zwar wie eine Instanz einer Klasse aus einem JAR-Paket mit Hilfe eines Klassenladers erstellt werden kann (siehe Abbildung 4.5).

```
package multieye.utility;

import java.net.URL;
import java.net.URLClassLoader;

/**
 *
 * @author Alexej Tietz
 */
public class ClazzLoader {
    private ClazzLoader() {
    }

    public static Object newInstanceFromJar(String fileAbsolute, String classname)
            throws Exception {
        URL url = new URL("jar:file:/" + fileAbsolute + "!/");
        URLClassLoader ucl = new URLClassLoader(new URL[]{url});
        return Class.forName(classname, true, ucl).newInstance();
    }
}
```

Abbildung 4.5: Klassenlader – Laden einer Klasse aus einem JAR-Paket

Die Methode „*newInstanceFromJar*“ aus der Klasse „*ClazzLoader*“ ist eine statische Methode, was heißt, dass die Klasse gar nicht instanziiert werden muss und direkt über den Klassennamen aufgerufen werden kann.
In der Methode wird zuerst eine spezielle URL (**U**niform **R**esource **L**ocator) erstellt, in der ausdrücklich steht, dass die übergebene Datei ein JAR-Paket ist. Danach wird eine Instanz von dem URL-Klassenlader mit der URL zu einem JAR-Paket als Parameter erstellt. Zum Schluss wird über die Klasse „*Class*“ mit der Verwendung der Methode „*forName*“ für den übergebenen Namen die Klasse geladen und über die Methode „*newInstance*“ wird eine neue Instanz der gewünschten Klasse zurückgegeben.
Man muss nur Eins berücksichtigen und zwar, dass die zurückgegebene Instanz von der Klasse „*Object*“ ist und sie muss auf die gewollte Instanz (z. B. „*MEConnector*“) gecastet werden.

Als Nächstes kommt der ganze Verlauf (siehe Abbildung 4.6), der all die Teilphasen beinhält.

```
String ia = "localhost";
int port = "10500";
MEConnector connector = (MEConnector) ClazzLoader.newInstanceFromJar(
                        "modules", "multieye.connectors.MEConnectorUDP");
connector.setConnectionProperties(ia, port);
connector.setParentME(this);
MEPlugIn plugIn = (MEPlugIn) ClazzLoader.newInstanceFromJar(
                    "modules", "multieye.plugins.MEPlugInSimpleEyeView");
plugIn.setConnector(connector);
plugIn.setParentME(this);
plugIn.setParentJFrame(this);
plugIn.setPlugInSize(getSize());
getContentPane().add((JPanel) plugIn, "Center");
((JPanel) plugIn).setLocation(new Point(0, 0));
((JPanel) plugIn).requestFocus();
plugIn.start();
```

Abbildung 4.6: Starten eines Plug-Ins im Fullscreen-Modus

Um den Code besser zu verstehen, muss man wissen, dass dieser Code sich in einer Klasse befindet, die von der Klasse „*JFrame*“ abgeleitet wird und das Interface „*MultiEye*“ der API implementiert, also „***this***“ ist der Verweis auf die Klasse. Damit das Verständnis für den Code höher wird, werden im Nächsten ein paar Variablen erklärt:

- ***String ia = "localhost"*** – die Variable ist von der Klasse „*String*“ und repräsentiert eine Zeichenkette. Sie enthält eine Adresse, zu der sich ein Connector aus dem Beispiel verbindet.

- ***int port = "10500"*** – die Variable ist von der Klasse „*Integer*“ und repräsentiert eine ganze Zahl. Sie hat den Port der Verbindung und ist ein Teil der Verbindungsidentifikation.

- ***MEConnector connector*** – die Variable ist von dem Interface „*MEConnector*” und repräsentiert eine Schnittstelle zu einem Connector. In dem Beispiel ist sie eine Schnittstelle zu einem UDP-Connector.

- ***MEPlugIn plugIn*** – die Variable ist von dem Interface „*MEPlugIn*” und repräsentiert eine Schnittstelle zu einem Plug-In. Die Schnittstelle ist zum Plug-In „Simple Eye View“.

Der Verlauf in dem Beispiel ist schon fast erklärt. Es fällt nur eine oder andere Erklärung, die man eigentlich gar nicht braucht, weil die Methoden vielsagende Namen haben. Die Erläuterungen kommen trotzdem demnächst.

Die Methoden des Connectors:
- ***setConnectionProperties*** – setzt die Verbindungsidentifikationen zu einem Eye-Tracker.
- ***setParentME*** – setzt die Referenz zu einer obenstehenden API-Instanz.

Die Methoden des Plug-Ins:
- ***setConnector*** – setzt die Referenz zu einem Connector.
- ***setParentME*** – setzt die Referenz zu einer obenstehenden API-Instanz.
- ***setParentJFrame*** – setzt die Referenz zu einer obenstehenden „*JFrame*".
- ***setPlugInSize*** – setzt die Ausführungsauflösung eines Plug-Ins.
- ***start*** – startet ein Plug-In, also nach der Ausführung der Methode wird ein Plug-In aktiviert.

Jetzt fehlt nur die Erklärung, wie das Plug-In in einem Fullscreen-Modus ausgeführt wird. Es ist auf folgende Weise gelöst. Wenn es erstellt und konfiguriert ist, wird das Plug-In zu dem Hauptpanel in der Mitte der „*JFrame*" von der API eingefügt. Zu einer kleinen Korrektur wird das Plug-In als „*JPanel*" angesprochen, dabei wird die linke Ecke des Plug-Ins auf die (0, 0)-Position im Applikationsfenster gesetzt. Nachfolgend wird das Plug-In fokussiert (Fokus wird auf „*JPanel*" eines Plug-Ins setzt), und zum Schluss wird es gestartet.

4.4. Implementierung eines UDP-Connectors

Die nächstkommende Implementierung eines UDP-Connectors (siehe Abbildung 4.7) listet nicht alle Methoden auf, die die Implementierung enthält. Die Methoden sind aus zwei folgenden Gründen nicht aufgelistet:
- Pädagogischer Grund und zwar, dass die Implementierung nach einer Erklärung einer Problematik im Teilabschnitt 4.8 fortgesetzt wird.
- Vermeidung der Auflistung von trivialen Aspekten, die eine Erklärung gar nicht benötigen.

Wenn ein alternativer Connector entwickelt werden muss, muss er die empfangenen Daten von einem Eye-Tracker auf eine bestimmte Weise formatieren. Die Daten müssen in einem Byte-Array zurückgegeben werden, das einen Integer (Nachrichtentyp), Double (X-Position von Augen), Double (Y-Position von Augen) und Integer (Latenzzeit in Millisekunden) enthält. Der Inhalt des Arrays muss in den Little Endian Format (Kapitel 2 Teilabschnitt 1) abgespeichert werden, damit der alternative Connector mit den Plug-Ins (Kapitel 5) kompatibel sein wird.
Der verwendete Eye-Tracker muss auch einigen Eigenschaften erfüllen. Er muss in einem 60Hzr-Betrieb arbeiten und die Positionierung von Augen als eine Fließkommazahl übermitteln, die in einem bestimmten Wertebereich ist. Der Wertebereich fängt bei 0,0 an und endet bei 1.0, also 0.0 wäre für X-Positionierung ganz links und 1.0 ganz rechts.
Falls ein Eye-Tracker einen anderen Aufbau der Nachricht hat und/oder die Primitiven der Nachricht andere Bytereihenfolge haben, muss die Nachricht von einem Connector in die geforderte Form konvertiert werden.

Die Implementierung hat drei Klassenvariablen, deren Bedeutung in der Folge erläutert wird:

- ***DatagramSocket dSocket*** – die Variable ist von der Klasse „*DatagramSocket*“ und repräsentiert einen Datagramm-Socket, welcher für eine UDP-Verbindung verwendet wird.

- ***DatagramPacket packet*** – die Klasse der Variable repräsentiert ein Datagramm-Paket, welcher die Nutzdaten enthält, die über eine UDP-Verbindung versendet werden.

- ***List<byte[]> history = new ArrayList<byte[]>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit einem Byte-Array typisiert. Die Liste enthält im Verlauf all die Daten, die von einem Eye-Tracker empfangen worden sind.

Die Methoden der Klasse „*MEConnectorUDP*“ (siehe Abbildung 4.7) enthalten die grundsätzliche Funktionalität und Verläufe der Klasse.
Bei dem Aufruf der Methode „*setConnectionProperties*“ wird ein Datagramm-Socket mit übergebenen Verbindungsidentifikationen erstellt, der für eine nächststehende UDP-Verbindung genutzt wird.

```java
package multieye.connectors;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 *
 * @author Alexej Tietz
 */
public class MEConnectorUDP implements MEConnector {
    List<byte[]> history = new ArrayList<byte[]>();
    DatagramPacket packet;
    DatagramSocket dSocket;

    public byte[] getEyeTrackerData(int dataLength) throws Exception {
        byte[] data = new byte[dataLength];
        packet = new DatagramPacket(new byte[dataLength], dataLength);
        try {
            dSocket.receive(packet);
            data = packet.getData();
            history.add(data);
        } catch (IOException ex) {
            parentME.setInfo(getClass().getName() + ":" + ex);
        }
        return Arrays.copyOf(data, dataLength);
    }

    public byte[][] getHistoryEyeTrackerData() {
        return (byte[][]) history.toArray();
    }

    public void setConnectionProperties(String ip, int port) {
        try {
            dSocket = new DatagramSocket(port, InetAddress.getByName(ip));
        } catch (Exception ex) {
            parentME.setInfo(getClass().getName() + ex);
        }
    }
}
```

Abbildung 4.7: Implementierung des UDP-Connectors – „*MEConnectorUDP*“

Die Methode „*getEyeTrackerData*“ ist dank der Einfachheit des Aufbaus dynamisch konzipiert und zwar, dass bei dem Aufruf die Anzahl von Bytes gesagt wird, die man empfangen will. Dadurch kann ein Connector für verschiedene Eye-Tracker ohne Codeänderungen genutzt werden, es muss nur eine andere Größe des Datagramm-Pakets in Bytes übergeben werden. Das bedeutet, dass die Komplexität in einem Plug-In steigt, welches so ein Connector verwendet, weil die Logik bezüglich des Entnehmens der Daten aus dem Datagramm-Paket in einem Plug-In enthalten sein muss.
Der Verlauf des Empfangens der Daten von einem Eye-Tracker ist ziemlich simpel. Zuerst wird ein Array für Nutzdaten erstellt und das Datagramm-Paket initialisiert. Demnächst wird mit dem Aufruf der Methode „*receive*“ das Empfangen der Daten und das Schreiben in das Datagramm-Paket initiiert. Der Aufruf blockiert bis die Daten von einem Eye-Tracker versandt sind. Wenn ein Paket mit den Nutzdaten ankommt, werden diese in ein Array geschrieben, welches für die Nutzdaten vorgesehen ist. Danach werden die Nutzdaten eines Eye-Trackers zu den Verlaufsdaten hinzufügt und eine Kopie der Daten einem Plug-In zurückgegeben, welches das Empfangen gestartet hat. Was dann mit den empfangenen Daten geschieht, ist die Sache eines Plug-Ins.

Mit der Methode „*getHistoryEyeTrackerData*“ kann man auf die Verlaufsdaten eines Eye-Trackers zugreifen, die im Laufe einer Verbindung durchgangen sind. Die Verlaufsdaten werden als ein zweidimensionales Array zurückgegeben.

ANMERKUNG: *Es war ein Filtermodul verwendet, das die relevanten Daten von einem Eye-Tracker entgegennahm, sie filterte und als UDT-Pakete versandt. Das Filtermodul verwendet die Hilfsbibliotheken, die von dem Eye-Tracker-Hersteller (Tobii Technology AB) mitgeliefert wurden und nur unter OS-„Windows“ nutzbar sind. Die Verwendung des Filtermoduls ist nur aus dem Grunde nötig, da eins der Hauptziele der Anwendungssuite OS-Plattformunabhängigkeit ist. Der Aufbau des UDP-Pakets kommt im nächsten Kapitel (Kapitel 5). Zusätzlich wurde das Filtermodul noch für die Kalibrierung des verwendeten Eye-Trackers genutzt.*
Das Filtermodul wurde von ***Lorenz Barnkow*** *bei seiner Abschlussarbeit – „****Ein Eye-Tracking-basiertes System für Usability-Untersuchungen von benutzeradaptiven TV-Newstickern****“ entwickelt. Die Arbeit wurde am 28. Juli 2009 an der HAW abgegeben und vom* ***Prof. Dr. Kai von Luck*** *und* ***Prof. Dr. Birgit Wendholt*** *betreut.*

4.5. Kopplungen zwischen den Komponenten der Anwendungssuite

Die Kopplungen zwischen den Komponenten der Anwendungssuite können am besten durch Klassendiagramme und deren darin abgebildeten Beziehungen präsentiert werden. Die nächstkommende Klassendiagramme werden drei Kopplungen zwischen den beteiligten Komponenten (API, Connector und Plug-In) darstellen. Damit die Diagramme nicht überfüllt und übersichtlich erscheinen, werden die Beziehungen im Einzelnen durchgangen. Es werden folgende Assoziationen dargestellt:

- API und Connector (siehe Abbildung 4.8)
- API und Plug-In (siehe Abbildung 4.9)
- Plug-In und Connector (siehe Abbildung 4.10)

Zusätzlich repräsentieren die Klassendiagramme die Beziehungen zwischen Interfaces und deren Implementierungen von den Komponenten der Anwendungssuite.

4.5.4. Assoziationen zwischen API und Connector

Die Beziehung zwischen den Klassen „*MultiEyeGUI*“ und „*MEConnectorUDP*“ ist beschränkt, da die Kommunikation zwischen ihnen nur über die Interfacen „*MultiEye*“ und „*MEConnector*“ erfolgt. Natürlich hat die GUI-Klasse der API vielmehr Macht über die Connector-Klasse, da sie schließlich in der „*MultiEyeGUI*“ angelegt wird und daher nur dort vernichtet werden kann. Die Connector-Klasse ist ein passives Element und kommuniziert mit der API nur im Fehlerfall, um die Fehlermeldung mitzuteilen. Außer der Erstellung und der Vernichtung der Klasse „*MEConnectorUDP*“ wird sie über die Klasse „*MultiEyeGUI*“ konfiguriert.

4.5.5. Assoziationen zwischen API und Plug-In

Die Kopplung zwischen den Klassen „*MultiEyeGUI*“ und beispielweise mit einer Klasse „*MEPlugInSimpleEyeView*“ der Plug-In-Klassen ist eine besondere Beziehung, weil die Kommunikation der Komponenten bei ihr nicht nur über die Interfacen „*MultiEye*“ und „*MEPlugIn*“ läuft. Sie läuft zusätzlich über die Klasseninstanz-Referenz der Klasse „*JFrame*“, von der die Klasse „*MultiEyeGUI*“ erbt. Es bedeutet, dass ein Plug-In den Zugriff auf das Hauptfenster der API und auf alle darunterliegenden GUI-Komponenten hat. So eine Kopplung ist gefährlich, da ein nicht erfahrener Programmierer „viel Unheil“ anrichten könnte, aber auf diese Weise kann man schnell auf die internen Eigenschaften der GUI zugreifen. Mit solch einer Bindung kann man einige Sachen (z. B. einen Tastatur-Handler oder Maus-Handler dem Hauptfenster der API anhängen) schnell erreichen, ohne dass man sich mit irgendwelchen Interfacen auseinander setzen muss. Wenn man diese Kopplung anfasst, muss man **ganz vorsichtig** damit umgehen.

4.5.6. Assoziationen zwischen Plug-In und Connector

Die Kommunikation zwischen den Klassen eines Plug-Ins z. B. „*MEPlugInSimpleEyeView*“ und der Klasse „*MEConnectorUDP*“ eines Connectors erfolgt ausschließlich nur über ein Interface „*MEConnector*“. Die Bindung ist unidirektional, da ein Connector eine passive Komponente ist. Ein Connector gibt von sich den Laut nur in einem Fehlerfall, sonst wird er nur zum Holen von den Daten eines Eye-Trackers aus einem Plug-In angestoßen.

ANMERKUNG: *Um einen schnellen Überblick über die Diagrammen zu gewähren, enthalten die nächstkommenden Klassendiagramme nicht alle Attribute und Methoden der darin befindlichen Klassen.*

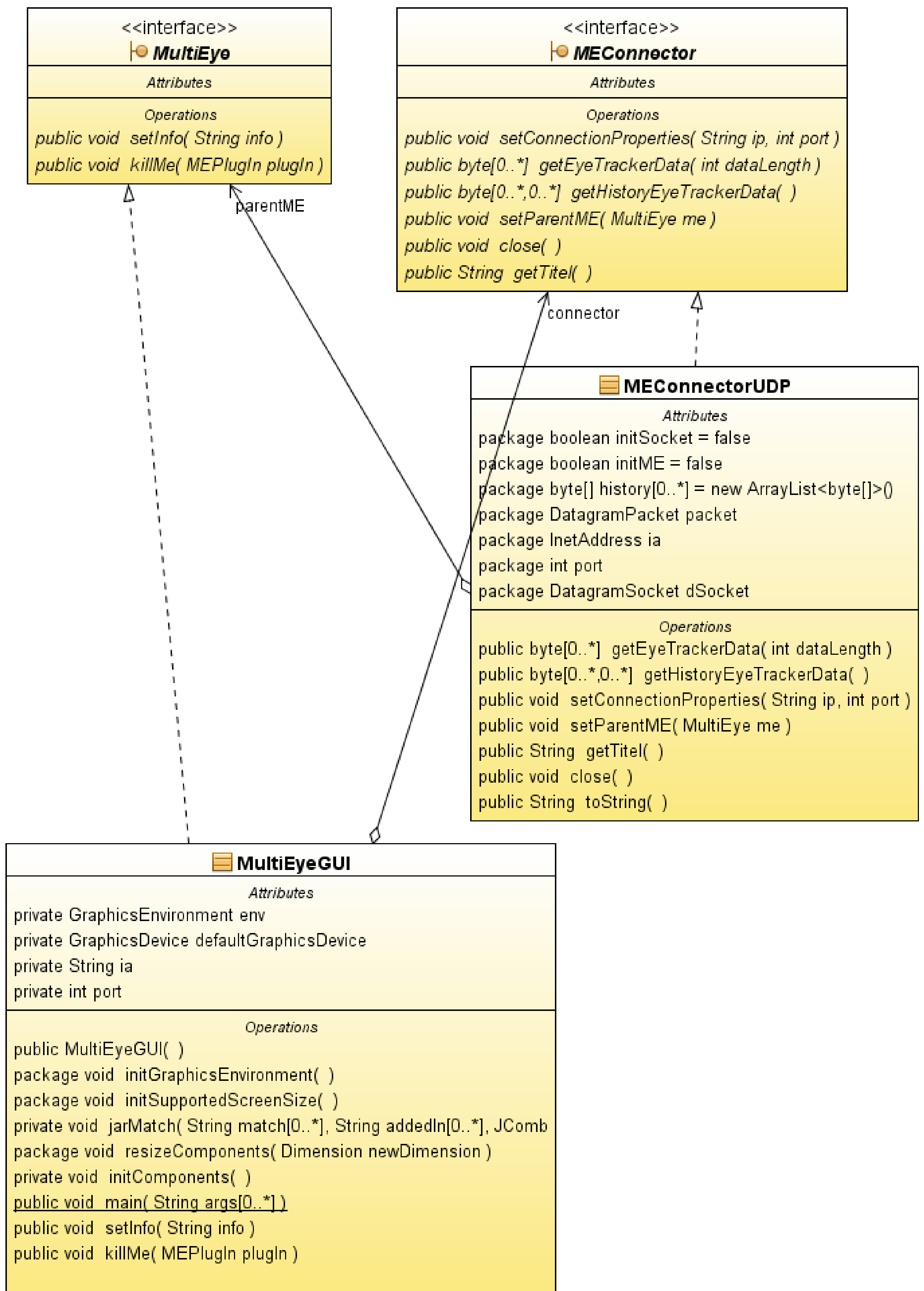


Abbildung 4.8: Klassendiagramm von API und Connector

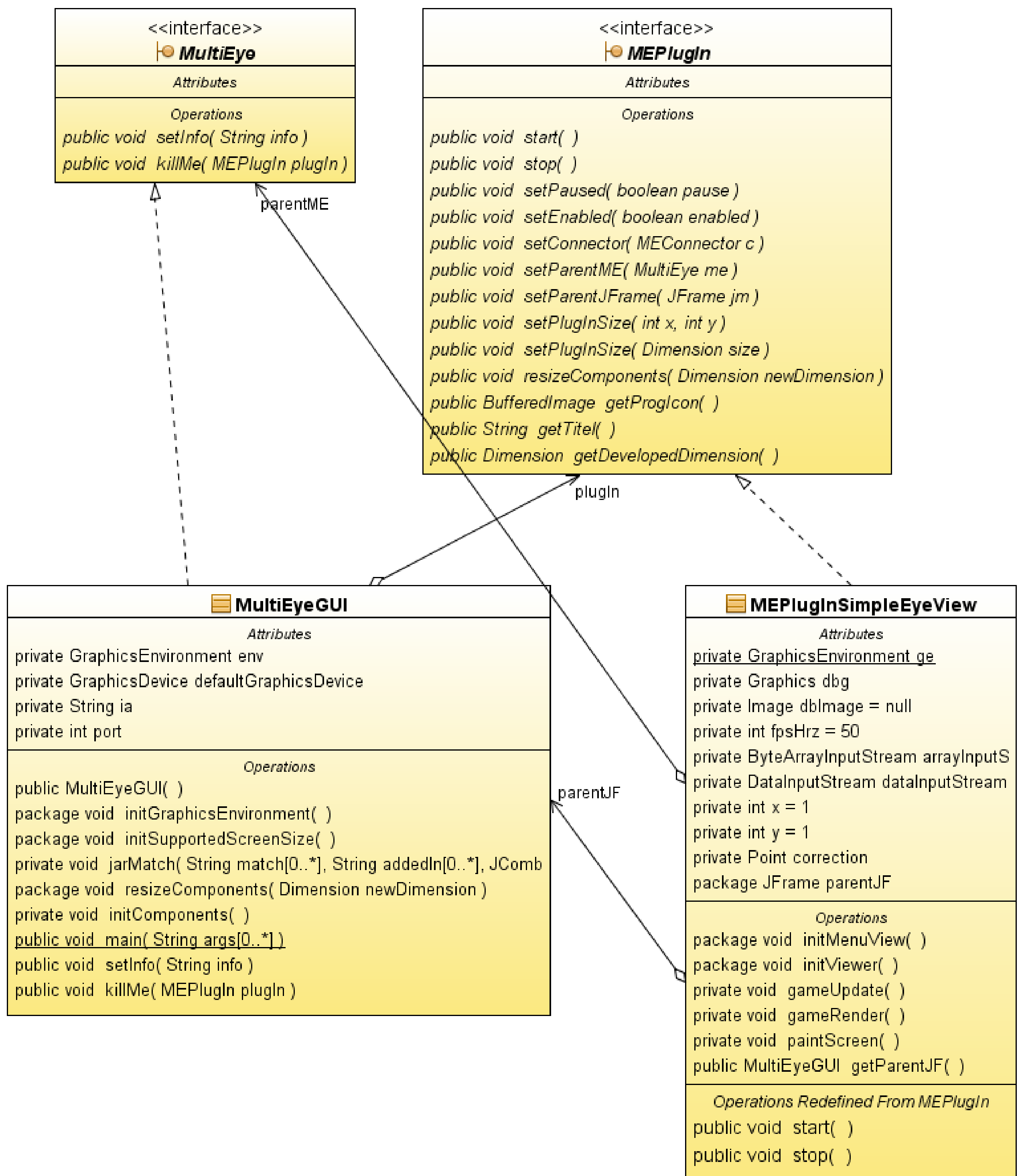


Abbildung 4.9: Klassendiagramm von API und Plug-In

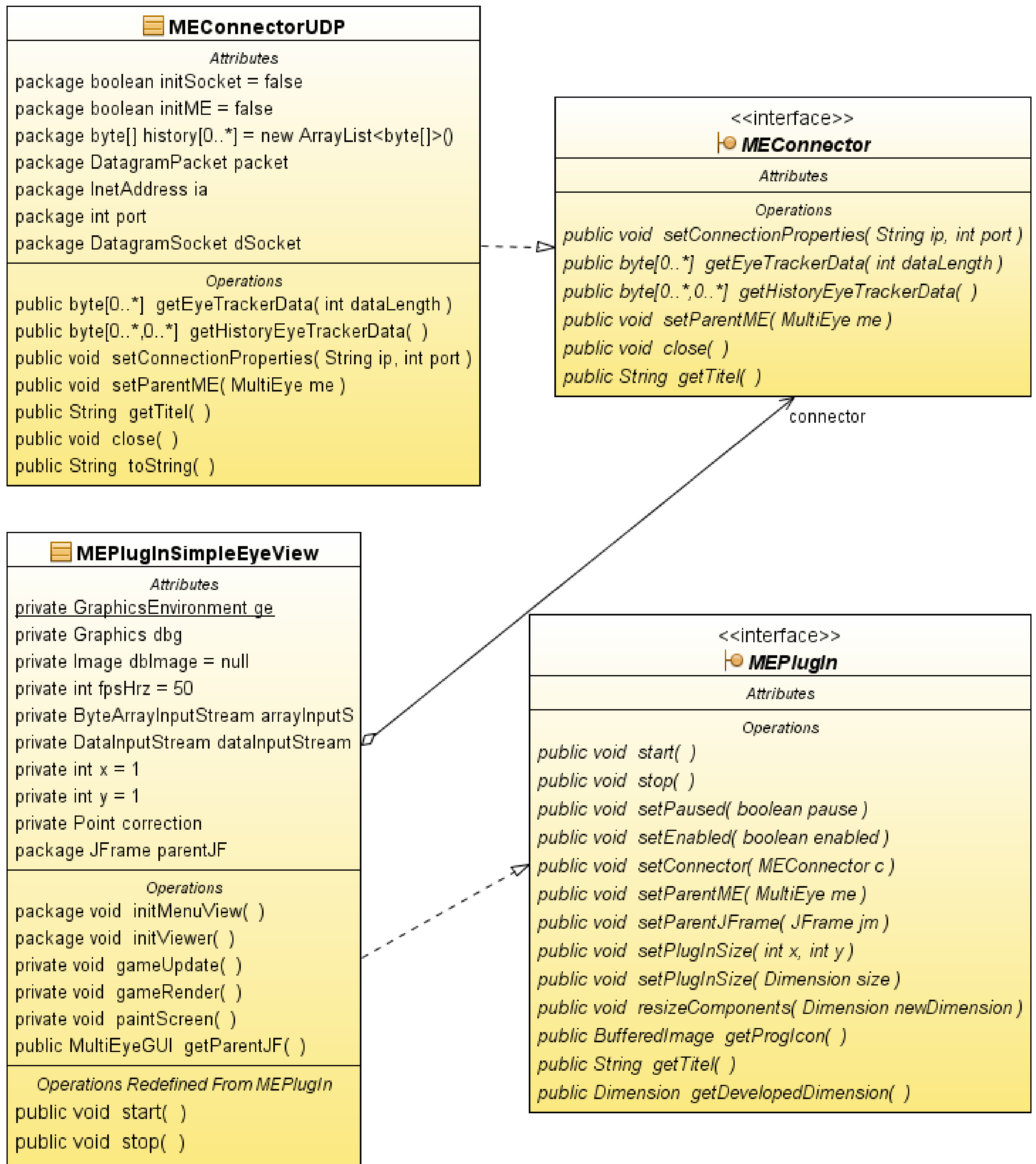


Abbildung 4.10: Klassendiagramm von Plug-In und Connector

4.6. Testmöglichkeiten

Um die Entwicklung der Komponenten der Anwendungssuite zu erleichtern, ist eine Schnittelle zu der API – „Multi Eye“ über das Interface „*MultiEye*“ für die Informationsmitteilung angeboten und die Möglichkeit ein Plug-In ohne einen Connector zu starten.
Als Erstes wird die Testmöglichkeit erläutert, wie man über das Interface „*MultiEye*“ die Komponenten der API testen kann.
Danach wird ein Testungsmuster für das Testen eines Plug-Ins zu Hause gezeigt.

Die Methode „*setInfo*“ aus dem Interface zu der API ist aus zwei Gründen da. Der erste Grund ist die Zustandsmitteilung einer Komponente, egal ob es ein Plug-In oder ein Connector ist. Der zweite Grund ist für das Testen von Komponenten, sodass man die Fehler mitteilen und mit „*Printlines*“ testen kann. In dem nächsten Screenshot werden die beiden Mitteilungstypen (siehe Abbildung 4.11) gezeigt.

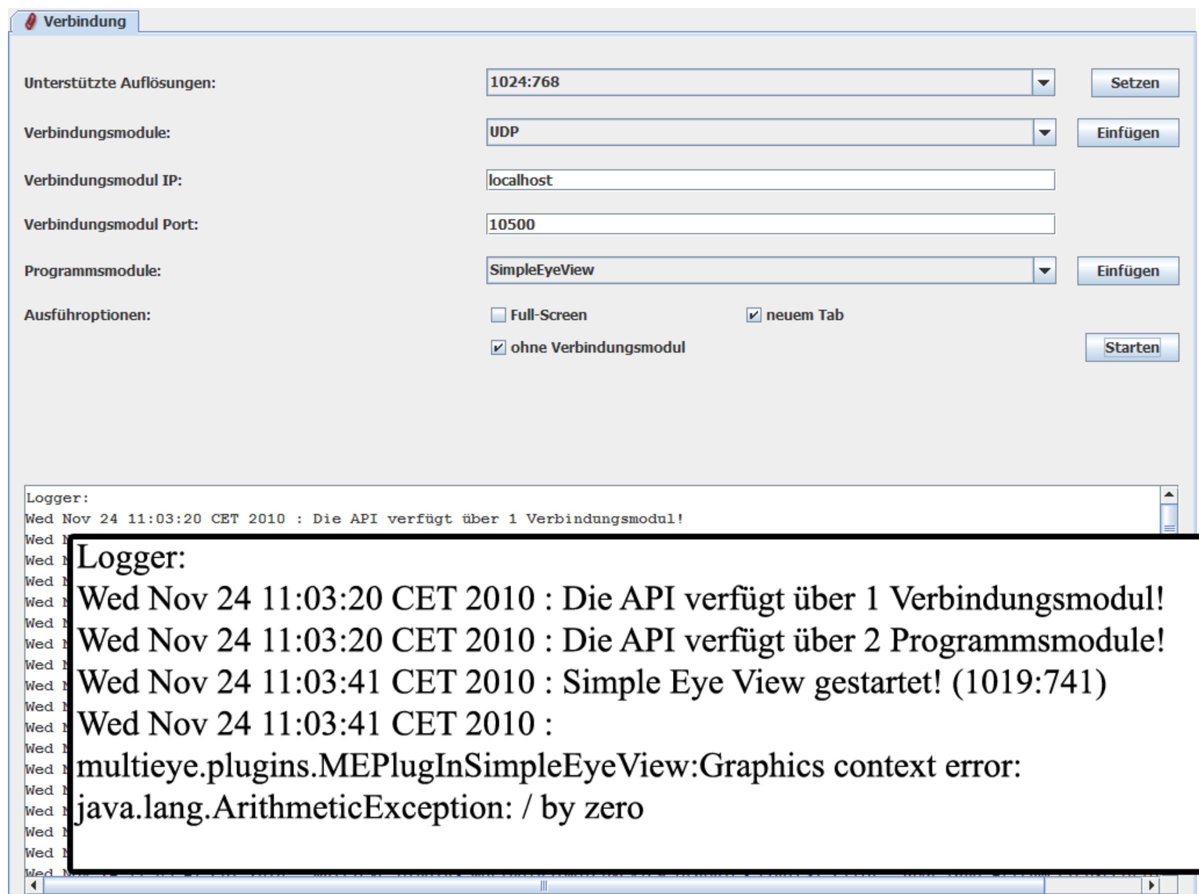


Abbildung 4.11: API – „*Multi Eye*" (das Hauptfenster – Logger)

Die erste und die zweite Mitteilung sind von der API, die nur benachrichtigen, dass ein Connector und zwei Plug-Ins in dem Programmverzeichnis gefunden sind.
Die dritte Mitteilung besagt, dass ein Plug-In namens „Simple Eye View“ mit der Auflösung (1019x741) gestartet wurde.
Die letzte Mitteilung ist eine Fehlermeldung, die Debugging -Information enthält, in der steht:

- der Verursacher der Meldung und zwar die Klasse:
- „*multieye.plugins.MEPlugInSimpleEyeView*“
- der Kontext, in dem der Fehler auftrat: „*Graphics context*“
- die Fehlermitteilung selbst: „*java.lang.ArithmeticException: / by zero*“

Um die Fehlermitteilungen richtig und verständlich der API mitzuteilen, kann man das folgende Muster (siehe Abbildung 4.12) verwenden.

```
try {
    process();//the Code, that could rise exception
    } catch (Exception ex) {
            parentME.setInfo(getClass().getName() + ":"
                            + "Error Context: " + ex);
    }
```

Abbildung 4.12: ein Muster der Fehlermitteilung an die API

Es besteht die Möglichkeit zum Testen von Plug-Ins ohne einen Eye-Tracker, wenn man bei der API den Häkchen „ohne Verbindungsmodul" setzt. Bei der Option wird die Connector-Referenz bei einem verwendeten Plug-In auf „***null***" gesetzt. Um dann ein Plug-In zu testen, muss die Referenz zu einem Connector auf „***null***" geprüft werden. Und wenn es der Fall ist, muss eine alternative Quelle verwendet werden. Eine Quelle kann z. B. eine Maus sein. Hier ist ein Beispielcode (siehe Abbildung 4.13), wie man eine alternative Quelle benutzen kann.

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

...

void setAlternative(){
    MouseAdapter testMousAdapter = new MouseAdapter() {
            @Override
            public void mouseMoved(MouseEvent e) {
                try {
                    int x = e.getX();
                    int y = e.getY();
                    aim.setPosition(new Point(x, y));
                } catch (Exception ex) {
                    ...
                }
            }
    };
    if (connector == null) {
            addMouseMotionListener(testMousAdapter);
    }
}

...
```

Abbildung 4.13: Alternative Quelle (Maus) zu einem Eye-Tracker

4.7. Erstellung eines Beispiel-Plug-Ins – „Hallo Welt!“

In diesem Teilabschnitt wird an einem Beispiel gezeigt, wie man ein einfaches Plug-In erstellen kann. Die Implementierung von einigen Methoden wird ausgeblendet, da die zu trivial sind und keine Erklärung brauchen oder weil die in vorherigen Teilabschnitten schon vorkamen.
Die Funktionalität des Plug-Ins wird simple gehalten, damit man in Kurzem schnell einen Überblick über die Implementierung eines Plug-Ins bekommt. Das Plug-In wird über keinen Connector verfügen, um die Implementierung kürzer und einfacher zu halten.

Der Code des Plug-Ins wird in fünf Sektionen geteilt, die:

- alle nötigen Importe von benutzten Klassen
- alle Klassenvariablen
- den Konstruktor
- eine Hilfsmethode
- die Methoden zum Starten und Stoppen eines Plug-Ins

enthalten.

Im Nächsten kommt ein Code-Teil, das die Importe der genutzten Klassen enthält (siehe Abbildung 4.14).

```
package multieye.plugins;

import java.awt.Button;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.image.BufferedImage;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import multieye.MultiEye;
import multieye.connectors.MEConnector;
import multieye.utility.BufImModification;
```

Abbildung 4.14: Beispiel-Plug-In – „Hallo Welt!“ Teil 1

Anhand der importierten Klassen, die in der ersten Sektion des Codes vorgekommen sind, kann man gleich feststellen, dass das Beispiel-Plug-In mit dem Java Swing-Framework entwickelt wird und dass eine Klasse der Hilfsklassen der Anwendungssuite verwendet wird.

Das Plug-In muss das Interface „*MEPlugIn*“ implementieren und von der Klasse „*JPanel*“ erben, damit es grafisch repräsentierbar ist. Die Klassenvariablen (siehe Abbildung 4.15), die das Plug-In besitzt, sind für verschiedene Bereiche (grafische Repräsentierung, Logik des Plug-Ins und Kopplungen zu anderen Komponenten) zuständig.

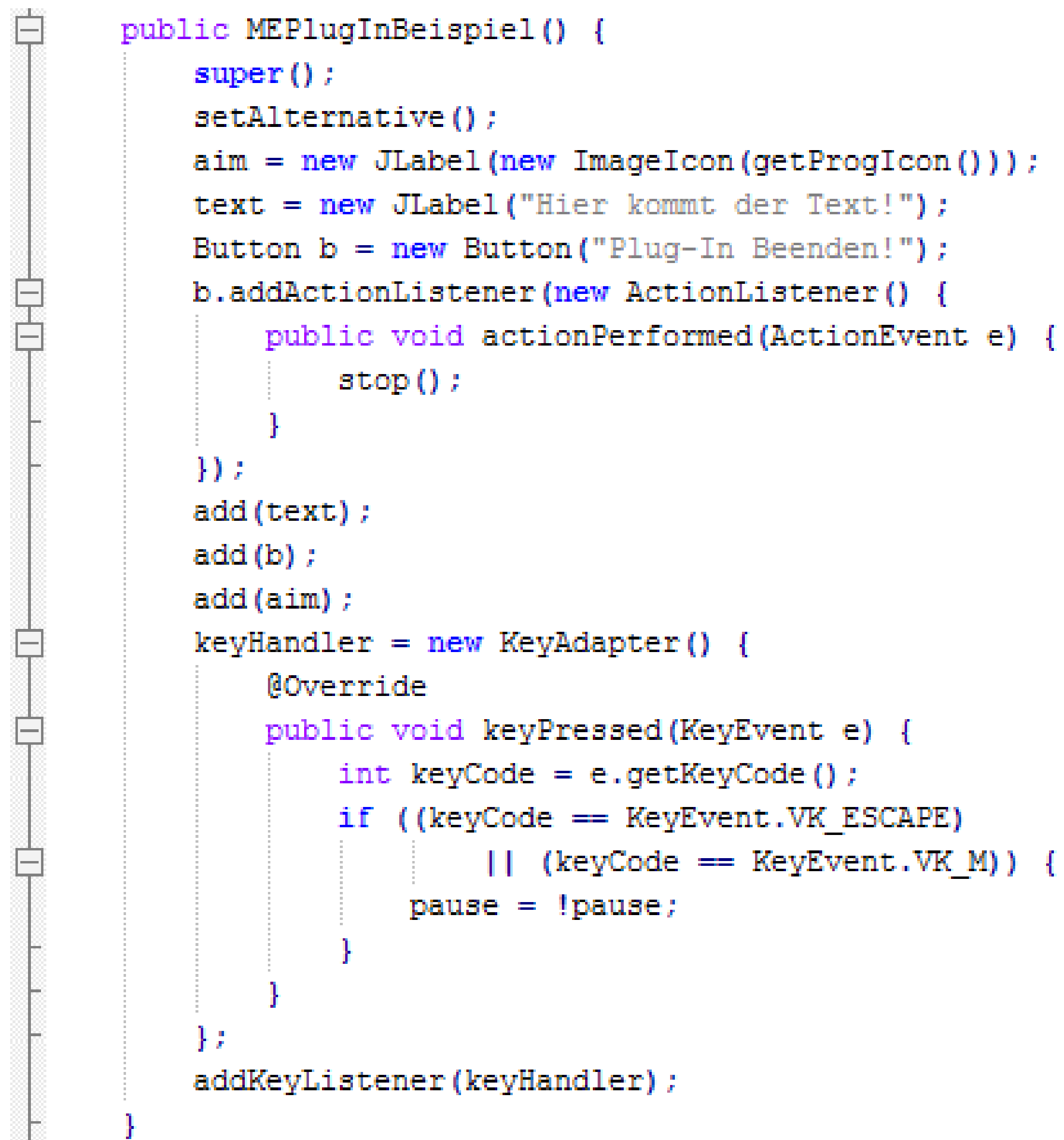
```
/**
 *
 * @author Alexej Tietz
 */
public class MEPlugInBeispiel extends JPanel implements MEPlugIn {
    private JFrame parentJF;
    private MultiEye parentME;
    private MEConnector connector;
    private JLabel aim;
    private JLabel text;
    private final MEPlugInBeispiel Beispiel = this;
    private boolean active = false;
    private boolean pause = true;
    private KeyAdapter keyHandler;
```

Abbildung 4.15: Beispiel-Plug-In – „Hallo Welt!“ Teil 2

Nachfolgend werden diese Klassenvariablen mit dessen Zugehörigkeit und Bedeutung erläutert:

- ***JFrame parentJF*** – die Variable ist von der Klasse „*JFrame*“ und repräsentiert ein Programmfenster. In diesem Fall verweist sie auf das Hauptfenster der API – „Multi Eye“.
- ***MultiEye parentME*** – die Variable ist von dem Interface „*MultiEye*“ und repräsentiert eine Schnittstelle zur API. Sie enthält die Referenz zu der API – „Multi Eye“.
- ***MEConnector connector*** – das Interface „*MEConnector*“, von dem die Variable ist, repräsentier eine Schnittstelle zu einem Connector. In dem Beispiel wird die Variable nicht genutzt, da das Plug-In ohne einen Connector ausgeführt wird.
- ***JLabel aim*** – die Variable ist von der Klasse „*JLabel*“ und kann ein Bild-Label und/oder Text-Label sein. In dem Fall repräsentiert es ein Bild, das den Punkt des Blickes zeigt.
- ***JLabel text*** – die Variable ist von der Klasse „*JLabel*“ und kann ein Bild-Label und/oder Text-Label sein. Das Label wird einen dynamischen Text enthalten, der sich abhängig von Aktionen ändern wird.
- ***final MEPlugInBeispiel Beispiel = this*** – die finale Variable ist von der Klasse „*MEPlugInBeispiel*“ des Beispiel-Plug-Ins und verweist auf das Plug-In selbst. Die Referenz muss final sein, da sie bei einigen anonymen Klassen in der Implementierung genutzt wird.
- ***boolean active = false*** – die Klasse der Variable repräsentiert einen booleschen Wert, der „***wahr***“ oder „***nicht wahr***“ sein kann. Sie zeigt die Aktivität des Plug-Ins und zwar „***true***“ – Plug-In läuft, „***false***“ Plug-In beendet oder noch nicht ausgeführt.
- ***boolean pause = true*** – die Klasse der Variable repräsentiert einen booleschen Wert, der „***wahr***“ oder „***nicht wahr***“ sein kann. Die Variable zeigt das Pausieren des Plug-Ins.
- ***KeyAdapter keyHandler*** – die Variable ist von der Klasse „*KeyAdapter*“ und repräsentiert einen Tastatur-Handler. Der Handler wird aufgerufen, wenn eine Taste gedrückt wird. In dem Beispiel dient er für das Pausieren des Plug-Ins.

Als Nächstes kommt der Konstruktor (siehe Abbildung 4.16) der Klasse des Beispiel-Plug-Ins. Bei dem Konstruktor werden die grafischen Komponenten und verschiedene Handler initialisiert. Als Erstes wird der Konstruktor der Super-Klasse aufgerufen, von der das Plug-In erbt, dadurch wird die Super-Klasse initialisiert. Als Nächstes wird die uns bekannte Methode „*setAlternative*“ aufgerufen, die im vorherigen Teilabschnitt schon vorgekommen ist. Danach kommt die Initialisierung der grafischen Komponenten: ein Bild-Label, ein Text-Label und ein Button (der das Plug-In beenden kann). Der Button beendet das Plug-In, indem bei dem Mausklick-Handler des Buttons die Methode „*stop*“ des Interfaces von dem Plug-In aufgerufen wird. Nachfolgend werden all die Grafikelemente dem „*JPanel*“ eingefügt, das eine optische Repräsentierung des Plug-Ins ist. Zum Schluss wird ein Tastatur-Handler erstellt und dem „*JPanel*“ als ein Tasten-Lauscher eingefügt, der bei dem Drücker von Tasten „*Esc*“ oder „*M*“ das Plug-In pausieren oder fortsetzen wird.



```
public MEPlugInBeispiel() {
    super();
    setAlternative();
    aim = new JLabel(new ImageIcon(getProgIcon()));
    text = new JLabel("Hier kommt der Text!");
    Button b = new Button("Plug-In Beenden!");
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            stop();
        }
    });
    add(text);
    add(b);
    add(aim);
    keyHandler = new KeyAdapter() {
        @Override
        public void keyPressed(KeyEvent e) {
            int keyCode = e.getKeyCode();
            if ((keyCode == KeyEvent.VK_ESCAPE)
                    || (keyCode == KeyEvent.VK_M)) {
                pause = !pause;
            }
        }
    };
    addKeyListener(keyHandler);
}
```

Abbildung 4.16: Beispiel-Plug-In – „Hallo Welt!“ Teil 3

Die Hilfsmethode „*setPosition*“ (siehe Abbildung 4.17) wird das Setzen der Position des Bild-Labels übernehmen, das den Punkt des Blickes zeigt. Das Setzen der Position des Bild-Labels soll nur dann erfolgen, wenn das Plug-In nicht pausiert ist.

```
void setPosition(Point p) {
    if (!pause) {
        aim.setLocation(p.x - aim.getWidth()
                / 2, p.y - aim.getHeight() / 2);
    }
}
```

Abbildung 4.17: Beispiel-Plug-In – „Hallo Welt!“ Teil 4

Als Letztes wird die Implementierung der Methoden „*start*“ und „*stop*“ (siehe Abbildung 4.18) des Interfaces „*MEPlugIn*“ gezeigt, die für das Starten und Stoppen des Plug-Ins verantwortlich sind.

```
public void start() {
    active = true;
    pause = false;
    new Thread() {
        @Override
        public void run() {
            parentME.setInfo(getTitel() + " gestartet! (" + Beispiel.getWidth()
            + ":" + Beispiel.getHeight() + ")");
            while (active) {
                if (getComponentAt(aim.getX() + aim.getWidth() / 2,
                        aim.getY() + aim.getHeight() / 2) == text
                        && !pause) {
                    text.setText("Hallo Welt!");
                } else if (text.getText().matches("(Pa.*)|(Ha.*)") && !pause) {
                    text.setText("Hier kommt der Text!");
                } else if (pause) {
                    text.setText("Pause!");
                }
                try {
                    Thread.sleep(200);
                } catch (InterruptedException ex) {
                    parentME.setInfo(getClass().getName() + ":"
                            + "Condition Thread Error: " + ex);
                }
            }
        }
    }.start();
}

public void stop() {
    active = false;
    parentJF.removeKeyListener(keyHandler);
    parentME.killMe(this);
}
}
```

Abbildung 4.18: Beispiel-Plug-In – „Hallo Welt!“ Teil 5

Bei der Methode „*start*“ wird ein Thread gestartet, der periodisch jede 200ms überprüft:

- ob das Bild-Label über dem Text-Label ist und wenn das Plug-In nicht pausiert ist, wird der Text in dem Text-Label auf „Hallo Welt!“ gesetzt.
- wenn das Bild-Label nicht über dem Text-Label ist und Plug-In nicht pausiert ist, und der Text in dem Text-Label mit „Ha“ für „Hallo Welt!“ oder mit „Pa“ für „Pause!“ beginnt, wird der Text auf Standardtext „Hier kommt Text!“ geändert.
- wenn das Plug-In pausiert ist, wird dem Text-Label der Text „Pause!“ zugewiesen.

Bei dem Starten des Threads wird der API mitgeteilt, das Plug-In gestartet wurde.

Die Methode „*stop*“ setzt die Bedingung der kopfgesteuerten Schleife aus dem Thread auf „***nicht wahr***“ und beendet damit den Thread, der in der Methode „*start*“ gestartet wurde. Danach wird der Tastatur-Handler der „*JFrame*“ des Hauptfensters der API – „Multi Eye“ entfernt. Zum Schluss wird die API gebeten, das Plug-In zu entfernen, da das Plug-In nicht mehr aktiv ist.

4.8. Problematik

Bei der Verwendung eines UDP-Connectors könnte die Methode „*close*“ (siehe Abbildung 4.19), die in dem vorherigen Teilabschnitt (4. Implementierung eines UDP-Connectors) ausgeblendet war, zu einem Deadlock führen. Die Implementierung der Methode erscheint auf den ersten Blick logisch und richtig zu sein, aber im folgenden Szenario führt ihre Benutzung zu einem Deadlock.

```
public void close() {
    if (dSocket != null) {
        dSocket.disconnect();
        dSocket.close();
    }
}
```

Abbildung 4.19: UDP-Connector - Methode „*close*"

Wenn man beim Starten eines Plug-Ins die falsche Verbindungsparameter eingibt, wird der verwendete Connector an einem falschen Socket lauschen. Und bei dem nächsten Schritt passiert der Deadlock. Man bemerkt, dass die Verbindungsdaten inkonsistent sind und dann wird versucht, ein Plug-In zu beenden. Egal ob ein Plug-In über die API oder über das Menü eines Plug-Ins zum Beenden gebracht wird, führt der Versuch zu einem Deadlock.
Um die Problematik zu verstehen, müssen einige Hintergründe erklärt werden.
Beim Starten eines Plug-Ins wird ein übergebener Connector quasi sofort zum Holen der Daten von einem Eye-Tracker verwendet. Die Methode „*getEyeTrackerData*“ eines UDP-Connectors ist blockierend, da bei ihr die blockierende Methode des Datagramm-Sockets „*receive*“ zum Holen eines Datagramm-Pakets aufgerufen wird. Dass das ganze blockiert, bemerkt man gar nicht, da das Empfangen der Daten von einem Eye-Tracker in einem separaten Thread passiert.

Beim Beenden eines Plug-Ins wird die API über die Methode „*killMe*“ die Aufräumungsarbeiten starten, zu den auch das Schließen eines Connectors gehört. Und wie man bei der Methode „*close*“ eines UDP-Connectors sieht, wird die Methode „*disconnect*“ des Datagramm-Sockets aufgerufen, die eine Verbindung nach der Vollendung des aktuellen Transfers trennt. Also ist die Methode „*close*“ eines UDP-Connectors auch blockierend und da die Daten von einem Eye-Tracker nicht empfangen werden können, blockiert der Thread eines Plug-Ins die API und das Ganze „*steht still*“.
Die Problematik hat zwei Lösungen:

- Den Thread mit einem Interrupt aufwecken.
- Den Datagramm-Socket schließen ohne ihn zu trennen.

Die beiden Lösungen führen zur Fehlermeldung und sie muss aufgefangen werden. Eine Fehlermeldung muss in dem Thread und die andere bei dem Connector aufgefangen werden.
Im Nächsten wird die Lösung (siehe Abbildung 4.20) präsentiert, bei der der Datagramm-Socket geschlossen wird. Die Lösung wird aus dem Grund gewählt, um nicht alles bei einem Plug-In machen zu müssen. Die Anpassungen im Code eines UDP-Connectors müssen an zwei Stellen vorgenommen werden. Die Methode „*close*“ muss den Datagramm-Socket sofort schließen, ohne ihn zu trennen. Die Methode „*getEyeTrackerData*“ muss die Fehlermeldung an die API nicht weiterleiten, da schließlich die API sie selbst anstoßt.

```java
public byte[] getEyeTrackerData(int dataLength) throws Exception {
    if (!initSocket || !initME) {
        throw new Exception("Setze Verbindungsdaten/MultiEye-Oberinstanz!");
    }
    byte[] data = new byte[dataLength];
    packet = new DatagramPacket(new byte[dataLength], dataLength);
    try {
        dSocket.receive(packet);
        data = packet.getData();
        history.add(data);
    } catch (IOException ex) {
        if (!ex.toString().startsWith("java.net.SocketException: socket closed")) {
            parentME.setInfo(getClass().getName() + ":" + ex);
            Logger.getLogger(getClass().getName()).log(Level.SEVERE, null, ex);
        }
    return Arrays.copyOf(data, dataLength);
    }
}

public void close() {
    if (dSocket != null) {
        dSocket.close();
    }
}
```

Abbildung 4.20: UDP-Connector - Lösung des Deadlocks

4.9. Test von dem UDP-Connector

Jede Software muss getestet werden, sonst kann man die gar nicht als qualitative Software bezeichnen, sondern „*grüne Bananen*“.

In diesem Teilabschnitt wird beschrieben wie der gefertigte UDP-Connector „*MEConnectorUDP*“ getestet wurde. Es wurden nur Grundfunktionen des Connectors getestet und zwar die Funktionalitäten, die hinter den Methoden „*getEyeTrackerData*“ und „getHistoryEyeTrackerData“ stecken.
Da der Connector ausschließlich nur die UDP-Pakete empfängt, wird das Testen nicht ohne Tricks klappen. Zum Testen wird ein UDP-Sender benötigt, der UDP-Testpakete für den UDP-Connector versenden wird.
Da der Connector in Java geschrieben ist, kann man „*JUnit Testing Framework*“ zum Testen von den Methoden des UDP-Connectors verwenden. Dank der Einfachheit des Aufbaus von dem Interface „*MEConnector*“, das von dem UDP-Connector „*MEConnectorUDP*“ implementiert wird, wird das Testen von ihm ein Kinderspiel sein. Also zwei Mal überlegen bei der Planung, lohnt sich zwei Mal bei der Entwicklung und bei dem Testen.

Der grundsätzliche Testablauf ist der Folgende:

- Der Connector wird konfiguriert, so dass er lokale UDP-Pakete empfangen kann.
- Ein UDP-Testsender versendet ein paar UDP-Pakete.
- Der Connector wird zum Empfangen von den gesendeten UDP-Paketen angestoßen.
- Der Inhalt der empfangenen UDP-Pakete wird mit dem Ursprungsinhalt verglichen.
- Ergebnisse des Tests entscheiden dann über die Korrektheit der Implementierung.

Demnächst kommt ein Codeabschnitt von dem Test und von dem UDP-Testsender (siehe Abbildung 4.21). Die eigentliche Testmethode heißt „*testMEConnectorUDP*“ und sie hat eine Annotation „*@Test*“, die sie als Testmethode markiert.
Der Konstruktor der Klasse „*MEConnectorUDPTest*“ initialisiert den UDP-Connector.
Die Methode „*sendData*“ sendet ein Bytearray als ein UDP-Paket, das von dem UDP-Connector empfangen wird.
Die Methode „*closeConnection*“ schließt den Datagramm-Socket, der zum versenden von UDP-Paketen verwendet wird.
Zwei Methoden der Klasse „*MEConnectorUDPTest*“ wurden ausgeblendet, die nichts mit dem Test zu tun haben. Die Klasse „*MEConnectorUDPTest*“ implementiert diese Methoden, weil der UDP-Connector für seine Konfigurierung eine Instanz der API – „*MultiEye*“ verlangt.
Die Methoden, deren Namen mit „assert“ beginnen, sind die Testmethoden, mit den man die Testbedingungen aufstellen kann.

```java
public class MEConnectorUDPTest implements MultiEye {
    private MEConnectorUDP connector;
    private int port = 0xDEAD;
    private byte[] message = "Text String!".getBytes();
    private DatagramSocket dsocket;

    public MEConnectorUDPTest() {
        connector = new MEConnectorUDP();
        connector.setParentME(this);
        connector.setConnectionProperties(null, port);
    }

    private void sendData(byte[] data) {
        try {
            InetAddress address = InetAddress.getByName("localhost");
            DatagramPacket packet = new DatagramPacket(data, data.length,
                    address, port);
            dsocket = new DatagramSocket();
            dsocket.send(packet);
        } catch (Exception e) {
            System.err.println(e);
        }
    }

    private void closeConnection() {
        if (dsocket != null) {
            dsocket.disconnect();
            dsocket.close();
        }
    }

    @Test
    public void testMEConnectorUDP() throws Exception {
        System.out.println("getEyeTrackerData & getHistoryEyeTrackerData");
        sendData(message);
        sendData(message);
        byte[] result = connector.getEyeTrackerData(message.length);
        result = connector.getEyeTrackerData(message.length);
        byte[][] result2 = connector.getHistoryEyeTrackerData();
        closeConnection();
        assertArrayEquals(message, result);
        assertEquals(new String(message), new String(result));
        assertArrayEquals(new byte[][]{message,message}, result2);
        assertEquals(new String(message), new String(result2[0]));
    }
}
```

Abbildung 4.21: Testklasse für den UDP-Connector „*MEConnectorUDP*“

Nach der Ausführung der Testklasse „*MEConnectorUDPTest*“ bekommt man das folgende Ergebnis (siehe Abbildung 4.22). Das Testergebnis besagt, dass der Test erfolgreich durchgeführt wurde und 125ms dauerte.

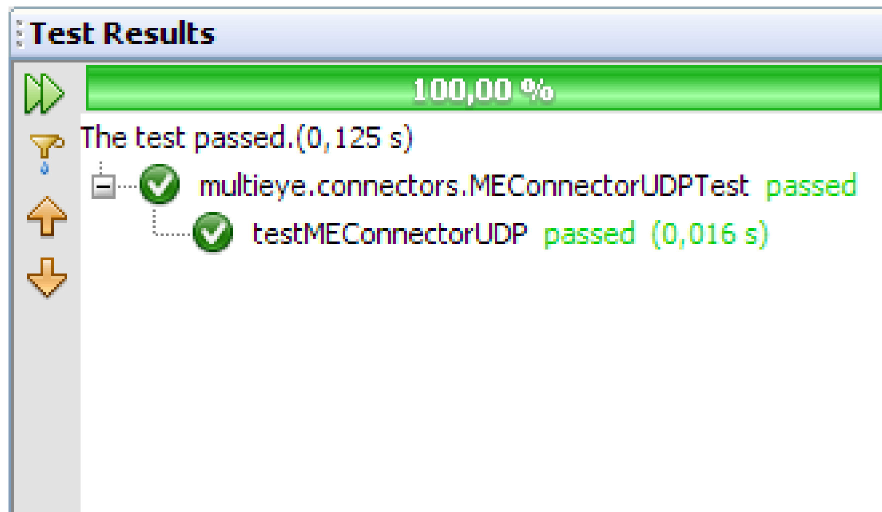


Abbildung 4 22: Testergebnis der Testklasse „*MEConnectorUDPTest*“

In dem folgenden Kapitel (Kapitel 5) wird vor allem ein grundlegender Aufbau eines 2D-Plug-Ins gezeigt. Nächstkommend wird ein grundlegendes multifunktionales Objekt der 2D-Welt zur Schau gestellt, das bei den beiden gefertigten Plug-Ins verwendet wird. Es wird der Aktivierungsmanager präsentiert, der die Benutzeraktionen erkennt und darauf reagiert. Nachdem alle Allgemeinheiten bekannt sind, kommen die gefertigten Plug-Ins mit dessen Realisierung.

5. Plug-Ins

Dieses Kapitel enthält das Konzept und die Realisierung von zwei folgenden angefertigten Plug-Ins:

- ***„Sky Defendor"*** - ein 2D-Spiel
- ***„Simple Eye View"*** - ein Dokument-Reader für Texte

Die beiden Plug-Ins sind strukturell ähnlich aufgebaut. Sie benutzen die gleichen Klassen und haben im Grunde genommen die gleichen Verläufe im Lebenszyklus des Programms. Deshalb bietet es sich an, die strukturellen Gleichheiten als Erstes zu präsentieren, um eine kleine Übersicht über den Aufbau eines 2D-Plug-Ins zu verschaffen. Im Anschluss wird ein Manager gezeigt, der auf Benutzerinteraktionen achtet und darauf reagiert. Zum Schluss werden die Plug-Ins zusammen mit dem Konzept ihres spezifischen Aufbaus und ihrer Codeumsetzung präsentiert.

Gliederung von Teilabschnitten des Kapitels:

- ***Grundlegender Aufbau eines 2D-Plug-Ins***
- ***Grundlegendes multifunktionales Objekt der 2D-Welt***
- ***Aktivierungsmanager***
- ***Plug-In: „Sky Defendor" (2D-Spiel)***
- ***Plug-In: „Simple Eye View" (Textdokument-Reader)***

5.1. Grundlegender Aufbau eines 2D-Plug-Ins

Ein 2D-Plug-In unterscheidet sich von einem einfachen Plug-In dadurch, dass bei einem 2D-Plug-In der Entwickler für eine grafische Repräsentierung eines Plug-Ins verantwortlich ist. Also muss das Zeichnen der Oberfläche eines Plug-Ins von dem Entwickler gemacht und angestoßen werden.
Der Aufbau eines 2D-Plug-Ins ist ähnlich einem einfachen Plug-In - es muss das Interface „*MEPlugIn*" implementieren und von der Klasse „*JPanel*" erben. Der Unterschied liegt darin, dass ein Animationsthread das optische Aussehen eines Plug-Ins zeichnen muss. Der Thread enthält eine Animationsschleife (Kapitel 2 Teilabschnitt 2), in der das Aktualisieren von Zuständen der 2D-Objekte und das Zeichnen der Repräsentation stattfinden.
Es gibt zwei aktive Komponenten in einem 2D-Plug-In. Die Erste ist ein Animationsthread und die Zweite ist ein Datagetterthread, der für das periodische Holen von Daten eines Eye-Trackers verantwortlich ist.

Demnächst kommt ein Zustandsdiagramm (siehe Abbildung 5.1), das den grundlegenden Lebenszyklus eines 2D-Plug-Ins darstellt. Das Diagramm entspricht dem allgemeinen Aufbau eines 2D-Plug-Ins und ist annähernd gleich dem Lebenszyklus der angefertigten Plug-Ins, deren Implementierungen in nächsten Teilabschnitten des Kapitels kommen.

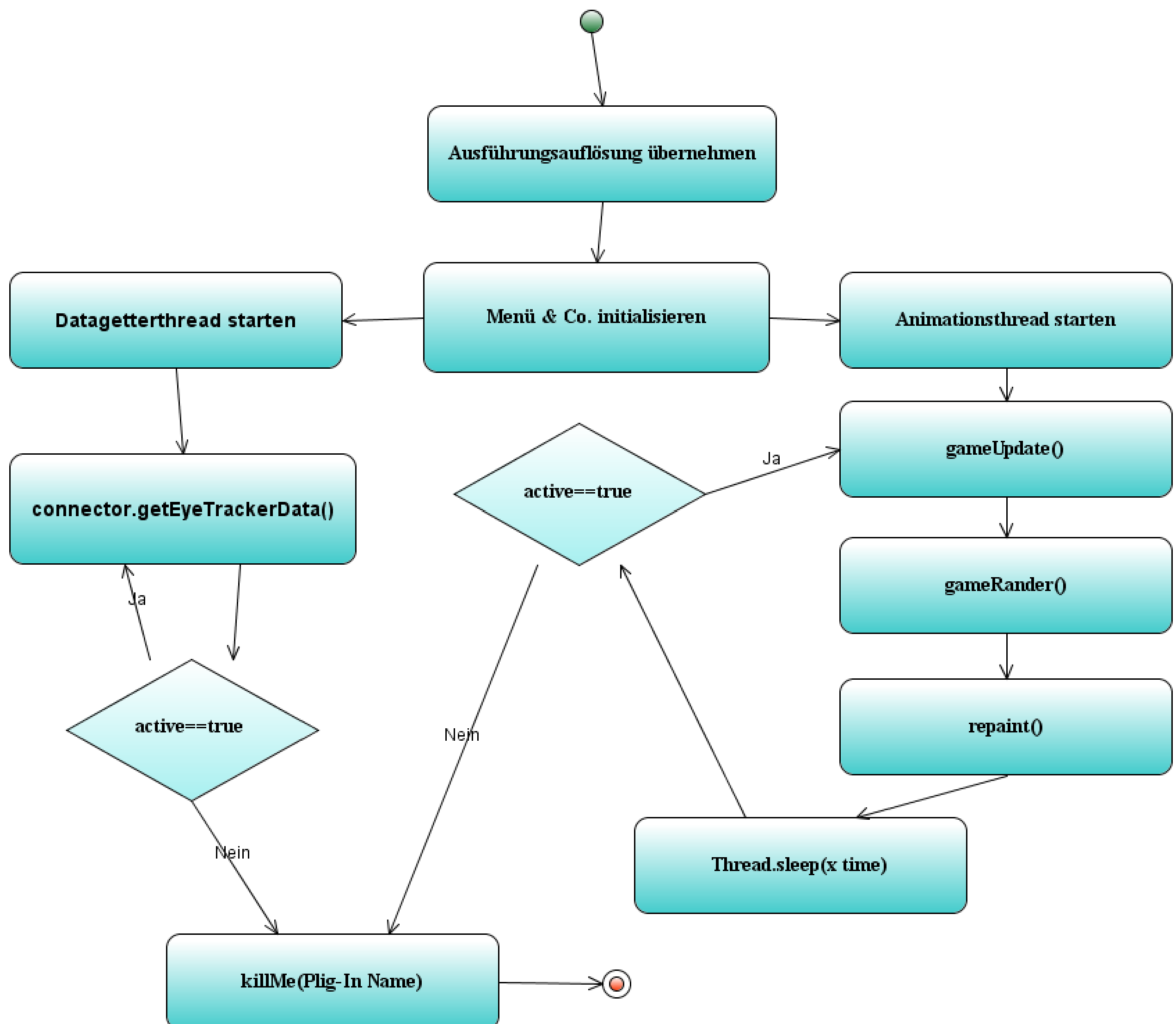


Abbildung 5.1: Zustandsdiagramm eines 2D-Plug-Ins

In dem Zustandsdiagramm sieht man, dass der Animationsthread schlaffen gelegt wird und der Datagetterthread nicht. Es bedarf einer Erklärung, warum der Datagetterthread nicht schlaffen muss. Der genutzte Eye-Tracker arbeitet in einem 60Hz-Modus (Kapitel 3 Teilabschnitt 2) und die Methode „*getEyeTrackerData()*“ des UDP-Connectors „*MEConnectorUDP*“ ist eine blockierende Methode. Daraus folgt, dass der Datagetterthread ca. jede 16ms ($1\ /\ 60Hz \approx 16,7ms$) ein UDP-Datagramm-Paket empfängt. Während er auf ein Datenpaket von einem Eye-Tracker wartet, wird er von dem Betriebssystemscheduler aus dem Zustand „***running***“ in den Zustand „***blocked***“ überführt, wo ein Thread keine CPU-Zeit beansprucht. Doch wenn ein UDP-Datagramm-Paket ankommt, wird der Datagetterthread aufgeweckt und ist wieder aktiv.
Der Datagetterthread ist wegen seines Aufbaus von der Befehlskette für die CPU eines Computers nicht lästig und aus diesem Grund muss er nicht manuell vom Entwickler eines 2D-Plug-Ins schlaffen gelegt werden.

5.2. Grundlegendes multifunktionales Objekt der 2D-Welt

Ein Objekt einer 2D-Welt besteht normalerweise aus zwei Ebenen und zwar:
- ***Logische Ebene***, die Zustände eines 2D-Objekts und dessen logische Repräsentierung enthält.
- ***Optische Repräsentierung***, die optisch den Zustand eines 2D-Objekts repräsentiert.

Ein multifunktionales Objekt einer 2D-Welt muss Vieles können, sonst hieße es nicht multifunktional.
Hier sind die Funktionalitäten aufgelistet, die das 2D-Objekt der 2D-Welt in den angefertigten Plug-Ins bietet:
- Ein 2D-Objekt kann eine Animation sein.
- Ein 2D-Objekt kann als eine Schaltfläche verwendet werden.
- Ein 2D-Objekt kann sich in der 2D-Welt bewegen und könnte dadurch eine Spielfigur sein.
- Ein 2D-Objekt kann einen begrenzten Lebenszyklus haben.
- Ein 2D-Objekt kann einen Verweilungszyklus haben.

Dieses Objekt hat die Zugriffsmöglichkeiten für ein Plug-In über das Interface „*Animatable*“ (siehe Abbildung 5.2), das auch ein bestimmtes Verhalten fordert. Das Interface bietet Methoden zum Konfigurieren des 2D-Objekts, zum Abfragen von Zuständen und Eigenschaften des 2D-Objekts und zum Manipulieren der Animation des 2D-Objekts. Als Nächstes werden die Methoden des Interfaces und dessen Bedeutung kurz erläutert.
Die Methoden zum:
- **Konfigurieren des 2D-Objekts**:
  - ***setDying*** – setzt die Zeit des Verweilens des 2D-Objekts in Millisekunden.
  - ***setLifeCycle*** – setzt die Anzahl der Teilzyklen über die Lebensdauer des 2D-Objekts.
  - ***setUpdateRate*** – setzt die Anzahl der Aktualisierungen, bis ein nächstes Bild der Animation des 2D-Objekts erscheint.
  - ***setPosition*** – setzt die Position des 2D-Objekts in der 2D-Welt des Plug-Ins.
  - ***setAnimationLine*** – setzt die Animationslinie des 2D-Objekts.
  - ***setSize*** – setzt die Größe der Bilder der Animationslinie des 2D-Objekts.
  - ***setMovementFunktion*** – setzt die Bewegungsfunktion des 2D-Objekts.
  - ***setMovementFunktionParam*** – setzt die Parameter (Startwert, Endwert und Schrittwert) der Bewegungsfunktion des 2D-Objekts.
- **Abfragen von den Zuständen und den Eigenschaften des 2D-Objekts**:
  - ***isDying*** – überprüft, ob das 2D-Objekts gerade verweilt.
  - ***isDead*** – überprüft, ob das 2D-Objekts bereits verweilt ist.
  - ***isFuncValAtEnd*** – überprüft, ob die Bewegungsfunktion den Endwert der Funktion erreicht hat.
  - ***getActualImage*** – fragt das aktuelle Bild der Animationslinie des 2D-Objekts ab.
  - ***getActualPosition*** – fragt die aktuelle Position des 2D-Objekts in der 2D-Welt des Plug-Ins ab.
  - ***getGeomethry*** – fragt das Polygon des 2D-Objekts ab.
  - ***getAnimationLineLength*** – fragt die Länge der Animationslinie des 2D-Objekts ab.
  - ***getAnimationLine*** – fragt die Animationslinie des 2D-Objekts ab.
- **Manipulieren der Animation des 2D-Objekts**:
  - ***nextStep*** – aktualisiert den aktuellen Wert der Bewegungsfunktion und den Aktualisierungszähler der Animationslinie des 2D-Objekts.
  - ***resetAnimation*** – startet die Animationslinie des 2D-Objekts vom Anfang an.
  - ***resumeAt*** – setzt die Animationslinie des 2D-Objekts an dem angegebenen Index des Bildes fort.

```java
package multieye.plugins;

import java.awt.Dimension;
import java.awt.Point;
import java.awt.Polygon;
import java.awt.image.BufferedImage;
import java.util.List;

/**
 *
 * @author Alexej Tietz
 */
public interface Animatable extends Cloneable {
    void nextStep();

    void resetAnimation();

    void setDying(int msec);

    void setLifeCycle(int cycles);

    void resumeAt(int index);

    boolean isDying();

    boolean isDead();

    boolean isFuncValAtEnd();

    BufferedImage getActualImage();

    Point getActualPosition();

    Polygon getGeomethry(int tolerance);

    int getAnimationLineLength();

    List<BufferedImage> getAnimationLine();

    void setUpdateRate(int rate);

    void setPosition(Point p);

    void setAnimationLine(List<BufferedImage> animation);

    void setSize(Dimension developedDimension, int factor);

    void setMovementFunktion(MovementFunktion f);

    void setMovementFunktionParam(double startValue, double endValue, double step);
}
```

Abbildung 5.2: Interface des 2D-Objekts „*Animatable*"

Da die Schnittstelle zu dem 2D-Objekt bekannt ist, kann mit der Implementierung des 2D-Objekts fortgesetzt werden. Es gibt noch eine erweiterte Implementierung des 2D-Objekts, die bei dem Plug-in „Sky Defendor“ verwendet wird.
Die Implementierung des 2D-Objekts „*AnimatableWorldObject*“ wird in sechs Sektionen unterteilt:

- ***Importe und Klassenvariablen***
- ***Konstruktor***
- ***Methoden zum Konfigurieren eines 2D-Objekts***
- ***Methoden zum Abfragen von den Zuständen und den Eigenschaften eines 2D-Objekts***
- ***Methoden zum Manipulieren der Animation eines 2D-Objekts***
- ***Methode „clone“ des 2D-Objekts***

Die Importe der genutzten Klassen und die Klassenvariablen (siehe Abbildung 5.3) folgen demnächst.

```java
package multieye.plugins;

import java.awt.Dimension;
import java.awt.Point;
import java.awt.Polygon;
import java.awt.image.BufferedImage;
import java.util.ArrayList;
import java.util.List;
import multieye.utility.BufImModification;

/**
 *
 * @author Alexej Tietz
 */
public class AnimatableWorldObject implements Animatable {
    protected List<String> animationList;
    protected List<BufferedImage> animation = new ArrayList<BufferedImage>();
    protected Dimension developedDimension;
    protected Point position;
    protected int animationIndex = 0;
    protected int updateCount = 0;
    protected int updateRate = 60;
    protected boolean dead = false;
    protected boolean dying = false;
    protected int dyingTime = 0;
    protected long dyingStart = 0;
    protected int lifeCycle = 0;
    protected int lifeCycleMax = 0;
    protected MovementFunktion f;
    protected double startValue;
    protected double endValue;
    protected double step;
    protected double stepBackup;
    protected double functionCounter;
```

Abbildung 5.3: Implementierung des 2D-Objekts „*AnimatableWorldObject*“ Teil 1

Um die Implementierung besser zu verstehen, werden die Klassenvariablen und ihre Bedeutung in dem Code des 2D-Objekts „*AnimatableWorldObject*“ hiernach erläutert:

- ***List<String> animationList*** – die Variable ist von dem Interface „*List*“ und mit der Klasse „*String*“ typisiert. Sie ist nicht initialisiert und nicht bei der Verwendung einer Implementierung beschränkt, egal ob sie eine „*AbstractList*“, eine „*ArrayList*“, eine „*LinkedList*“ oder ein „*Vector*“ ist. Die Liste wird bei der Erstellung eines 2D-Objekts übergeben und ist üblicherweise von der Klasse „*ArrayList*“. Die Variable enthält die Verweise auf die Bilder der Animation eines 2D-Objekts.

- ***List<BufferedImage> animation = new ArrayList<BufferedImage>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit der Klasse „*BufferedImage*“ typisiert. Die Liste enthält die Bilder der Animation eines 2D-Objekts.

- ***Dimension developedDimension*** – die Klasse dieser Variable repräsentiert eine Dimension. Sie enthält die Entwicklungsauflösung des Plug-Ins.

- ***Point position*** – die Variable ist von der Klasse „*Point*“ und repräsentiert einen 2D-Punkt (x und y). Sie enthält die Position eines 2D-Objekts in der 2D-Welt eines Plug-Ins.

- ***int animationIndex = 0*** – die Variable ist von der Klasse „*Integer*“ und repräsentiert eine ganze Zahl. Sie hat die Nummer des aktuellen Bildes der Animation eines 2D-Objekts.

- ***int updateCount = 0*** – die Variable ist von der Klasse „*Integer*“ und repräsentiert eine ganze Zahl, den Aktualisierungszähler. Der Ablauf des Aktualisierungszählers bedeutet, dass ein Bild in der Animation eines 2D-Objekts auf ein nächstes Bild aktualisiert werden muss.

- ***int updateRate = 60*** – die Variable ist von der Klasse „*Integer*“ und repräsentiert eine ganze Zahl. Sie hat die Aktualisierungsschranke, d.h. dass ein Bild bei jeder 60ten Aktualisierung ausgewechselt wird.

- ***boolean dead = false*** – die Klasse der Variable repräsentiert einen booleschen Wert, der „***wahr***“ oder „***nicht wahr***“ sein kann. Sie zeigt, ob ein 2D-Objekt verweilt ist oder nicht.

- ***boolean dying = false*** – die Klasse der Variable repräsentiert einen booleschen Wert, der „***wahr***“ oder „***nicht wahr***“ sein kann. Sie zeigt, ob ein 2D-Objekt gerade verweilt oder nicht.

- ***int dyingTime = 0*** – die Variable ist von der Klasse „*Integer*“ und repräsentiert eine ganze Zahl. Sie enthält die Verweildauer eines 2D-Objekts in Millisekunden.

- ***long dyingStart = 0*** – die Variable ist von der Klasse „*long*“ und repräsentiert eine ganze Zahl mit einem zweimal größeren Wertebereich als bei „Integer“. Sie enthält die Startzeit des Verweilens eines 2D-Objekts in Millisekunden.

- ***int lifeCycle = 0*** – die Variable ist von der Klasse „*Integer*“ und repräsentiert eine ganze Zahl. Sie enthält den aktuellen Lebenszyklus eines 2D-Objekes.

- ***int lifeCycleMax = 0*** – die Variable ist von der Klasse „*Integer*“ und repräsentiert eine ganze Zahl. Sie enthält die Lebensdauer (Anzahl von Zyklen) eines 2D-Objekts.

- ***MovementFunktion f*** – die Variable ist von dem Interface „*MovementFunktion*“ und repräsentiert eine Schnittstelle für eine Bewegungsfunktion eines 2D-Objektes.

- ***double startValue*** – die Klasse, von der die Variable ist, repräsentiert eine Fließkommazahl. Sie enthält den Startwert der Bewegungsfunktion.

- ***double endValue*** – die Klasse, von der die Variable ist, repräsentiert eine Fließkommazahl. Sie enthält den Endwert der Bewegungsfunktion.

- ***double step*** – die Klasse, von der die Variable ist, repräsentiert eine Fließkommazahl. Sie enthält den Schrittwert der Bewegungsfunktion.

- ***double stepBackup*** – die Klasse, von der die Variable ist, repräsentiert eine Fließkommazahl. Sie enthält den gesicherten Schrittwert der Bewegungsfunktion.

- ***double functionCounter*** – die Klasse, von der die Variable ist, repräsentiert eine Fließkommazahl. Sie enthält den Zähler der Bewegungsfunktion.

Als Nächstkommendes folgt ein Codeabschnitt mit dem Konstruktor (siehe Abbildung 5.4) der Klasse „*AnimatableWorldObject*“. In dem Konstruktor werden die Position, die Verweise auf die Bilder der Animation und die Auflösung der Bilder des 2D-Objekts zugewiesen. Das Entscheidende in dem Konstruktor ist der Aufruf der Methode „*setSize*“. Die Erklärung, was hinter der Methode steckt, kommt, nachdem der Code der Methode gezeigt wird.

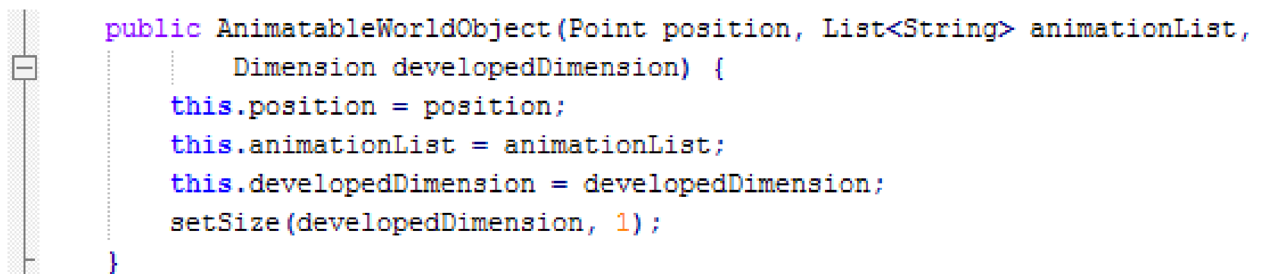


Abbildung 5.4: Implementierung des 2D-Objekts „*AnimatableWorldObject*“ Teil 2

Der nachfolgende Codeabschnitt enthält nur die Methode „*setSize*“ zum Konfigurieren eines 2D-Objekts (siehe Abbildung 5.5). Einige Methoden werden in Folge ausgeblendet, weil ihre Implementierung trivial ist. Die ausgeblendeten Methoden zum Konfigurieren eines 2D-Objekts können bei einem bestehenden Interesse von der beigelegten CD-ROM angeschaut bzw. bezogen werden. Die Methode „*setSize*“, die in dem Konstruktor aufgerufen wird, setzt nicht einfach die Auflösung der Bilder, sondern sie initialisiert die Animation mit skalierten Instanzen der Bilder.

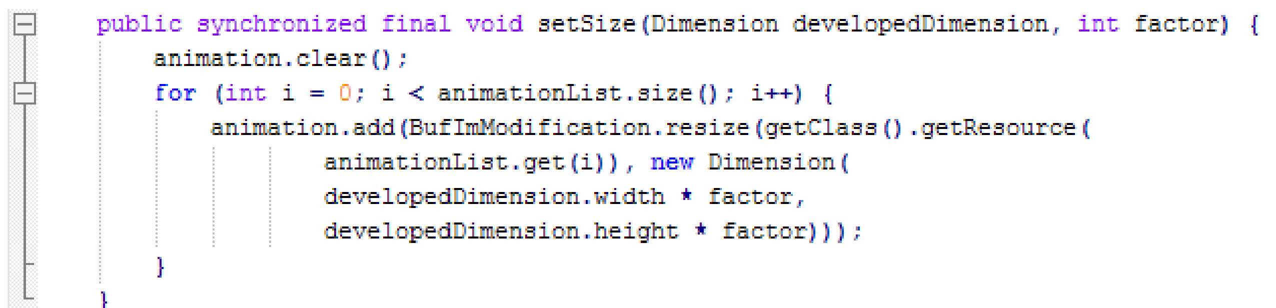


Abbildung 5.5: Implementierung des 2D-Objekts „AnimatableWorldObject“ Teil 3

Weiterhin folgen die Methoden zum Abfragen von Zuständen und Eigenschaften des 2D-Objekts (siehe Abbildung 5.6). Einige Methoden werden in Folge ausgeblendet, da ihre Implementierung trivial ist. Die ausgeblendeten Methoden zum Abfragen von Zuständen und Eigenschaften des 2D-Objekts können bei einem bestehenden Interesse von der beigelegten CD-ROM angeschaut bzw. bezogen werden.
Zunächst kommt die Methode „*getActualImage*“, die vom Namen her passiv zu sein scheint, aber das ist sie nicht. In dieser Methode wird ein aktuelles Bild gerendert. Wenn ein 2D-Objekt verweilt, wird es mit einem Fade-out-Effekt (Kapitel 2 Teilabschnitt 2) ausgeblendet. Der Fade-out-Effekt beruht auf einem Alphawert (Kapitel 2 Teilabschnitt 2), der in einem Bezug zu der vergangenen Verweilzeit steht.
Als Letztes wird die Methode „*getGeomethry*“ gezeigt, mit der man ein Polygon des 2D-Objekts erhalten kann. Die Methode hat als Parameter eine Toleranz:

- ***positive*** Toleranz – gibt an, dass das Polygon um die angegebene Toleranz in Pixeln größer als seine eigentlichen Maßen ist.
- ***negative*** Toleranz – gibt an, dass das Polygon um die angegebene Toleranz in Pixeln kleiner als seine eigentlichen Maßen ist.

Mit der Klasse „*Polygon*“ kann man mit der Methode „*contains*“ überprüfen, ob ein Punkt im Polygon liegt. Also kann man feststellen, wenn der Blickpunkt der Augen auf ein 2D-Objekt ausgerichtet ist.

```
public synchronized BufferedImage getActualImage() {
    if (dying) {
        long timeGone = (System.currentTimeMillis() - dyingStart) + 1;
        if (timeGone > dyingTime) {
            functionCounter = startValue;
            dead = true;
        }
        int alpha = 100 - (int) (timeGone * 100 / dyingTime);
        alpha = (alpha < 0) ? 0 : alpha;
        return BufImModification.setAlpha(
                animation.get(animationIndex), alpha);
    }
    return animation.get(animationIndex);
}

public synchronized Polygon getGeomethry(int tolerance) {
    Polygon p = new Polygon(new int[]{position.x - tolerance,
                position.x + getActualImage().getWidth() + 2 * tolerance,
                position.x + getActualImage().getWidth() + 2 * tolerance,
                position.x - tolerance},
            new int[]{position.y - tolerance, position.y - tolerance,
                position.y + getActualImage().getHeight() + 2 * tolerance,
                position.y + getActualImage().getHeight() + 2 * tolerance}, 4);
    return p;
}
```

Abbildung 5.6: Implementierung des 2D-Objekts „AnimatableWorldObject“ Teil 4

In dem nachfolgenden Codeabschnitt wird die wichtigste Methode zum Manipulieren der Animation des 2D-Objekts (Abbildung 5.7) gezeigt. Es gibt drei Methoden, mit denen man die Animation des 2D-Objekts manipulieren kann. Sie können bei einem bestehenden Interesse von der beigelegten CD-ROM angeschaut bzw. bezogen werden.

Die Methode „*nextStep*“ ist die wichtigste Methode der ganzen Klasse „*AnimatableWorldObject*“, da sie die Änderung der Position und die Aktualisierung des Bildes des 2D-Objekts anstoßt. Bei jedem Aufruf der Methode wird der Aktualisierungszähler inkrementiert. Wenn der maximale Aktualisierungswert erreicht ist, wird der Animationszähler inkrementiert, der mit seinem Wert auf ein Bild der Animation des 2D-Objekts verweist. Zusätzlich wird bei jedem Aufruf der Methode die neue Position eines 2D-Objekts berechnet. Es wird auch geschaut, ob ein Objekt verweilen muss und wenn dies der Fall ist, wird bei dem Objekt das Verweilen aktiviert. Die Methode erfüllt noch eine Funktion. Falls die Bewegungsfunktion ihren Endwert erreicht hat, wird der Funktionszähler auf den Startwert gesetzt.

```
public synchronized void nextStep() {
    updateCount = (updateCount + 1) % updateRate;
    if (updateCount == 0) {
        animationIndex = (animationIndex + 1) % animation.size();
        if (lifeCycleMax > 0) {
            lifeCycle++;
            if (!isDying() && lifeCycle >= lifeCycleMax) {
                setDying(1000);
            }
        }
    }
    if (f != null) {
        if ((endValue > startValue && functionCounter >= endValue)
                || (endValue < startValue && functionCounter <= endValue)) {
            functionCounter = startValue;
            position.setLocation(f.funktion(functionCounter));
        }
        position.setLocation(f.funktion(functionCounter += step));
    }
}
```

Abbildung 5.7: Implementierung des 2D-Objekts „AnimatableWorldObject“ Teil 5

Die Methode „*clone*“ ist aus den Gründen der Performanz implementiert, da eine Kopie eines Objekts zu erstellen weniger dauert als ein Objekt neu anzulegen. Dadurch wird das Flackern eines Bildes bei dem 2D-Rendering vermieden, weil ein Objekt innerhalb einer kurzen Zeit erstellt werden muss. Die Methode wird von der Mutterklasse „*Objekt*“ überschrieben und daher wird der Konstruktor der Superklasse zum Erstellen eines Objekts verwendet. Danach müssen alle interne Zustände von dem zu kopierenden Objekt dem neuerstellten Objekt zugewiesen werden. Man muss bei der Zuweisung darauf achten: wenn man eine Objektzuweisung macht, dass die Methode „*clone*“ des Objekts verwendet wird. Falls eine Objektzuweisung ohne die Methode „*clone*“ des Objekts geschieht, wird nur die Referenz zugewiesen und das kann zu nicht sofort erkennbaren Problemen führen.
Die genauere Implementierung der Methode „*clone*“ kann bei einem bestehenden Interesse von der beigelegten CD-ROM angeschaut bzw. bezogen werden.

5.3. Aktivierungsmanager

Wenn man ein Standardprogramm mit einer grafischen Oberfläche entwickelt, gibt es grafische Komponenten, die bestimmte Ereignisse auslösen können. Zum Beispiel wird bei einem Mausklick das Ereignis „*onClick*" ausgelöst und ein Callback (Rückruffunktion) für das Ereignis aufgerufen.
Aber worauf soll man reagieren, wenn man nur ein gerendertes Bild und nur den Ausrichtungspunkt von den Augen einer Person hat? Man könnte auf das Blinzeln reagieren, also wenn man ein bestimmtes Teil des Bildes anschaut und blinzt. Aber man kann unbewusst blinzeln und wenn man kurzeitig die Augen schließt und aufmacht, wird es von einem Eye-Tracker angenommen, der Kontakt zu den Augen einer Person wäre verloren. Das ist eine schlechte Idee!
In einem Interaktiven Programm kann man auf das Anvisieren reagieren.
Wie soll es dann mit Schaltflächen umgehen? Der Aktivierungsmanager erfüllt seine Rolle auf folgende Weise: wenn man eine Weile (z. B. 1s) eine Schaltfläche anstarrt, wird ein Callback-Mechanismus aufgerufen, der dann die Funktionalität der Schaltfläche enthält.

Der Aktivierungsmanager enthält:
- Ein Callback.
- Ein Zeitintervall, über dessen Länge eine Schaltfläche angeschaut werden muss.
- Eine gegebene Toleranz in Bezug auf das Zeitintervall.

Hiernach kommt das Interface „*Action*" (siehe Abbildung 5.8) der Rückruffunktion, das eine Schaltfläche implementiert und dem Aktivierungsmanager übergeben werden muss. Es muss nur eine Methode „*act*" implementiert werden, die dann die Aktivierungsfunktion der Schaltfläche enthält.

```java
package multieye.plugins;

/**
 *
 * @author Alexej Tietz
 */
public interface Action {
    void act();
}
```

Abbildung 5.8: Interface „*Action*" eines Callbacks für den Aktivierungsmanager „*ActivationManager*"

Als Nächstes folgt die Implementierung des Aktivierungsmanagers „*ActivationManager*" (siehe Abbildung 5.9) mit der Erklärung seiner Funktionalität.
Der Aktivierungsmanager ist sehr einfach aufgebaut. Er besitzt einen Konstruktor, eine Hauptmethode und eine Hilfsmethode.
Der Konstruktor der Klasse „*ActivationManager*" hat fünf Parameter:
- maximale Anvisierungszahl (Plug-Ins arbeiten mit 50Hz und es wird 50 Mal pro Sekunde aktualisiert. Pro eine Aktualisierung ist maximal eine Anvisierung möglich).
- maximale Anvisierungszeit zum Aktivieren eines Callbacks.
- Rücksetzungszeit (die Zeit ist für den Fall, falls man eine Schaltfläche zufällig anschaut, notwendig).
- eine Toleranz.
- eine Rückruffunktion.

Die Toleranz beeinflusst die maximale Anvisierungszahl, aus der dann die Toleranz für die Aktivierung der Rückruffunktion berechnet wird.

Die Hilfsmethode „*getSum*“ berechnet die Summe von Elementen einer Liste. Die Implementierung der Methode wird ausgeblendet, aber sie kann bei einem bestehenden Interesse von der beigelegten CD-ROM angeschaut bzw. bezogen werden.
Die Methode „*addHit*“ erfüllt die ganze Arbeit. Sie bekommt jedes Mal die Systemzeit, wenn eine Schaltfläche anvisiert wird. Aus der Systemzeit und der vorher übergebenen Zeit wird eine Differenz berechnet und in eine Liste hinzufügt. Wenn die Differenz die Rücksetzungszeit übersteigt, wird der Inhalt der Liste mit den Differenzzeiten gelöscht. Und falls die Größe der Liste den Wert der berechneten Toleranz erreicht und die Summe der Differenzzeiten kleiner als die doppelte maximale Anvisierungszeit ist, wird die Rückruffunktion aufgerufen und die Funktionalität der Schaltfläche aktiviert. Nach dem Aufruf des Callbacks wird der Inhalt der Liste mit den Differenzzeiten gelöscht. Damit wird die Anvisierung der Schaltfläche zurückgesetzt.

```java
public class ActivationManager {
    private List<Integer> hits;
    private int maxHits;
    private int maxTimeMillis;
    private int resetTime;
    private long lastHitTime = System.currentTimeMillis();
    private int tolerance;
    private Action a;

    public ActivationManager(int maxHits, int maxTimeMillis, int resetTime,
            int tolerance, Action a) {
        this.maxHits = maxHits;
        this.maxTimeMillis = maxTimeMillis;
        this.resetTime = resetTime;
        this.tolerance = maxHits * tolerance / 100;
        this.a = a;
        hits = new ArrayList<Integer>();
    }

    public void addHit(long hitTime) {
        int time = (int) (hitTime - lastHitTime);
        if (time > resetTime) {
            hits.clear();
        }
        hits.add(time);
        if (hits.size() > tolerance) {
            hits.remove(0);
        }
        if (hits.size() == tolerance
                && getSum(hits) < maxTimeMillis * 2) {
                a.act();
                hits.clear();
        }
        lastHitTime = hitTime;
    }
}
```

Abbildung 5.9: Implementierung des Aktivierungsmanagers „ActivationManager“

5.4. Plug-In: „Sky Defendor“ (2D-Spiel)

„Sky Defendor“ (Das Wort Defendor wird eigentlich als „*Defender*“ geschrieben, aber es ist wegen der Einzigartigkeit so gewollt.) ist ein interaktives 2D-Spiel, bei dem man einer Anzahl (zwei Schwierigkeitsstufen) von Schafen den Übergang von einem zum anderen Bunker ermöglichen muss. Wieso ermöglichen? Weil im Himmel die Flugzeuge von links nach rechts und umgekehrt fliegen und Bomben auf Schafe abwerfen. Dabei muss man die abgeworfenen Bomben mit den Augen anvisieren, um diese explodieren zu lassen. Anderenfalls können die Bomben die Schafe treffen, was zum Verlust der Schafe führt.

In dem Spiel gibt es verschiedene Objekte, die alle von dem 2D-Objekt „*AnimatableWorldObject*“ stammen:

- ***Menübuttons*** sind Schaltflächen, die mit Hilfe des Aktivierungsmanagers aktiviert werden.
- ***Flugzeuge*** sind animierte und bewegliche 2D-Objekte.
- ***Bomben*** sind animierte und bewegliche 2D-Objekte.
- ***Explosionen*** sind animierte 2D-Objekte.
- ***Wolken*** sind stille und passive 2D-Objekte.
- ***Sonne*** ist ein animiertes und passives 2D-Objekt.
- ***Zwei Bunker*** sind stille und passive 2D-Objekte.
- ***Schafe*** sind bewegliche 2D-Objekte.
- ***Sky Defendor*** ist ein Abwehrturm mit einem Geschütz. Das Geschütz wird durch das Anvisieren mit den Augen animiert. Der Sky Defendor erweitert das 2D-Objekt „*AnimatableWorldObject*“.
- ***Visier*** ist ein 2D-Objekt, das durch die Augen eines Spielers bewegt wird.

5.4.1. Erweiterung des 2D-Objekts „*AnimatableWorldObject*“ mit der Klasse „*SDDefendor*“

Als Erstes wird die Erweiterung des 2D-Objekts „*AnimatableWorldObject*“ gezeigt, die in der Klasse „*SDDefendor*“ vorgenommen wurde.
In dem hier nachfolgenden Codeabschnitt (siehe Abbildung 5.10) kommt ein Teil der Implementierung der Klasse „*SDDefendor*“, das die Importe, die Klassenvariablen und den Konstruktor der Klasse enthält.
Es gibt folgende Klassenvariablen:

- ***BufferedImage tower*** – die Variable ist von der Klasse „*BufferedImage*“ und repräsentiert ein Bild, das mit Hilfe eines Zwischenspeichers gepuffert wird. Sie enthält das Bild des Abwehrgeschützes.

- ***double towerAngle = 0.0*** – die Klasse, von der die Variable ist, repräsentiert eine Fließkommazahl. Sie hat den Winkelwert des Abwehrgeschützes, also um wie viel Grad es bei dem Rendering rotiert werden muss.

- ***double towerAngleBefore = -1.0*** – die Klasse, von der die Variable ist, repräsentiert eine Fließkommazahl. Sie hat den alten Winkelwert und ist für die Vermeidung des Renderings des Bildes, falls der alte Wert dem aktuellen Wert entspricht/gleicht.

- ***Point towerPosition*** – die Variable ist von der Klasse „*Point*“ und repräsentiert einen 2D-Punkt (x und y). Sie enthält die Position des Abwehrgeschützes.

- ***BufferedImage im*** – die Variable ist von der Klasse „*BufferedImage*“ und repräsentiert ein Bild, das mit Hilfe eines Zwischenspeichers gepuffert wird. Die Variable enthält das vollständige gerenderte Bild des Sky Defendors, das aus den Bildern eines Turms und eines Abwehrgeschützes gezeichnet wird.

- ***int towerLowered = 40*** – die Variable ist von der Klasse „*Integer*“ und repräsentiert eine ganze Zahl. Sie hat den Wert, der angibt, um wie viel Prozent das Abwehrgeschütz bei dem Rendering gesenkt werden muss.

```
package multieye.plugins.skydefendor;

import multieye.plugins.AnimatableWorldObject;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.image.BufferedImage;
import java.util.List;
import multieye.utility.BufImModification;

/**
 *
 * @author Alexej Tietz
 */
public class SDDefendor extends AnimatableWorldObject {
    protected BufferedImage tower;
    protected double towerAngle = 0.0;
    protected double towerAngleBefore = -1.0;
    protected Point towerPosition;
    protected BufferedImage im = null;
    protected int towerLowered = 40;

    public SDDefendor(Point position, List<String> animationList,
            Dimension developedDimension, String tower, int towerLowered) {
        super(position, animationList, developedDimension);
        setTower(BufImModification.resize(getClass().getResource(tower),
                new Dimension(super.getActualImage().getWidth(),
                super.getActualImage().getHeight())), towerLowered);
    }
```

Abbildung 5.10: Implementierung des Abwehrturms „*SDDefendor*“ Teil 1

Bei dem Konstruktor wird zum Erstellen des Objekts der Konstruktor der Mutterklasse aufgerufen, der einen Turm erstellt. Danach wird die Methode „*setTower*“ aufgerufen, um das Abwehrgeschütz zu erstellen.

Der nächste Codeabschnitt (siehe Abbildung 5.11) enthält drei Methoden. Es fehlen einige Methoden zu der vollständigen Implementierung der Klasse, deren Implementierung trivial ist.
Die Methode „*setTowerAngle*“ setzt nach einer Vorüberprüfung auf Gültigkeit den Winkelwert des Abwehrgeschützes.

```
public void setTowerAngle(double angle) {
    if (angle != Double.NaN || (angle > -75 && angle < 75)) {
        towerAngle = angle;
    }
}

public void setTower(BufferedImage tower, int towerLowered) {
    this.towerLowered = towerLowered;
    towerPosition = new Point(0,
            super.getActualImage().getHeight() * towerLowered / 100);
    this.tower = tower;
}

@Override
public BufferedImage getActualImage() {
    if (im == null || towerAngle != towerAngleBefore) {
        int over = tower.getHeight() * towerLowered / 100;
        im = new BufferedImage(super.getActualImage().getWidth(),
                super.getActualImage().getHeight() + over,
                BufferedImage.TYPE_INT_ARGB);
        Graphics g = im.getGraphics();
        g.drawImage(BufImModification.rotate(tower, towerAngle),
                0, 0, null);
        g.drawImage(super.getActualImage(), 0, tower.getHeight() - over, null);
        g.dispose();
        towerAngleBefore = towerAngle;
    }
    return im;
}
```

Abbildung 5.11: Implementierung des Abwehrturms „*SDDefendor*" Teil 2

Die Methode „*setTower*" setzt den Senkungswert für das Bild des Abwehrgeschützes und berechnet die Position des Abwehrgeschützes. Sie wird im Konstruktor aufgerufen.
Die Methode „*getActualImage*" ist die wichtigste in dieser Erweiterung der Klasse „*AnimatableWorldObject*". Sie überschreibt die Methode „*getActualImage*" der Mutterklasse, von der sie erbt. In der Methode der Klasse „*SDDefendor*" besteht das Rendering aus zwei Phasen. Zuerst wird das rotierte Abwehrgeschütz gezeichnet und dann obendrauf der Turm. Das Bild ist dann der vollständige Sky Defendor (Abwehrturm).

5.4.2. Importe, Klassenvariablen und Konstruktor der Klasse „MEPlugInSkyDefendor"

Nachdem die Erweiterung der Klasse „*AnimatableWorldObject*" bekannt ist, wird mit der Realisierung des Plug-Ins „Sky Defendor" fortgesetzt.

Als Erstes kommen die Importe der Klassen des Plug-Ins „*MEPlugInSkyDefendor*" (siehe Abbildung 5.12), die bei dem Plug-In „Sky Defendor" genutzt sind. Die Importe sind in einer abgekürzten Form, daher wundern Sie sich nicht, wenn der Compiler bei so einer Eingabe meckert. Die Abkürzung ist in folgender Form dargestellt: alle Klassen, die nach einem Komma kommen, sind aus demselben Package wie die vorhergehende Klasse.

```
package multieye.plugins;

import java.awt.Color,Component,Dimension,Font,Graphics,Image,Point,Polygon,Toolkit;
import java.awt.event.KeyAdapter,event.KeyEvent,event.MouseAdapter,event.MouseEvent
import java.awt.image.BufferedImage;
import java.io.ByteArrayInputStream,DataInputStream;
import java.util.ArrayList,HashMap,HashSet,List,Map,Set,logging.Level,logging.Logger;
import javax.swing.JFrame,swing.JPanel;
import multieye.MultiEye,connectors.MEConnector,plugins.MovementFunktion;
import multieye.plugins.skydefendor.SDDefendor;
import multieye.utility.BufImModification,Converter,ModDimension,ResizeFactor;
```

Abbildung 5.12: Importe der Klasse „*MEPlugInSkyDefendor*“

Es folgen nun die Klassenvariablen der Klasse „*MEPlugInSkyDefendor*“ (siehe Abbildung 5.13 und Abbildung 5.14) und deren Bedeutung in dem Plug-In „Sky Defendor“:

- ***Graphics dbg*** – die Variable ist von der abstrakten Klasse „*Graphics*" und stellt eine allgemeine Schnittstelle zur grafischen Ausgabe in einem bestimmten Kontext dar. Sie wird in der Methode „*gameRender*" verwendet, um das Bild der 2D-Welt zu zeichnen (vgl. dpunkt.Verlag 2002).

- ***int killedSheeps = 0, shotBombs = 0, sheepsToSave = 1, savedSheeps = 0*** – die Variablen sind von der Klasse „*Integer*“ und repräsentieren jeweils eine ganze Zahl. Sie beziehen sich auf: die Anzahl der getöteten Schafe, die Anzahl der abgewehrten Bomben, die Anzahl der noch zu rettenden Schafe und die Anzahl der geretteten Schafe.

- ***static final Color lightBlue = new Color(0.17f, 0.87f, 1.0f)*** – die Klasse, von der die Variable ist, repräsentiert eine Farbe. Sie enthält „***hell-blaue***“ Farbe, die im Spiel die Farbe des Himmels darstellt.

- ***Image dbImage*** – die Variable ist von der abstrakten Klasse „*Image*" und repräsentiert ein Bild. Sie nimmt das Bild der 2D-Welt auf.

- ***int fontSize = 18, groundHeight = 7*** – die Variablen sind von der Klasse „*Integer*“ und repräsentieren je eine ganze Zahl. Sie beziehen sich auf die Schriftgröße und die Höhe des Grunds im Bild der 2D-Welt.

- ***ResizeFactor resizeFactor*** – die Variable ist von der Hilfsklasse „*ResizeFactor*“ und repräsentiert einen Skalierungsfaktor. Sie enthält das Verhältnis zwischen der Entwicklungsauflösung und der aktuellen Ausführungsauflösung des Plug-Ins.

- ***BufferedImage menuBackground*** – die Variable ist von der Klasse „*BufferedImage*“ und repräsentiert ein Bild, das mit Hilfe eines Zwischenspeichers gepuffert wird. Sie enthält das Hintergrundbild für das Menü im Spiel.

- ***List<String> game1StartAnimation, game2StartAnimation, continueGameAnimation, endGameAnimation = new ArrayList<String>()*** – die Variablen sind von der Klasse „*ArrayList*“ und mit der Klasse „*String*“ typisiert. Die Variablen enthalten die Verweise auf die Bilder der Schaltflächen (Spiel 1 starten, Spiel 2 starten, Fortsetzen und Beenden) im Menü des Plug-Ins.

```
/**
 * @author Alexej Tietz
 */
public class MEPlugInSkyDefendor extends JPanel implements MEPlugIn {
    private Graphics dbg;
    private int killedSheeps = 0,shotBombs = 0,sheepsToSave = 1,savedSheeps = 0;
    private static final Color lightBlue = new Color(0.17f, 0.87f, 1.0f);
    private Image dbImage = null;
    private int fontSize = 18,groundHeight = 7;//InitValuePercent
    private ResizeFactor resizeFactor;
    private BufferedImage menuBackground;
    private List<String> game1StartAnimation = new ArrayList<String>();
    private List<String> game2StartAnimation = new ArrayList<String>();
    private List<String> continueGameAnimation = new ArrayList<String>();
    private List<String> endGameAnimation = new ArrayList<String>();
    private Dimension buttonsSize = new Dimension(420, 140);
    private List<Animatable> buttons = new ArrayList<Animatable>();
    private List<ActivationManager> acts = new ArrayList<ActivationManager>();
    private List<String> bunkerAnimation = new ArrayList<String>();
    private Dimension bunkerSize = new Dimension(120, 120);
    private List<Animatable> bunkers = new ArrayList<Animatable>();
    private List<String> cloudsAnimation = new ArrayList<String>();
    private Dimension cloudsSize = new Dimension(150, 150);
    private List<Animatable> clouds = new ArrayList<Animatable>();
    private int sheepsMax = 4;
    private Dimension sheepsSize = new Dimension(110, 110);
    private List<Animatable> sheeps = new ArrayList<Animatable>();
    private List<String> sheepsAnimation = new ArrayList<String>();
    private int jetsMax = 3,jetsSpeed = -3;
    private Dimension jetsSize = new Dimension(200, 90);
    private List<Animatable> jets = new ArrayList<Animatable>();
    private List<String> jetsAnimation = new ArrayList<String>();
    private List<String> jetsAnimationInverted = new ArrayList<String>();
    private Dimension bombsSize = new Dimension(50, 50);
    private AnimatableWorldObject bomb;
    private List<Animatable> bombs = new ArrayList<Animatable>();
    private List<String> bombsAnimation = new ArrayList<String>();
    private Map<Integer, Integer> bombsPos = new HashMap<Integer, Integer>();
    private Dimension explosionsSize = new Dimension(50, 50);
    private AnimatableWorldObject explosion;
    private List<Animatable> explosions = new ArrayList<Animatable>();
    private List<String> explosionsAnimation = new ArrayList<String>();
```

Abbildung 5.13: Klassenvariablen der Klasse „*MEPlugInSkyDefendor*“ Teil 1

- ***Dimension buttonsSize = new Dimension(420, 140)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Größe (Höhe und Breite) der Schaltflächen im Menü.

- ***List<Animatable> buttons = new ArrayList<Animatable>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit dem Interface „*Animatable*“ typisiert. Die Liste enthält alle Schaltflächen von Menüs des Plug-Ins.

```
private List<String> defendorAnimation = new ArrayList<String>();
private Dimension defendorSize = new Dimension(120, 120);
private SDDefendor defendor;
private Dimension sunSize = new Dimension(150, 150);
private List<String> sunAnimation = new ArrayList<String>();
private AnimatableWorldObject sun;
private List<String> aimAnimation = new ArrayList<String>();
private Dimension aimSize = new Dimension(75, 75);
private AnimatableWorldObject aim;
protected boolean active = false,pause = false,continueGame = false;
protected boolean loose = false,win = false;
private int fpsHrz = 50;
private static int maxSkippedFrames = 3,skippedFrames = 0;
private int mouseMaxHits = fpsHrz * 3 / 2,mouseOverTimeMax = 1500;
private int missMouseResetTime = 900,mouseTolerance = 85;
final private MEPlugInSkyDefendor skyDefendor = this;
private ByteArrayInputStream arrayInputStream;
private DataInputStream dataInputStream;
private Dimension developedPanelSize = new Dimension(1019, 741);
private Dimension runDimension = new Dimension(1019, 741);
private BufferedImage progIcon = BufImModification.createBI(
        this.getClass().getResource("skydefendor/SkyDefendorProgIcon.png"));
private int messageType, x = 1, y = 1, latency;
private long millis = System.currentTimeMillis();
private Point correction;
MEConnector connector;
JFrame parentJF;
MultiEye parentME;
KeyAdapter keyHandler;
MouseAdapter testMousAdapter = new MouseAdapter() {
    @Override
    public void mouseMoved(MouseEvent e) {
        try {
            int x = e.getX()/* - correction.x*/;
            int y = e.getY()/* - correction.y*/;
            defendor.setTowerAngle(calcRotAngle(x, y));
            aim.setPosition(new Point(x, y));
        } catch (Exception ex) {
        }
    }
};
```

Abbildung 5.14: Klassenvariablen der Klasse „*MEPlugInSkyDefendor*“ Teil 2

- ***List<ActivationManager> acts = new ArrayList<ActivationManager>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit der Klasse „*ActivationManager*“ typisiert. Die Liste enthält alle Aktivierungsmanager, die Aktionen der Schaltflächen im Menü ausführen.

- ***List<String> bunkerAnimation = new ArrayList<String>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit der Klasse „*String*“ typisiert. Die Liste enthält die Verweise auf die Bilder der Bunker.

- ***Dimension bunkerSize = new Dimension(120, 120)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Größe (Höhe und Breite) der beiden Bunker.

- ***List<Animatable> bunkers = new ArrayList<Animatable>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit dem Interface „*Animatable*“ typisiert. Die Liste enthält die 2D-Objekte der beiden Bunker.

- ***List<String> cloudsAnimation = new ArrayList<String>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit der Klasse „*String*“ typisiert. Die Liste enthält Verweise auf die Bilder der Wolken.

- ***Dimension cloudsSize = new Dimension(150, 150)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Größe (Höhe und Breite) der Wolken.

- ***List<Animatable> clouds = new ArrayList<Animatable>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit dem Interface „*Animatable*“ typisiert. Die Liste enthält die 2D-Objekte der Wolken.

- ***int sheepsMax = 4*** – die Variable ist von der Klasse „*Integer*“ und repräsentiert eine ganze Zahl. Sie enthält einen Wert, der besagt, wie viele Schafe maximal gleichzeitig im Sichtfeld zu sehen sind.

- ***Dimension sheepsSize = new Dimension(110, 110)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Größe (Höhe und Breite) der Schafe.

- ***List<Animatable> sheeps = new ArrayList<Animatable>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit dem Interface „*Animatable*“ typisiert. Die Liste enthält die 2D-Objekte der Schafe.

- ***List<String> sheepsAnimation = new ArrayList<String>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit der Klasse „*String*“ typisiert. Die Liste enthält die Verweise auf die Bilder der Schafe.

- ***int jetsMax = 3, jetsSpeed = -3*** – die Variablen sind von der Klasse „*Integer*“ und repräsentieren je eine ganze Zahl. Sie enthalten: maximale Anzahl der Flugzeuge im Sichtfeld und ihre Geschwindigkeit.

- ***Dimension jetsSize = new Dimension(200, 90)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Größe (Höhe und Breite) der Flugzeuge.

- ***List<Animatable> jets = new ArrayList<Animatable>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit dem Interface „*Animatable*“ typisiert. Die Liste enthält die 2D-Objekte der Flugzeuge.

- ***List<String> jetsAnimation, jetsAnimationInverted = new ArrayList<String>()*** – die Variablen sind von der Klasse „*ArrayList*“ und mit der Klasse „*String*“ typisiert. Die Listen enthalten die Verweise auf die Bilder der Flugzeuge, die von links nach rechts oder umgekehrt fliegen.

- ***Dimension bombsSize = new Dimension(50, 50)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Größe (Höhe und Breite) der Bomben.

- ***AnimatableWorldObject bomb*** – die Variable ist von der Klasse „*AnimatableWorldObject*" und repräsentiert ein 2D-Objekt einer Bombe.

- ***List<Animatable> bombs = new ArrayList<Animatable>()*** – Variable ist von der Klasse „*ArrayList*" und mit dem Interface „*Animatable*" typisiert. Die Liste enthält die 2D-Objekte der Bomben.

- ***List<String> bombsAnimation = new ArrayList<String>()*** – die Variable ist von der Klasse „*ArrayList*" und mit der Klasse „*String*" typisiert. Die Liste enthält die Verweise auf die Bilder der Bomben.

- ***Map<Integer, Integer> bombsPos = new HashMap<Integer, Integer>()*** – die Variable ist von der Klasse „*HashMap*" und repräsentiert eine Tabelle, die aus zwei Spalten besteht. Die Spalten sind je mit der Klasse „*Integer*" typisiert. Die erste Spalte enthält die Schlüssel, die auf die Werte in der zweiten Spalte verweisen, wobei die Schlüssel eine mathematische Menge bilden. Die Tabelle enthält die x-Positionen der Bombenabwürfe.

- ***Dimension explosionsSize = new Dimension(50, 50)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Größe (Höhe und Breite) der Explosionen.

- ***AnimatableWorldObject explosion*** – die Variable ist von der Klasse „*AnimatableWorldObject*" und repräsentiert ein 2D-Objekt einer Explosion.

- ***List<Animatable> explosions = new ArrayList<Animatable>()*** – Variable ist von der Klasse „*ArrayList*" und mit dem Interface „*Animatable*" typisiert. Die Liste enthält die 2D-Objekte der Explosionen.

- ***List<String> explosionsAnimation = new ArrayList<String>()*** – die Variable ist von der Klasse „*ArrayList*" und mit der Klasse „*String*" typisiert. Die Liste enthält die Verweise auf die Bilder der Explosionen.

- ***List<String> defendorAnimation = new ArrayList<String>()*** – Variable ist von der Klasse „*ArrayList*" und mit der Klasse „*String*" typisiert. Die Liste enthält das Bild des Turms.

- ***Dimension defendorSize = new Dimension(120, 120)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Größe (Höhe und Breite) des Turms.

- ***SDDefendor defendor*** – die Variable ist von der Klasse „*AnimatableWorldObject*" und repräsentiert ein 2D-Objekt eines Abwehrturmes.

- ***Dimension sunSize = new Dimension(150, 150)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Größe (Höhe und Breite) der Sonne.

- ***List<String> sunAnimation = new ArrayList<String>()*** – die Variable ist von der Klasse „*ArrayList*" und mit der Klasse „*String*" typisiert. Die Liste enthält die Verweise auf die Bilder der Sonne.

- ***AnimatableWorldObject sun*** – die Variable ist von der Klasse „*AnimatableWorldObject*“ und repräsentiert ein 2D-Objekt der Sonne.

- ***List<String> aimAnimation = new ArrayList<String>()*** – die Variable ist von der Klasse „*ArrayList*“ und mit der Klasse „*String*“ typisiert. Die Liste enthält den Verweis auf das Bild des Visiers.

- ***Dimension aimSize = new Dimension(75, 75)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Größe (Höhe und Breite) des Visiers.

- ***AnimatableWorldObject aim*** – die Variable ist von der Klasse „*AnimatableWorldObject*“ und repräsentiert ein 2D-Objekt des Visiers.

- ***boolean active = false, pause = false, continueGame = false*** – die Klasse der Variablen repräsentiert einen booleschen Wert, der „***wahr***“ oder „***nicht wahr***“ sein kann. Die Variablen zeigen an: Aktivität des Plug-Ins, Pausieren des Plug-Ins und ob ein Spiel fortsetzbar ist.

- ***boolean loose = false, win = false*** – die Klasse der Variablen repräsentiert einen booleschen Wert, der „***wahr***“ oder „***nicht wahr***“ sein kann. Die Variablen zeigen, ob man verloren oder gewonnen hat.

- ***int fpsHrz = 50*** – Variable ist von der Klasse „*Integer*“ und repräsentiert eine ganze Zahl. Sie hat den Arbeitsfrequenzwert eines Plug-Ins. In diesem Fall ist er 50Hz.

- ***static int maxSkippedFrames = 3, skippedFrames = 0*** – die Variablen sind von der Klasse „*Integer*“ und repräsentieren je eine ganze Zahl. Sie besagen, wie viele Frames beim Rendering weggelassen werden können und wie viele Frames gerade verpasst wurden. Die Variablen spielen eine wichtige Rolle bei der Optimierung des Renderings.

- ***int mouseMaxHits = fpsHrz * 3 / 2, mouseOverTimeMax = 1500, missMouseResetTime = 900, mouseTolerance = 85*** – die Variablen sind von der Klasse „*Integer*“ und repräsentieren je eine ganze Zahl. Diese Variablen dienen als Parameter für die Aktivierungsmanager.

- ***MEPlugInSkyDefendor skyDefendor = this*** – die finale Variable ist von der Klasse „*MEPlugInSkyDefendor*“ und repräsentiert ein Plug-In „Sky Defendor“. Die Variable verweist auf die Instanz des eigenen Objekts dieser Klasse.

- ***ByteArrayInputStream arrayInputStream*** – die Variable ist von der Klasse „*ByteArrayInputStream*“ und repräsentiert ein gepuffertes Array, das z. B. zum Lesen von einem Stream verwendet werden kann. Sie wird verwendet um ein UDP-Paket zu lesen.

- ***DataInputStream dataInputStream*** – die Variable ist von der Klasse „*DataInputStream*“ und repräsentiert eine Datenquelle, von der man die primitiven Datentypen lesen kann, die zuvor als ein binäres Array vorliegen. Die Variable wird verwendet, um die einzelnen Variablen aus dem UDP-Paket heraus zu lesen.

- ***Dimension developedPanelSize = new Dimension(1019, 741)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die Entwicklungsauflösung des Plug-Ins.

- ***Dimension runDimension = new Dimension(1019, 741)*** – die Klasse der Variable repräsentiert eine Dimension. Sie hat die aktuelle Auflösung des Plug-Ins.

- ***BufferedImage progIcon*** – die Variable ist von der Klasse „*BufferedImage*“ und repräsentiert ein Bild, das mit Hilfe eines Zwischenspeichers gepuffert wird. Sie enthält das Bild des Programmicons.

- ***int messageType, x = 1, y = 1, latency*** – die Variablen sind von der Klasse „*Integer*“ und repräsentieren je eine ganze Zahl. Die Variablen enthalten die aktuellen Werte eines UDP-Pakets von einem Eye-Tracker. Das UDP-Paket besteht aus: ***Nachrichtentyp***, ***x-Position der Augen***, ***y-Position der Augen*** und ***Latenz***.

- ***long millis = System.currentTimeMillis()*** – die Variable ist von der Klasse „*long*“ und repräsentiert eine ganze Zahl mit einem zweimal größeren Wertebereich als bei „Integer“. Sie enthält die letzte Zeit des Renderings in Millisekunden.

- ***Point correction*** – die Variable ist von der Klasse „*Point*“ und repräsentiert einen 2D-Punkt (x und y). Die Variable enthält die x-y-Korrektur für das Plug-In, wenn ein Plug-In in einem Tab ausgeführt wird, da das Plug-In nicht mehr in dem (0,0)-Punkt liegt.

- ***MEConnector connector*** – die Variable ist von dem Interface „ *MEConnector*“ und repräsentiert eine Schnittstelle zu einem Connector, der eine Verbindung zu einem Eye-Tracker anbietet.

- ***JFrame parentJF*** – die Variable ist von der Klasse „*JFrame*“ und repräsentiert ein Programmfenster. In diesem Fall verweist sie auf das Hauptfenster der API – „Multi Eye“.

- ***MultiEye parentME*** – die Variable ist von dem Interface „*MultiEye*“ und repräsentiert eine Schnittstelle zur API. Sie enthält die Referenz zu der API – „Multi Eye“.

- ***KeyAdapter keyHandler*** – die Variable ist von der Klasse „*KeyAdapter*“ und repräsentiert einen Tastatur-Handler. Der Handler wird aufgerufen, wenn eine Taste (M oder Escape) gedrückt wird, wobei das Plug-In pausiert wird.

- ***MouseAdapter testMousAdapter*** – die Variable ist von der Klasse „*MouseAdapter*“ und repräsentiert einen Maus-Handler. Der Handler ist für die Testzwecke da und wird nur dann aktiviert, wenn das Plug-In ohne einen Connector in dem Testmodus ausgeführt wird.

Abschließend kommt der Konstruktor der Klasse „*MEPlugInSkyDefendor*“ (siehe Abbildung 5.15).
Zuerst wird bei dem Konstruktor der Konstruktor der Mutterklasse „*JPanel*“ aufgerufen, damit die Instanz initialisiert werden kann. Als Nächstes wird der Instanz ein hell-blaues Hintergrund zugewiesen. Danach wird die Fokussierungsfähigkeit eingeschaltet und das Panel der Instanz wird fokussiert. Nachfolgend wird der Tastatur-Handler „*keyHandler*“ durch eine anonyme Klasse implementiert, der bei dem Panel der Instanz als Tastatur-Handler registriert wird.

```
public MEPlugInSkyDefendor() {
    super();
    setBackground(lightBlue);
    setFocusable(true);
    requestFocus();
    keyHandler = new KeyAdapter() {
        @Override
        public void keyPressed(KeyEvent e) {
            int keyCode = e.getKeyCode();
            if (continueGame && ((keyCode == KeyEvent.VK_ESCAPE)
                    || (keyCode == KeyEvent.VK_M))) {
                pause = !pause;
            }
        }
    };
    addKeyListener(keyHandler);
}
```

Abbildung 5.15: Konstruktor der Klasse „*MEPlugInSkyDefendor*"

5.4.3. Hilfsmethoden der Klasse „MEPlugInSkyDefendor"

Die Hilfsmethoden sind für mehrere Bereiche zuständig. Es sind folgende Bereiche da:

- ***Vorinitialisierung***
  Die Methode „*resetVars*" setzt die Variablen der Spiellogik zurück, um ein neues Spiel anfangen zu können.
  Die Methode „*clearWorld*" entfernt alle 2D-Objekte der 2D-Welt (es verbleiben nur Menüschaltflächen), damit sie mit ihren Anfangszuständen neu initialisiert werden können.

- ***Anpassung an die Umgebung***
  Die Methode „*resizeComponents*" berechnet einen Skalierungsfaktor aus der Entwicklungs- und Ausführungsauflösung des Plug-Ins. Anhand dessen wird die neue Schriftgröße und die neue Auflösung von den Bildern der 2D-Objekte berechnet.

- ***Beeinflussung der 2D-Welt***
  Die Methode „*addExplosion*" fügt eine Explosion an dem übergebenen Punkt hinzu. Die Explosion wird mit Hilfe der Methode „*clone*" erstellt, wobei bei der Kopie die Positionierung in der 2D-Welt geändert, der Lebenszyklus des Objekts auf sechs Zyklen gesetzt und die Updaterate des Objekts auf 5Hz (fünf Bilder pro Sekunde) eingestellt werden.
  Die Methode „*calcRotAngle*" beeinflusst nicht direkt die 2D-Welt des Plug-Ins. Sie berechnet den Winkel, um den das Abwehrgeschütz rotiert werden muss. Der Winkel wird durch die Formel des Arkustangens ($\boldsymbol{\alpha = \arctan(a / b)}$) berechnet. Da die Methode „*atan*" aus dem Package „*Math*" den Winkel in Radiant berechnet, muss das Ergebnis in Grad durch die Methode „*toDegrees*" umgerechnet werden. In Abhängigkeit davon, in welchen Teil des Bildes das Visier sich befindet, muss der Winkel angepasst werden. Falls sich das Visier im linken Teil des Bildes befindet, muss der Winkel negativ werden, andernfalls positiv bleiben.

Das nächste Bild (siehe Abbildung 5.16) macht die Berechnung des Winkels anschaulicher und zeigt, wo das Koordinatensystem in Java 2D den Ursprung nimmt.

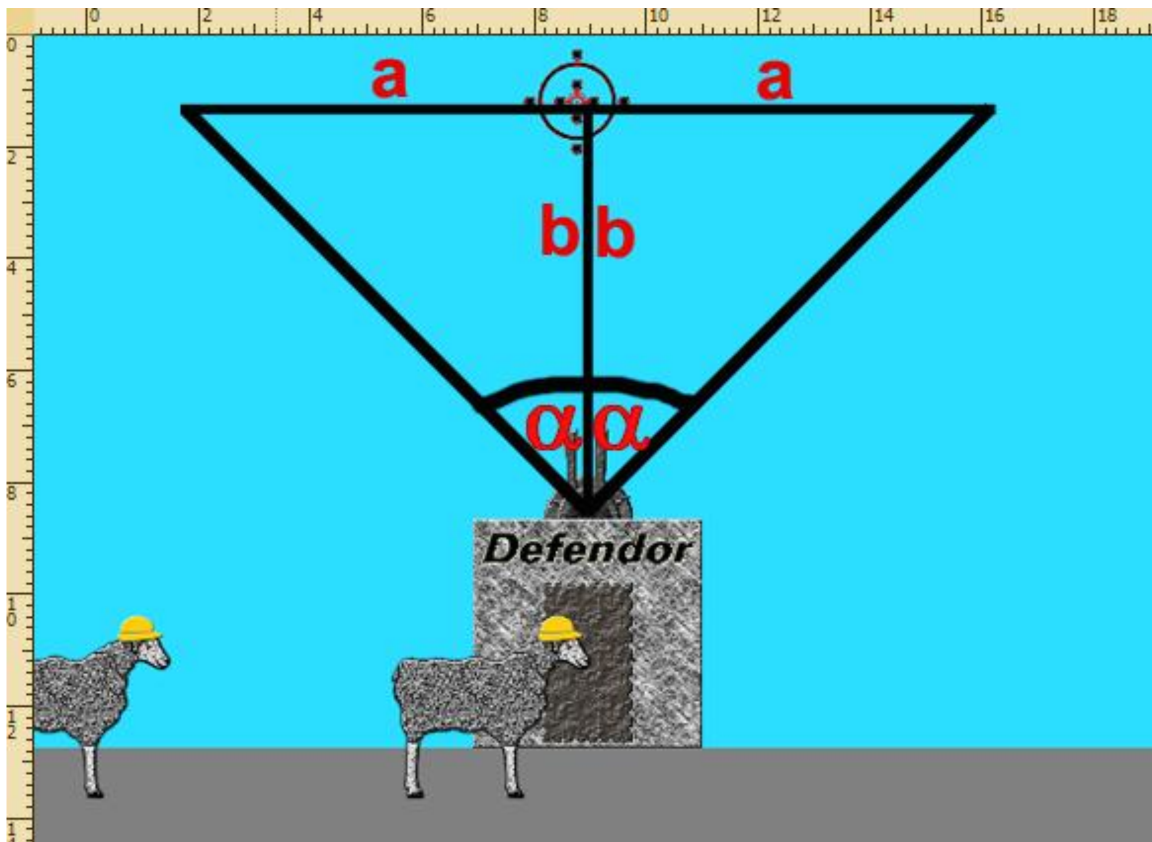


Abbildung 5.16: Winkelberechnung des Abwehrgeschützes

5.4.4. Initialisierung des Menüs und der 2D-Welt

Die Initialisierung des Menüs findet in drei Phasen statt. Zuerst wird die 2D-Welt visuell initialisiert, indem ein Menü-Hintergrundbild geladen wird. Danach werden die Schaltflächen positioniert, die Bilder für optische Repräsentierung von Schaltflächen geladen und die Größe der Bilder von Schaltflächen festgesetzt.
Bei der dritten Phase werden die Aktivierungsmanager initialisiert, die für die Aktivierung der Schaltflächenfunktionalität zuständig sind.

Die Initialisierung der 2D-Welt ist viel komplexer als die des Menüs und besteht aus der Initialisierung von den einzelnen 2D-Objekten. Jedes 2D-Objekt wird wiederum in mehreren Phasen zusammengesetzt. Die Initialisierung der 2D-Objekte besteht aus:

- ***Bunker*** – Initialisierung der visuellen Repräsentierung, Positionierung der Bunker und Zuweisung von Bildgrößen an die Bunker.

- ***Sonne*** – Initialisierung der Animation der Sonne, Positionierung der Sonne, Zuweisung von Bildergrößen der Sonnenanimation und Setzen der Frequenz für die Sonnenanimation.

- ***Wolken*** – Initialisierung der visuellen Repräsentierung, Positionierung der Wolken und Zuweisung von Bildgrößen an die Wolken.

- ***Schafe*** – Initialisierung der visuellen Repräsentierung, Positionierung von Schafen, Zuweisung von Bildgrößen an die Schafe und Setzen der Bewegungsfunktion.

- ***Flugzeuge*** – Initialisierung der Animation von Flugzeugen, Positionierung von Flugzeugen, Zuweisung von Bildgrößen der Flugzeuganimation und Setzen der Bewegungsfunktion.

- ***Bomben*** – Initialisierung der Animation von Bomben und Zuweisung von Bildgrößen der Bombenanimation. Weitere Zuweisungen finden in der Methode „*gameUpdate*“ statt, in der die Bomben in die 2D-Welt gesetzt werden.

- ***Explosionen*** – Initialisierung der Animation von Explosionen und Zuweisung von Bildgrößen der Explosionsanimation. Weitere Zuweisungen finden in der Methode „*addExplosion*“ statt, in der die Bomben mit den Explosionen in der 2D-Welt ausgetauscht werden.

- ***Abwehrturm*** –Initialisierung der visuellen Repräsentierung des Turms, Positionierung des Turms, Zuweisung von Bildgrößen des Turms und Setzen des Abwehrgeschützes auf den Turm.

- ***Visier*** –Initialisierung der visuellen Repräsentierung des Visiers, Positionierung des Visiers und Zuweisung von Bildgrößen des Visiers.

5.4.5. Datagetterthread und Animationsthread

Vor der Erläuterung der Threads (Datagetterthread und Animationsthread) kommt eine Erläuterung, wie die Threads überhaupt zu Stande kommen. Die Methode „*start*“ erweckt das Plug-In zum Leben. In ihr werden die Threads implementiert und gestartet, die den Lebenszyklus des Plug-Ins beschreiben. In dieser Methode kommt als Erstes ein Abschnitt, der für das Testen zuständig ist. Falls kein Connector übergeben wird, wird eine alternative Quelle (Kapitel 4 Teilabschnitt 6) benutzt.
Da das Plug-In durch die Methode „*start*“ zum Laufen gebracht wird, initialisiert die Methode die 2D-Welt und das Menü des Plug-Ins, bevor die Threads gestartet werden.

5.4.5.1. Datagetterthread

Als Nächstes wird der Datagetterthread (Abbildung 5.17) erläutert, in dem ein übergebener Connector zum Empfangen von Daten eines Eye-Trackers verwendet wird. Vor der Implementierungserklärung zu den Threads, muss eine Sache im Auge behalten werden und zwar, dass die UDP-Pakete von dem verwendeten Eye-Tracker in Little Endian Byte-Reihenfolge (Kapitel 2 Teilabschnitt 1) kommen und in Big Endian umgewandelt werden müssen, weil Java mit der Big Endian Byte-Reihenfolge arbeitet.
Die Daten von dem Eye-Tracker werden periodisch empfangen, dann in ein ArrayInputStream reingesteckt, das einem DataInputStream übergeben wird. Aus dem DataInputStream kann man die Bytes als primitive Datentypen auslesen, die danach in primitiven Datentype in Big Endian umgewandelt werden.

Wenn die Daten von dem Eye-Tracker richtig ausgelesen sind, müssen sie korrigiert werden, falls das Plug-In in einem Tab ausgeführt wird. Falls das Plug-In in einem Fullscreen-Modus läuft, hat die Korrektur keine Auswirkung, da die Korrekturdaten genullt sind. Die korrigierten Daten werden dann zum Setzen des Visiers und des Winkels des Abwehrgeschützes verwendet.

```
Thread dataGetter = new Thread() {
    @Override
    public void run() {
        byte[] data;
        while (active) {
            try {
                data = connector.getEyeTrackerData(24);
                if (data == null) {
                    continue;
                }
                arrayInputStream = new ByteArrayInputStream(data);
                dataInputStream = new DataInputStream(arrayInputStream);
                messageType = Converter.l2bEndianI(dataInputStream.readInt());
                x = (int) (Converter.l2bEndianD(dataInputStream.readDouble())
                        * parentJF.getWidth());
                y = (int) (Converter.l2bEndianD(dataInputStream.readDouble())
                        * parentJF.getHeight());
                latency = Converter.l2bEndianI(dataInputStream.readInt());
                x -= correction.x;
                y -= correction.y;
                defendor.setTowerAngle(calcRotAngle(x, y));
                aim.setPosition(new Point(x, y));
                arrayInputStream.reset();
            } catch (Exception ex) {
                parentME.setInfo(getClass().getName() + ":" + ex);
                Logger.getLogger(MEPlugInSkyDefendor.class.getName()).
                        log(Level.SEVERE, null, ex);
            }
        }
    }
};
```

Abbildung 5.17: Datagetterthread der Klasse „MEPlugInSkyDefendor“

5.4.5.2. Animationsthread

Der Animationsthread arbeitet drei Phasen ab. Jede der Phasen ist durch folgende Methoden implementiert/beschrieben:

- ***gameUpdate*** – die Methode enthält die ganze Logik des Spiels. In der Methode wird auf die Benutzerinteraktionen reagiert und es werden die Zustände der 2D-Objekte aktualisiert.
- ***gameRender*** – die Methode ist für das 2D-Rendering verantwortlich und für die Optimierung des 2D-Renderingsvorgangs zuständig.
- ***paintScreen*** – die Methode zeichnet das vorher gerenderte Bild, das die Repräsentierung der 2D-Welt ist.

5.4.5.2.1. Methode „*gameUpdate*“

Die Methode „*gameUpdate*“:

- ***überprüft auf das Ende des Spiels.***
  Es wird einfach überprüft, ob die Anzahl von den zu rettenden Schafen erreicht ist, dann hat man gewonnen. Es wird gecheckt, ob die Anzahl der gefallenen Schafe die Anzahl der zu rettenden Schafe übersteigt, dann hat man verloren.

- ***überprüft, ob die Schafe in Sicherheit sind.***
  Es wird bei jedem Schaf überprüft, ob seine Bewegungsfunktion den Endwert erreicht hat. Wenn das der Fall ist, ist das Schaf in Sicherheit, danach müssen seine Bewegungsfunktionswerte zurückgesetzt werden.

- ***generiert die Positionen, wo die Bomben abgeworfen werden.***
  Es gibt zwei Zonen für den Bombenabwurf. Die Erste ist zwischen dem ersten Bunker und dem Abwehrturm. Die Zweite ist zwischen dem Abwehrturm und dem zweiten Bunker. Die Berechnung der Bombenabwurfposition findet für jedes Flugzeug einzeln statt, wenn sich ein Flugzeug vor dem Betreten in der jeweiligen Abwurfzone befindet. Jedes Flugzeug kann nur eine Bombe in die jeweilige Zone abwerfen. Deshalb wird für jedes Flugzeug nur eine Bombenabwurfposition gespeichert, da sich ein Flugzeug nicht gleichzeitig in zwei Zonen befinden kann. Die berechnete Position wird zu einer Tabelle mit der zugehörigen Nummer des Flugzeuges eingefügt.

- ***fügt die Bomben zu den generierten Positionen hinzu, wenn ein Flugzeug über der Position ist.***
  Eine Bombe wird aus dem Hinterteil eines Flugzeugs abgeworfen, wenn es mehr als die Hälfte seiner Länge über der Bombenabwurfposition durchfliegt. Eine Bombe wird erst dann durch die Methode „*clone*“ erstellt und konfiguriert, wenn ein Flugzeug an der Abwurfposition ist. Die Bombe bekommt die Startposition, die Aktualisierungsrate für die Bombenanimation und die Bewegungsfunktion mit den Funktionsparametern. Danach wird die Position der abgeworfenen Bombe aus der Tabelle mit den Bombenabwurfpositionen entfernt.

- ***entfernt die explodierten Bomben und fügt stattdessen die Explosionen.***
  Es gibt nur drei Gründe, wieso eine Bombe explodieren kann. Der erste Grund ist das Antreffen der Bombe auf dem Boden, also eine Explosion durchs Aufschlagen auf dem Grund. Der zweite Grund ist das Treffen eines Schafes (dadurch wird ein Schaf erwischt und das Verweilen bei dem Schaf wird aktiviert). Der letzte Grund ist das Anvisieren einer Bombe (die Bombe wird abgeschossen und richtet keinen Schaden an). In jedem Fall wird eine Explosion an der Stelle der explodierten Bombe eingefügt. Danach wird die explodierte Bombe aus der Liste mit den Bomben entfernt.

- ***entfernt die vollendeten Explosionen.***
  Jede der Explosionen hat eine Lebensdauer (Explosionsdauer), die bei jeder Explosion standardmäßig auf sechs Lebenszyklen gesetzt wird. Nachdem eine Explosion vollendet ist, ist sie nicht mehr sichtbar und muss aus der Liste der Explosionen entfernt werden. Damit werden unbedeutsame Explosionen aus Optimierungsgründen nicht mehr betrachtet.

- ***aktualisiert die bewegbare und animierbare 2D-Objekte.***
  Es gibt fünf 2D-Objekte/2D-Objektgruppen (Sonne, Schafe, Flugzeuge, Bomben und Explosionen), die aktualisiert werden müssen. Die Aktualisierung geschieht durch den Aufruf der Methode „*nextStep*" eines 2D-Objekts. Die Methode aktualisiert die Position eines 2D-Objekts, wenn es eine Bewegungsfunktion hat. Es wird auch der Zähler der Aktualisierungsrate von der Animation eines 2D-Objekts aktualisiert.

- ***teilt den Aktivierungsmanger das Anvisieren der Schaltflächen mit.***
  Es wird bei jeder Schaltfläche des Menüs überprüft, ob sie augenblicklich anvisiert ist. Wenn das der Fall ist, wird dem zuständigen Aktivierungsmanager die aktuelle Systemzeit mitgeteilt, sodass aus dieser und der vorherigen Systemzeit des Anvisierens die Latenz berechnet werden kann. Die Häufigkeit des Anvisierens und die Summe der Latenzzeiten entscheiden in dem Aktivierungsmanager, ob die Funktion einer Schaltfläche ausgeführt werden muss.

5.4.5.2.2. Methode „*gameRender*"

In der Methode „*gameRender*" wird das Bild vorgerendert, das die 2D-Welt repräsentiert. Aber das ist nicht alles, was in der Methode passiert. Es wird für die Optimierung des Renderings gesorgt.
Das Konzept der Optimierung des Renderings (siehe Abbildung 5.18) ist ganz einfach. Aus einer gegebenen Frequenz, z. B. 50 Hz, resultiert über die Berechnung (***1000ms / 50Hz***) die Dauer des Renderings 20ms, die exakt eingehalten werden muss. Sonst lässt man einen Vorgang des Renderings aus und es wird natürlich gemerkt, wie viel Mal ausgelassen wurde. Damit es nicht zum Stottern kommt, lässt man in einer Folge maximal nur eine bestimmte Anzahl der Renderingsvorgänge aus. Um zu wissen, wie lange ein Renderingsvorgang dauert, wird nach jedem Renderingsvorgang die Systemzeit abgespeichert. Wenn man zu lange für das Rendering gebraucht hat, wird der nächste Vorgang des Renderings vermieden. Falls der letzte Renderingsvorgang weniger als 20ms gedauert hat, muss der Animationsthread schlafengelegt werden.

```
private void gameRender() {
    int clkLength = 1000 / fpsHrz;
    long toWait = System.currentTimeMillis() - millis;
    if (skippedFrames == maxSkippedFrames || toWait <= clkLength) {
        skippedFrames = 0;
        //Rendering
        try {
            toWait = System.currentTimeMillis() - millis;
            if (toWait < clkLength) {
                Thread.sleep(clkLength - toWait);
            }
        } catch (InterruptedException ex) {
            parentME.setInfo(getClass().getName() + ":" + ex);
            Logger.getLogger(MEPlugInSkyDefendor.class.getName()).log(Level.SEVERE, null, ex);
        }
    } else {
        skippedFrames++;
    }
    millis = System.currentTimeMillis();
}
```

Abbildung 5.18: Optimierung des Renderingsvorgangs in der Methode „gameRender"

Bei dem Renderingsvorgang werden alle 2D-Objekte an der Position eines 2D-Objekts aufeinander gezeichnet.

5.4.5.2.3. Methode „*paintScreen*“

Die Methode „*paintScreen*“ (siehe Abbildung 5.19) zeichnet in die grafische Komponente des Plug-Ins das Bild, das in der Methode „*gameRende*r“ vorgerendert wurde.
Es wird zuerst eine Instanz der abstrakten Klasse „*Graphics*“ erstellt. Sie erlaubt die Methoden zum Zeichen zu verwenden. Danach wird in sie die Referenz auf die grafische Komponente des Plug-Ins geschrieben. Als Nächstes wird in einem threadsicheren Kontext überprüft, ob die grafische Komponente des Plug-Ins und das vorher gerenderte Bild in Ordnung sind. Erst danach wird das Bild in die grafische Komponente des Plug-Ins gezeichnet. Anschließend wird der grafische Zustand des Plug-Ins mit dem Aufruf der Methode „*sync*“ aus dem Package „*Toolkit*“ synchronisiert. Zuletzt werden alle angelegten Ressourcen der Klasse „*Graphics*“ und die grafische Komponente des Plug-Ins freigegeben, was durch den Aufruf der Methode „*dispose*“ stattfindet.

```
private void paintScreen() {
    Graphics g;
    try {
        g = this.getGraphics();
        synchronized (this) {
            if ((g != null) && (dbImage != null)) {
                g.drawImage(dbImage, 0, 0, null);
            }
        }
        Toolkit.getDefaultToolkit().sync();
        g.dispose();
    } catch (Exception ex) {
        if (active) {
            parentME.setInfo(getClass().getName() + ":"
                    + "Graphics context error: " + ex);
        }
    }
}
```

Abbildung 5.19: Methode „paintScreen“ des Plug-Ins „Sky Defendor“

5.5. Plug-In: „Simple Eye View” (Textdokument-Reader)

„Simple Eye View” ist ein Textdokument-Reader, mit dem man SEVD-Textdokumente öffnen und mit den Augen steuernd lesen kann. Ein SEVD-Textdokument ist ein eigenes Textdokumentformat, dass in UTF8 (**U**nicode **T**ransformation **F**ormat, **8** Bit) abgespeichert wird und eine erweiterte Zeichensatzwahl erlaubt.
Das Plug-In „Simple Eye View“ ist sehr ähnlich dem Plug-In „Sky Defendor“ aufgebaut, da es auch mit dem 2D-Rendering realisiert ist. Einige Methoden wurden vollständig übernommen und andere teilweise modifiziert. Das Plug-In besteht aus zwei Views. Das erste View ist das Hauptmenü, wo man Dokumente zum Öffnen selektieren kann und das zweite View ist das Dokumentenansichtsmenü, wo man ein geladenes Dokument lesen und blättern kann. Das Plug-In enthält nur passive 2D-Objekte, die sich nicht bewegen und nicht animiert sind. Die 2D-Objekte sind nur als Schaltflächen da.

Der Aufbau des Teilabschnitts ist folgendermaßen aufgebaut:
- ***Klasse „SEVDoc", die die SEVD-Textdokumente umhüllt***
- ***Importe und Klassenvariablen der Klasse „MEPlugInSimpleEyeView"***
- ***Initialisierung der Views***
- ***Animationsthread***
    - ***Methode „gameUpdate"***
    - ***Methode „gameRender"***

Aspekte der Realisierung, auf die nicht eingegangen wird:
- Der Konstruktor der Klasse „*MEPlugInSimpleEyeView*" unterscheidet sich von dem Konstruktor der Klasse „*MEPlugInSkyDefendor*" nur um eine Codezeile und zwar, dass das Plug-In eine andere Hintergrundfarbe gesetzt bekommt.
- Einige der Hilfsmethoden der Klasse „*MEPlugInSimpleEyeView*" sind den Hilfsmethoden der Klasse „*MEPlugInSkyDefendor*" sehr ähnlich.
- Der Datagetterthread ist fast identisch mit dem Thread aus der Klasse „*MEPlugInSkyDefendor*.
- Die Implementierung der Methode „*paintScreen*" ist die Gleiche.

Die Implementierung von den ausgeblendeten Methoden kann bei einem bestehenden Interesse von der beigelegten CD-ROM angeschaut bzw. bezogen werden.

5.5.1. Klasse „*SEVDoc*" – Umhüllung für die SEVD-Textdokumente

Die Klasse „*SEVDoc*" (siehe Abbildung 5.20) ist eine Umhüllung für die SEVD-Textdokumente, die die Dokumente in dem UTF8-Zeichensatzformat einliest und sowohl den Titel als auch den Text in die dafür definierten Felder speichert.
Die Importe der Klasse sind hauptsächlich aus dem Package „*java.io*", da die SEVD-Textdokumente als ein IO-Stream aus einem lokalen Dateisystem ausgelessen werden.
Es gibt nur zwei Klassenvariablen:

- ***String titel*** – die Variable ist von der Klasse „*String*" und repräsentiert eine Zeichenkette. Sie enthält den Titel des Textdokuments.

- ***List<String> text = new ArrayList<String>()*** – die Variable ist von der Klasse „*ArrayList*" und mit der Klasse „*String*" typisiert. Die Liste enthält einen eingelesenen Text eines SEVD-Textdokuments, der zeilenweise gespeichert wird (ein Eintrag der Liste ist eine Textzeile).

Die Klasse hat nur einen Konstruktor und zwei Methoden zum Abfragen des eingelesenen Titels und des Textes.
Der Konstruktor ist das Wichtigste an der Klasse, da darin eine übergebene Datei eingelesen wird, die den Inhalt des SEVD-Textdokuments enthält. Die Datei wird über die Klasse „*InputStreamReader*" eingelesen. Der eingelesene Text wird in eine Variable der Klasse „*StringBuilder*" gespeichert, da „*StringBuilder*" viel schneller als „*String*" konkateniert werden kann. Wenn die Datei vollständig ausgelesen ist, dann wird der Inhalt der Datei als eine Zeichenkette im UTF8-Format in die Felder für den Text und den Titel geschrieben. Als Letztes wird die abschließende Zeile des Texts überprüft, ob sie eine leere Zeile ist. Falls dem so ist, wird sie entfernt.
Die Methode „*getTitel*" gibt den Titel des SEVD-Textdokuments als eine Zeichenkette zurück.
Die Methode „*getText*" gibt den Text des SEVD-Textdokuments als eine Liste von Zeichenketten zurück.

```java
package multieye.plugins.simpleeyeview;

import java.io.File, FileInputStream, FileNotFoundException,
import java.io.IOException, InputStreamReader;
import java.nio.charset.Charset;
import java.util.ArrayList, List, logging.Level, logging.Logger;

/**
 *
 * @author Alexej Tietz
 */
public class SEVDoc {
    private String titel;
    private List<String> text = new ArrayList<String>();

    public SEVDoc(File file, String charset) throws FileNotFoundException {
        try {
            InputStreamReader isr = new InputStreamReader(
                    new FileInputStream(file), Charset.forName(charset));
            int x = -1;
            StringBuilder temp = new StringBuilder();
            while ((x = isr.read()) != -1) {
                temp.append((char) x);
            }
            byte[] utf8 = temp.toString().getBytes(charset);
            String s[] = new String(utf8, charset).split("\n");
            for (int i = 1; i < s.length; i++) {
                if (i == 1) {
                    titel = s[i];
                } else if (i != 0 && i != 2) {
                    text.add(s[i]);
                }
            }
            if (text.get(text.size() - 1).trim().equals("")) {
                text.remove(text.size() - 1);
            }
        } catch (IOException ex) {
            Logger.getLogger(SEVDoc.class.getName()).
                    log(Level.SEVERE, null, ex);
        }
    }

    public String getTitel() {
        return titel;
    }

    public List<String> getText() {
        return text;
    }
}
```

Abbildung 5.20: Klasse „*SEVDoc*“ – Implementierung der Umhüllung für die SEVD-Textdokumente

5.5.2. Importe und Klassenvariablen der Klasse „*MEPlugInSimpleEyeView*"

Nachdem die Umhüllung der SEVD-Textdokumente bekannt ist, kann man mit der Realisierung des Plug-Ins weitermachen.

Als Erstes kommen die Importe der Klassen des Plug-Ins „*MEPlugInSimpleEyeView*" (siehe Abbildung 5.21), die bei dem Plug-In „Simple Eye View" genutzt werden. Die Importe sind in einer abgekürzten Form, in der sie sonst von einem Compiler nicht akzeptiert werden.

```
package multieye.plugins;

import java.awt.Color, Component, Dimension, Font, Graphics;
import java.awt.Image, Point, Toolkit;
import java.awt.event.KeyAdapter, KeyEvent, MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.image.BufferedImage;
import java.io.ByteArrayInputStream, DataInputStream;
import java.io.File, FileFilter;
import java.net.URL;
import java.util.ArrayList, List;
import java.util.logging.Level, Logger;
import javax.swing.JFrame, JPanel;
import multieye.MultiEye;
import multieye.connectors.MEConnector;
import multieye.plugins.simpleeyeview.SEVDoc;
import multieye.utility.BufImModification, Converter;
import multieye.utility.ModDimension, ResizeFactor;
```

Abbildung 5.21: Importe der Klasse „*MEPlugInSimpleEyeView*"

Als Nächstes folgen die Klassenvariablen der Klasse „*MEPlugInSimpleEyeView*" (siehe Abbildung 5.22). Viele der Klassenvariablen heißen genauso und haben die gleiche Bedeutung wie die Klassenvariablen der Klasse „*MEPlugInSkyDefendor*" (Kapitel 5 Teilabschnitt 4.2). Deswegen werden nur die neugekommenen Klassenvariablen mit ihre Bedeutung für die Klasse „*MEPlugInSimpleEyeView*" erläutert:

- ***BufferedImage emptyPage*** – die Variable ist von der Klasse „*BufferedImage*" und repräsentiert ein Bild, das mit Hilfe eines Zwischenspeichers gepuffert wird. Sie enthält das Hintergrundbild für das Blatt, auf dem der Text eines SEVD-Textdokuments gezeichnet wird.

- ***List<SEVDoc> docs = new ArrayList<SEVDoc>()*** – die Variable ist von der Klasse „*ArrayList*" und mit der Klasse „*SEVDoc*" typisiert. Die Liste enthält alle gefundenen SEVD-Textdokumente, die mit der Klasse „*SEVDoc*" umhüllt sind.

- ***int actualDoc = 0, actualPage = 0, linesAtPage = 15*** – die Variablen sind von der Klasse „*Integer*" und repräsentieren je eine ganze Zahl. Sie enthalten: den Index und die Seite des aktuell geöffneten Dokuments, sowie die Anzahl der Textzeilen pro Seite.

```
/**
 * @author Alexej Tietz
 */
public class MEPlugInSimpleEyeView extends JPanel implements MEPlugIn {
    private Graphics dbg;
    private Image dbImage = null;
    private int fontSize = 18;
    private ResizeFactor resizeFactor;
    private BufferedImage emptyPage;
    private List<SEVDoc> docs = new ArrayList<SEVDoc>();
    private int actualDoc = 0;
    private int actualPage = 0;
    private int linesAtPage = 15;
    private List<String> continueAnimation = new ArrayList<String>();
    private List<String> menuAnimation = new ArrayList<String>();
    private List<String> dokOpenAnimation = new ArrayList<String>();
    private List<String> upAnimation = new ArrayList<String>();
    private List<String> downAnimation = new ArrayList<String>();
    private List<String> leftAnimation = new ArrayList<String>();
    private List<String> rightAnimation = new ArrayList<String>();
    private List<String> endAnimation = new ArrayList<String>();
    private Dimension buttonsSize = new Dimension(347, 120);
    private Dimension buttonsUpDownSize = new Dimension(105, 217);
    private Dimension buttonsLeftRightSize = new Dimension(140, 77);
    private List<Animatable> buttonsMenu = new ArrayList<Animatable>();
    private List<Animatable> buttonsView = new ArrayList<Animatable>();
    private List<ActivationManager> actsMenu = new ArrayList<ActivationManager>();
    private List<ActivationManager> actsView = new ArrayList<ActivationManager>();
    private List<String> aimAnimation = new ArrayList<String>();
    private Dimension aimSize = new Dimension(75, 75);
    private AnimatableWorldObject aim;
    protected boolean active = false;
    protected boolean pause = false;
    protected boolean continueView = false;
    private int fpsHrz = 50;
    private static int maxSkippedFrames = 3;
    private static int skippedFrames = 0;
    private int mouseMaxHits = fpsHrz * 3 / 2;
    private int mouseOverTimeMax = 1500;
    private int missMouseResetTime = 900;
    private int mouseTolerance = 85;
    final private MEPlugInSimpleEyeView simlpeEyeView = this;
    private ByteArrayInputStream arrayInputStream;
    private DataInputStream dataInputStream;
    private Dimension developedPanelSize = new Dimension(1019, 741);
    private Dimension runDimension = new Dimension(1019, 741);
    private BufferedImage progIcon = BufImModification.createBI(
            this.getClass().getResource("simpleeyeview/SimpleEyeViewProgIcon.png"));
    private int messageType, x = 1, y = 1, latency;
    private long millis = System.currentTimeMillis();
    private Point correction;
    MEConnector connector;
    JFrame parentJF;
    MultiEye parentME;
    KeyAdapter keyHandler;
    MouseAdapter testMousAdapter = new MouseAdapter() {
        ...
    };
```

Abbildung 5.22: Klassenvariablen der Klasse „*MEPlugInSimpleEyeView*“

- ***List<String> continueAnimation, menuAnimation, dokOpenAnimation, downAnimation, leftAnimation, rightAnimation, endAnimation = new ArrayList<String>()*** – die Variablen sind von der Klasse „*ArrayList*“ und mit der Klasse „*String*“ typisiert. Die Listen enthalten die Verweise auf die Bilder der Schaltflächen: Fortsetzen, Menü, Dokument Öffnen, vorheriger Dokument, nächster Dokument, vorherige Seite des aktuellen geöffneten Dokuments, nächste Seite des aktuellen geöffneten Dokuments und Beenden.

- ***Dimension buttonsSize = new Dimension(347, 120), buttonsUpDownSize = new Dimension(105, 217), buttonsLeftRightSize = new Dimension(140, 77)*** – die Klasse der Variablen repräsentiert eine Dimension. Sie enthalten die Größen (Höhe und Breite) der Schaltflächen in den Menüs.

- ***List<Animatable> buttonsMenu, buttonsView = new ArrayList<Animatable>()*** - die Variablen sind von der Klasse „*ArrayList*“ und mit dem Interface „*Animatable*“ typisiert. Die Listen enthalten alle Schaltflächen der Menüs.

- ***List<ActivationManager> actsMenu, actsView = new ArrayList<ActivationManager>()*** – die Variablen sind von der Klasse „*ArrayList*“ und mit der Klasse „*ActivationManager*“ typisiert. Die Listen enthalten die Aktivierungsmanager, die die Aktionen der Schaltflächen in den Menüs ausführen.

- ***final MEPlugInSimpleEyeView simlpeEyeView = this*** – die finale Variable ist von der Klasse „*MEPlugInSimpleEyeView*“ und repräsentiert das Plug-In „Simple Eye View“. Die Variable verweist auf die Instanz des eigenen Objekts der Klasse.

5.5.3. Initialisierung der Views

Das Plug-In hat zwei Views, die in vier Phasen initialisiert werden.
Zuerst werden die 2D-Objekte der Schaltflächen initialisiert. Sie bekommen die Positionierung und die Bilder für die Schaltflächen zugewiesen.
Als Nächstes werden die Aktivierungsmanager für die Schaltflächen angelegt. Sie bekommen die Aktionen der Schaltflächen zugewiesen.
Danach werden die 2D-Objekte der Schaltflächen an die Ausführungsauflösung angepasst, indem die Bilder der Schaltflächen skaliert werden. Alle drei Initialisierungsphasen sind den Initialisierungsvorgängen (Kapitel 5 Teilabschnitt 4.3 und Teilabschnitt 4.4) des Plug-Ins „Sky Defendor“ sehr ähnlich und wurden in den vorherigen Abschnitten schon behandelt.
Zum Schluss wird die Initialisierung von SEVD-Textdokumenten durchgeführt (siehe Abbildung 5.23), bei der das Dokumentverzeichnis mit Hilfe eines Dateifilters nach Dateien mit der Endung „***sevd***“ durchsucht wird. Aus den gefundenen Dateien werden die Instanzen der Klasse „*SEVDoc*“ erstellt, die die SEVD-Textdokumente umhüllen. Die erzeugten Instanzen „*SEVDoc*“ werden letztendlich zu einer Liste zusammengefügt.

```
void initViewer() {
    try {
        File dir = new File("./modules/docs/");
        FileFilter ff = new FileFilter() {
            public boolean accept(File pathname) {
                return pathname.getName().matches(".*\\.sevd");
            }
        };
        for (File f : dir.listFiles(ff)) {
            docs.add(new SEVDoc(f, "UTF-8"));
        }
    } catch (Exception ex) {
        parentME.setInfo(getClass().getName() + ":" + ex);
        Logger.getLogger(getClass().getName()).
                log(Level.SEVERE, null, ex);
    }
}
```

Abbildung 5.23: Initialisierung von den SEVD-Textdokumenten

5.5.4. Animationsthread

Der Animationsthread des Plug-Ins „Simple Eye View" hat dieselbe Struktur wie der Animationsthread des Plug-Ins „Sky Defendor". Der Thread wird auch durch drei Phasen (Kapitel 5 Teilabschnitt 4.5.2) beschrieben/implementiert.

5.5.4.1. Methode „*gameUpdate*"

Die Methode „*gameUpdate*" (siehe Abbildung 5.24) bei dem Animationsthread des Plug-Ins „Simple Eye View" ist viel einfacher als bei dem Plug-In „Sky Defendor" implementiert, da bei dem Plug-In „Simple Eye View" nur auf Aktionen von den Schaltflächen in den Views reagiert werden muss.
Es gibt zwei Views bei dem Plug-In „Simple Eye View": das Hauptmenü und die Dokumentansicht. Bei den Ansichten muss auf die Aktionen der Schaltflächen reagiert werden, was durch die Methode „*gameUpdate*" übernommen wird.
Zuerst wird auf das Anvisieren der Schaltflächen in der Dokumentenansicht überprüft. Wenn das der Fall ist, wird der zuständige Aktivierungsmanger darüber informiert.
Als Letztes werden die Schaltflächen im Menü des Plug-Ins „Simple Eye View" auf das Anvisieren überprüft und wenn es nötig ist, wird der verantwortliche Aktivierungsmanager darüber in Kenntnis gesetzt.

```
private void gameUpdate() {
    if (active && !pause && simlpeEyeView.hasFocus()) {
        for (int i = 0; i < buttonsView.size(); i++) {
            if (buttonsView.get(i).getGeomethry(0).
                    contains(aim.getActualPosition())) {
                actsView.get(i).addHit(System.currentTimeMillis());
            }
        }
    } else if (active && pause && simlpeEyeView.hasFocus()) {
        for (int i = 0; i < buttonsMenu.size(); i++) {
            if (!continueView && i == 0) {
                continue;
            }
            if (buttonsMenu.get(i).
                    getGeomethry(0).contains(aim.getActualPosition())) {
                actsMenu.get(i).addHit(System.currentTimeMillis());
            }
        }
    }
}
```

Abbildung 5.24: Methode „*gameUpdate*“ des Plug-Ins „Simple Eye View“

5.5.4.2. Methode „*gameRender*“

Bei der Methode „*gameRender*“ des Plug-Ins „Simple Eye View“ wird die Optimierung des Renderings (Kapitel 5 Teilabschnitt 4.5.2.2) auf die gleiche Weise, wie bei dem Plug-In „Sky Defendor“ durchgeführt. Das gerenderte Bild besteht nur aus den Abbildungen der Schaltflächen und dem Text eines SEVD-Textdokuments.

6. Zusammenfassung und Ausblick

Dieses Kapitel schließt die Bachelorarbeit ab, indem es eine Bewertung der gewonnenen Erfahrungen vorstellt und eine Zusammenfassung über die entstandenen Ergebnisse darstellt. Abschließend wird ein Ausblick auf einige Verbesserungen und Erweiterungen gegeben, die im Rahmen dieser Arbeit wegen des zeitlichen Aspekts nicht möglich waren.

6.1. Bewertung

Bei der Erarbeitung der Anwendungssuite zeigte sich, dass viele Problematiken erst bei der Realisierung des Konzeptes entstehen, auch wenn das Konzept nach Softwareengineeringsmaßstäben entworfen war. Eine der entstandenen Problematiken, die Deadlock-Problematik, wurde in einem der vorangegangenen Teilabschnitte (Kapitel 4, Teilabschnitt 8) beschrieben. Manche Anwendungsfälle sind erst nach dem Erreichen der Meilensteine des Konzeptes zu Stande gekommen. Diese Anwendungsfälle verlangten anschließend nach einer Erweiterung oder einer Veränderung des Entwurfes.

Nach der Entwicklung des Plug-Ins „Sky Defendor" hat sich gezeigt, dass ein Eye-Tracker eine gute Unterhaltungsquelle sein kann, die aber einiger Nachbesserungen bedarf, damit das Gerät für jedermann geeignet ist. Auf diese Nachbesserungen wird im nächsten Teilabschnitt des Kapitels eingegangen.

Das Plug-In „Simple Eye View" beweist, dass ein Eye-Tracker nicht nur für Spaß sorgen kann, sondern dass das Gerät von bestimmten Personengruppen verwendet werden kann, die gewisse gesundheitliche Einschränkungen bei der Bewegung haben.

Allgemein kann noch hinzugefügt werden, dass die Entwicklung der Anwendungssuite, die ein technisches Gerät verwendet, ein höheres Abstraktionsniveau verlangt. Dabei dürfen einige technische Aspekte nicht aus den Augen verloren gehen, wie z. B. die Byte-Reihenfolge des Inhalts einer UDP-Nachricht.

6.2. Zusammenfassung

Das Hauptziel dieser Bachelorarbeit war eine Anwendungssuite für einen Eye-Tracker (Tobii X120 Eye Tracker) zu erstellen, die leicht für einen anderen Eye-Tracker angepasst werden kann.

Die Anpassungsfähigkeit wurde durch das Verbindungsmodul (Connector) realisiert. Um einen andern Eye-Tracker verwenden zu können, müsste von einem Softwareentwickler ein Connector für den gegebenen Eye-Tracker implementiert werden. Danach wäre es möglich die bestehende Software der Anwendungssuite zu verwenden. Der Connector ist letztendlich nicht nur ein Verbindungsmodul, er ist auch ein Adapter.
Die Erschaffung einer Anwendungssuite ist auch erfolgreich gelungen. Es wurde gezeigt, dass ein Eye-Tracker für verschiedene Softwaretypen verwendet werden kann.
Durch die Erstellung eines Aktivierungsmanagers ist es möglich geworden, einige Benutzerinteraktionen zu erkennen, die durch die menschlichen Augen ausgelöst werden.
Das Schwierigste war der Entwurf und die Realisierung der API – „Multi Eye“, da gewährleistet werden musste, dass ein Entwickler bei der Fertigung eines Plug-Ins durch die API keinen Tunnelblick bekommt. Durch die Einfachheit der API und die wenigen Vorschriften bei der Erstellung eines Plug-Ins kann ein Entwickler seinen Ideen freien Lauf lassen. Das Plug-In ist der API untergeordnet, da es durch sie erstellt, gestartet und entsorgt wird. Während seiner Laufzeit bekommt es fasst die volle Macht und gibt keinem anderen Programm nach.

Es gibt aber einige Probleme bei dem Eye-Tracking, z. B. dass man keine kleinen Objekte (unter 24*24 Pixel) anvisieren kann und dass die menschlichen Augen nie an einem Punkt bleiben, sondern ständig in Bewegung sind. Es existieren verschiedene Algorithmen, mit deren Hilfe sich die Daten von einem Eye-Tracker filtern lassen. Dadurch können große Blicksprünge (Abweichungen) vermieden werden, was das ständige Bewegen der Augen nicht mehr so problematisch macht. Aber es bleibt das Problem des Anvisierens von kleinen Objekten. Zusätzlich muss man die Augen trainieren, um die Objekte in einem beweglichen Bild schnell und genau anschauen zu können.
Ohne ein automatisches Profiling wird es schwierig einen Eye-Tracker von mehreren Benutzern zu verwenden.
Als Letzes könnte man hinzufügen, dass die Verwendung eines Gerätes dieser Art das Leben von vielen Menschen erleichtern kann. Aber es bräuchte viel mehr zeitlichen Aufwand und würde den Rahmen dieser Bachelorarbeit sprengen.

6.3. Ausblick

Um einen Eye-Tracker als ein interaktives Medium zu verwenden, müssten ein paar Automatisierungen vorgenommen werden.
Als Erstes könnte eine automatische Kalibrierung eingeführt werden, die die lästige manuelle Kalibrierung ersetzen würde.
Die Benutzbarkeit des Geräts könnte durch eine automatische Profilerkennung gesteigert werden, sodass die API bestimmte Einstellungen, z. B. den passenden Filtrierungsalgorithmus, selbst wählen kann.
Um das Gerät wirklich betriebssystemunabhängig zu machen, müssten neben dem Connector die Hilfsklassen des Herstellers in eine betriebssystemunabhängige Sprache (z. B. Java) umgeschrieben werden. Die mitgelieferten Hilfsklassen für den Tobii X120 Eye-Tracker sind nur unter Windows verwendbar, daher ist man auf Windows als einen Zwischenadapter angewiesen. Der Versuch einen betriebssystemunabhängigen Adapter für die Low-Level-Verbindung zu erstellen, ist gescheitert, da die Beschreibung der Low-Level-Verbindung in „***Developer's Guide***“ von dem Hersteller Tobii Technology AB zu abstrakt und knapp gehalten ist.
Um einen Computer zu bedienen, reicht es nicht nur eine Maus zu haben. Die meisten Eingaben verlangen die Benutzung einer Tastatur. Genauso kann man mit einem Eye-Tracker allein keinen PC steuern. Es bedarf einer alternativen Steuerungsquelle. Es käme beispielsweise eine Sprachsteuerung infrage, die die Interaktion mit einem Computer noch einmal vereinfachen würde.

Anhang: Ergebnisse der praktischen Arbeit auf CD-ROM

Dieser Arbeit enthält eine CD-ROM mit folgenden Inhalten:

- Ordner „***API***“ – Kompilierte API – „Multi Eye“, UDP-Connector und drei Plug-Ins („Sky Defendor“, „Simple Eye View“ und „Hallo Welt!“).
- Ordner „***Projekt***“ – Projektdateien aus NetBeans 6.9.1.
- Ordner „***PDF***“ – Diese Arbeit in PDF-Format.
- Ordner „***EyeTrackerFilter***“ – Ein Filtermodul, das vom Lorenz Barnkow bei seiner Abschlussarbeit – „*Ein Eye-Tracking-basiertes System für Usability-Untersuchungen von benutzeradaptiven TV-Newstickern*“ entwickelt wurde.
- Ordner „***eBooks***“ – Verwendete Literatur.

Literaturverzeichnis

Davison 2004a
Davison, Andrew: *Java Prog. Techniques for Games. Chapter 1. An Animation Framework* Stand: 2004 http://fivedots.coe.psu.ac.th/~ad/jg/ch1/ch1.pdf Abruf: 2010-10-03

dpunkt.Verlag 2002
dpunkt.Verlag : *Klasse java.awt.Graphics* Stand: 2002 http://www.dpunkt.de/java/Referenz/Das_Paket_java.awt/59.html Abruf: 2010-12-16

Kumar 2007a
Kumar, Manu: *GAZE-ENHANCED USER INTERFACE DESIGN*. Stand; 2007 http://hci.stanford.edu/research/GUIDe/publications/Manu%20Kumar%20Dissertation%20-%20Gaze-enhanced%20User%20Interface%20Design.pdf Abruf: 2010-09-22, S.10-12

Lischka 2008a
Lischka, Thomas: *Untersuchung eines Eye Tracker Prototypen zur automatischen Operationsmikroskopsteuerung*. Stand: 2007 http://www.sub.uni-hamburg.de/opus/volltexte/2008/3672/ Abruf: 2010-09-24, S.7

Milekic 2003a
MILEKIC, Slavko: *The more you look the more you get: Intention-based interface using gaze-tracking*. In: *Museums and the Web*, Stand: 2003 http://www.uarts.edu/faculty/smilekic/paper/milekicMW03.pdf Abruf: 2010-09-17, S. 22

Oracle 2010a
Oracle : *The Java^{TM} Tutorials - Java 2D Rendering* Stand: 2010
http://download.oracle.com/javase/tutorial/2d/overview/rendering.html Abruf: 2010-09-15

Oracle 2010b
Oracle : *Java^{TM} 2 Platform Std.Ed. v1.4.2 - java.awt.image.BufferedImage* Stand: 2010 http://download.oracle.com/javase/1.4.2/docs/api/java/awt/image/BufferedImage.html Abruf: 2010-09-14

Oracle 2010c
Oracle : *Java^{TM} 2 Platform Std.Ed. v1.4.2 - java.awt.Color* Stand: 2010 http://download.oracle.com/javase/1.4.2/docs/api/java/awt/Color.html Abruf: 2010-09-14

Oracle 2010d
Oracle : *Java™ 2 Platform Std.Ed. v1.4.2 - java.awt.geom.AffineTransform* Stand: 2010 http://download.oracle.com/javase/1.4.2/docs/api/java/awt/geom/AffineTransform.html Abruf: 2010-09-15

Oracle 2010e
Oracle : *Java™ 2 Platform Std.Ed. v1.4.2 - java.awt.Polygon* Stand: 2010 http://download.oracle.com/javase/1.4.2/docs/api/java/awt/Polygon.html Abruf: 2010-09-15

Ray 2010a
Rey, Günter D.: *E-Learning. Theorien, Gestaltungsempfehlungen und Forschung*. Stand: 2010-09-17 http://www.elearning-psychologie.de/eyetracker_i.html Abruf: 2010-09-17

Schäfers 2009a
Schäfers, Michael: *Rechnerstrukturen* Stand: 2009 https://pub.informatik.haw-hamburg.de/home/pub/prof/schafers/S09S_RS/RSTI4_no05_vw05_cw17_090422_v06.pdf
Abruf: 2010-09-17, S.61

Tobii 2010a
Tobii Technology AB: *Tobii Glasses, X60 & X120, T60 & T120 Eye-Trackers.* Stand: 2010-09-10 http://www.tobii.com/market_research_usability/products_services/eye_tracking_hardware.aspx
Abruf: 2010-09-10

Tobii 2010b
Tobii Technology AB: *Tobii X60 & X120 Eye-Trackers.* Stand: 2010-06-23 http://www.tobii.com/archive/files/17995/Tobii_TX_Series_Eye_Trackers_product_description.pdf.aspx Abruf: 2010-10-13, S. 9

VIM 2008a
Joint Committee for Guides in Metrology: *International vocabulary of metrology – Basic and general concepts and associated terms (VIM)*, Definition 2.39 Stand: 2008
http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2008.pdf#page=42
Abruf: 2010-09-17

Abbildungsverzeichnis

Abbildung 2.1: Tobii Glasses, X60 & X120, T60 & T120 Eye-Trackers (Tobii 2010a) 4
Abbildung 2.2: Beispielbilder für „dunkle Pupille” (A) und „infrarot-licht beleuchtete Pupille” (B) sowie der Infrarot-Reflektionen auf der Hornhaut (C) (Milekic 2003a) 4
Abbildung 2.3: Byte-Reihenfolge: Little Endian und Big Endian (32Bit-Wert:0xdead0102).......... 5
Abbildung 2.4: Alphawert des Marmorsteins 7
Abbildung 2.5: Animationsschleife-Struktogramm nach Nassi-Shneiderman................................ 7

Abbildung 3.1: Use Case Diagramm der API 10
Abbildung 3.2: Zustandsdiagramm der API.. 11
Abbildung 3.3: Tobii X120 Series Eye Trackers – Abmessungen (Tobii 2010b) 12
Abbildung 3.4: Use Case Diagramm eines Connectors 13
Abbildung 3.5: Zustandsdiagramm eines Connectors.. 14
Abbildung 3.6: Use Case Diagramm eines Plug-Ins 15
Abbildung 3.7: Zustandsdiagramm eines Plug-Ins 17

Abbildung 4.1: Interface des Connectors – „*MEConnector*“ 20
Abbildung 4.2: Interface der API – „*Multi Eye*" 20
Abbildung 4.3: Interface des Plug-Ins – „*MEPlugIn*".. 21
Abbildung 4.4: Beispielcode der Selbst-Adaptivität an eine gegebene Auflösung........................ 23
Abbildung 4.5: Klassenlader – Laden einer Klasse aus einem JAR-Paket 24
Abbildung 4.6: Starten eines Plug-Ins im Fullscreen-Modus.. 25
Abbildung 4.7: Implementierung des UDP-Connectors – „*MEConnectorUDP*“ 28
Abbildung 4.8: Klassendiagramm von API und Connector 31
Abbildung 4.9: Klassendiagramm von API und Plug-In... 32
Abbildung 4.10: Klassendiagramm von Plug-In und Connector 33
Abbildung 4.11: API – „*Multi Eye*" (das Hauptfenster – Logger) 34
Abbildung 4.12: ein Muster der Fehlermitteilung an die API.. 35
Abbildung 4.13: Alternative Quelle (Maus) zu einem Eye-Tracker 35
Abbildung 4.14: Beispiel-Plug-In – „Hallo Welt!“ Teil 1... 36
Abbildung 4.15: Beispiel-Plug-In – „Hallo Welt!“ Teil 2... 37
Abbildung 4.16: Beispiel-Plug-In – „Hallo Welt!“ Teil 3... 38
Abbildung 4.17: Beispiel-Plug-In – „Hallo Welt!“ Teil 4... 39
Abbildung 4.18: Beispiel-Plug-In – „Hallo Welt!“ Teil 5... 39
Abbildung 4.19: UDP-Connector - Methode „*close*" 40
Abbildung 4.20: UDP-Connector - Lösung des Deadlocks 41
Abbildung 4.21: Testklasse für den UDP-Connector „*MEConnectorUDP*“ 43
Abbildung 4 22: Testergebnis der Testklasse „*MEConnectorUDPTest*“...................................... 44

Abbildung 5.1: Zustandsdiagramm eines 2D-Plug-Ins ... 46
Abbildung 5.2: Interface des 2D-Objekts „*Animatable*“ ... 48
Abbildung 5.3: Implementierung des 2D-Objekts „*AnimatableWorldObject*“ Teil 1 ... 49
Abbildung 5.4: Implementierung des 2D-Objekts „*AnimatableWorldObject*“ Teil 2 ... 51
Abbildung 5.5: Implementierung des 2D-Objekts „AnimatableWorldObject“ Teil 3 ... 51
Abbildung 5.6: Implementierung des 2D-Objekts „AnimatableWorldObject“ Teil 4 ... 52
Abbildung 5.7: Implementierung des 2D-Objekts „AnimatableWorldObject“ Teil 5 ... 53
Abbildung 5.8: Interface „*Action*“ eines Callbacks für den Aktivierungsmanager „*ActivationManager*“ ... 54
Abbildung 5.9: Implementierung des Aktivierungsmanagers „ActivationManager“ ... 55
Abbildung 5.10: Implementierung des Abwehrturms „*SDDefendor*“ Teil 1 ... 57
Abbildung 5.11: Implementierung des Abwehrturms „*SDDefendor*“ Teil 2 ... 58
Abbildung 5.12: Importe der Klasse „*MEPlugInSkyDefendor*“ ... 59
Abbildung 5.13: Klassenvariablen der Klasse „*MEPlugInSkyDefendor*“ Teil 1 ... 60
Abbildung 5.14: Klassenvariablen der Klasse „*MEPlugInSkyDefendor*“ Teil 2 ... 61
Abbildung 5.15: Konstruktor der Klasse „*MEPlugInSkyDefendor*“ ... 66
Abbildung 5.16: Winkelberechnung des Abwehrgeschützes ... 67
Abbildung 5.17: Datagetterthread der Klasse „MEPlugInSkyDefendor“ ... 69
Abbildung 5.18: Optimierung des Renderingsvorgangs in der Methode „gameRender“ ... 71
Abbildung 5.19: Methode „paintScreen“ des Plug-Ins „Sky Defendor“ ... 72
Abbildung 5.20: Klasse „*SEVDoc*“ – Implementierung der Umhüllung für die SEVD-Textdokumente ... 74
Abbildung 5.21: Importe der Klasse „*MEPlugInSimpleEyeView*“ ... 75
Abbildung 5.22: Klassenvariablen der Klasse „*MEPlugInSimpleEyeView*“ ... 76
Abbildung 5.23: Initialisierung von den SEVD-Textdokumenten ... 78
Abbildung 5.24: Methode „*gameUpdate*“ des Plug-Ins „Simple Eye View“ ... 79

Tabellenverzeichnis

Tabelle 3.1: Bezug der Interaktionen auf Anforderungen der API 11
Tabelle 3.2: Tobii X120 Series Eye Trackers – Technische Spezifikationen (Tobii 2010b).......... 12
Tabelle 3.3: Bezug der Interaktionen auf Anforderungen eines Connectors 13
Tabelle 3.4: Bezug der Interaktionen auf Anforderungen eines Plug-Ins 16

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28. März 2011
Ort, Datum

Unterschrift