Hochschule Neubrandenburg
University of Applied Sciences

Hochschule Neubrandenburg

Studiengang Geoinformatik

Entwicklung eines Kommandozeilen-Interfaces für den VizzAnalyzer™

Bachelorarbeit

vorgelegt von: *Daniel Rohde*

Zum Erlangen des akademischen Grades

„Bachelor of Engineering“ (B.Eng.)

Erstprüfer: Prof. Dr.-Ing. Andreas Wehrenpfennig

Zweitprüfer: Phil. Lic. Rüdiger Lincke

Bearbeitungszeitraum: 08. August 2008 – 29. September 2008

urn:nbn:de:gbv:519-thesis2008-0104-4

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Neubrandenburg, den 29.09.2008

Unterschrift

Danksagung

An dieser Stelle möchte ich allen Personen danken, die mir nicht nur bei der Erarbeitung der Arbeit eine Hilfe waren, sondern auch mich auch an meinen mentalen Tiefpunkten immer wieder motivierten. Mein ganz besonderer Dank geht dabei an Rüdiger Lincke, der mir in allen Fragen der Entwicklung der Applikation immer mit gutem Rat zur Seite stand. Er lieferte mit auch immer wieder neue Ansatzpunkte, die ich in meine Arbeit mit aufnehmen konnte.
Besonders hervor heben möchte ich auch meine Mutter, die immer an mich glaubte, als ich schon den Glauben an die Fertigstellung der Arbeit verloren hatte.

Vielen Dank an alle, ihr wart mir eine große Hilfe.

Kurzfassung

Diese Bachelorarbeit befasst sich mit der Neuentwicklung eines Kommandozeilen-Interfaces. Der Hintergrund der Arbeit ist ein an der Universität Växjö, Schweden, entwickeltes Konzept, welches sich mit der Qualitätsanalyse von Softwareprodukten beschäftigt.

Aus der dort vorgenommenen Forschung entstand eine Anwendung, die es einem Entwickler ermöglicht seine Software auf ihre Qualität hin zu untersuchen. Diese Anwendung ist der VizzAnalyzer™ und wird von der Firma Arisa AB vertrieben. Der VizzAnalyzer™ steht schon in verschiedenen Versionen zur Verfügung und das Kommandozeilen-Interface soll nun den Betrieb auf fensterlosen Servern oder Stapelverarbeitungsbetrieb ermöglichen. Dazu müssen Eingaben analysiert und interpretiert werden. Die Arbeit befasst sich mit den verschiedenen Konzepten und Möglichkeiten der Implementierung dieser.

Abstract

This thesis describes the new development of a command-line interface. The background of this assignment is a concept, which was developed at the University of Växjö, Sweden and is concerned with the quality analysis of software products.

Out of this research an application was created. This application gives a developer the possibility to analyze the quality of his software products. This application is called VizzAnalyzer™ and is distributed by the company Arisa AB. The VizzAnalyzer™ is already available in various versions and the command-line interface enables the possibility to analyze software in a server environment. The interface requires the possibility to analyze and interpret any user input. The thesis concerned with various concepts and ways of implementation.

Inhaltsverzeichnis

1 Einführung

In der heutigen technisierten Zeit spielen Computer eine immer wichtigere Rolle, der rasante Fortschritt im Bereich der Technologie führt auch zu einem immer größeren Einsatzfeld von Softwareprodukten. Im Jahr 2003 wurden alleine im Automobilbereich 25 Milliarden Euro für Software ausgegeben, laut einer Studie der Mercer Management Consulting, der Fraunhofer Gesellschaft und Bosch wird das Marktvolumen in dem Bereich bis in das Jahr 2015 auf 133 Mrd. Euro ansteigen [1].
Der steigende Bedarf bringt immer größere und komplexere Softwaresysteme hervor. Mit zunehmendem Konkurrenz- und Zeitdruck kann es jedoch vorkommen, dass die Qualität der Software leidet. Zwei wichtige Qualitätsmerkmale sind die Wartbarkeit und die Erweiterbarkeit. Diese beiden Punkte sind es, die eine Software am Leben halten. Die Voraussetzung dafür ist, dass die Software funktioniert, also das sie arbeitet wie es erwartet wird.
Bill Gates sagte 1978, „640 KByte (Arbeitsspeicher) sollten für jeden genug sein.“. Heute dürfte fast jedem Menschen bekannt sein, dass kaum eine Software mehr mit so geringem Speicher auskommt. Was aus diesem Zitat mitgenommen werden kann, ist dass sich Wünsche und Ansprüche von Kunden ändern. Software sollte dabei in der Lage sein, auch nach der Einführung beim Kunden darauf flexibel zu reagieren. Die Entwicklung von Software kostet vor allem Geld und Zeit, von beidem möchte der Kunde nur wenig investieren. Das Problem besteht auch nach der Einführung beim Kunden weiter. Daher muss die Wartbarkeit und Erweiterbarkeit so kostengünstig und einfach wie nur möglich sein. Denn wenn die Kosten für die Weiterentwicklung die Kosten für eine Neuentwicklung übersteigen, so wird die Software nach ökonomischen Gesichtspunkten nicht mehr lange überleben.
Der VizzAnalyzer™ [2] stellt dem Entwickler nun ein Werkzeug zur Verfügung, mit dessen Hilfe dieser in der Lage ist schon vor der Auslieferung des Softwareproduktes die Qualität zu analysieren. Um gegebenenfalls Änderungen vorzunehmen, welche die Qualität der entwickelten Software verbessern und so möglicherweise auch ihre Lebenszeit verlängern.

Problem

Das zuvor erwähnte Werkzeug, VizzAnalyzer™, steht in mehreren Varianten zur Verfügung. Es kann als ein separates Programm für die Analyse und Visualisierung von Java Systemen und als ein Framework erworben werden. Das Framework kann um neue Analyse- und Visualisierungswerkzeuge, aber auch für andere

Programmiersprachen erweiterbar werden. Für die Steuerung des Programms wird dabei eine grafische Benutzeroberfläche (GUI – Graphical User Interface, grapfische Benutzerschnittstelle) verwendet. Dafür werden Elemente wie Assistenten, Tabellen, Menüs und Grafiken benutzt. Diese Elemente sind jedoch nicht immer vorteilhaft, auch wenn sie die Benutzerfreundlichkeit einer Software steigern. Im Bereich der Serveranwendungen ist es nicht immer möglich oder wünschenswert, mit einem GUI zu arbeiten. Oft wird noch auf Terminals und Konsolen zurückgegriffen, da diese für die Aufgaben ausreichen, und der gleiche Bedienungskomfort auch über Remoteverbindungen (z.B. SSH oder Telnet) zur Verfügung steht. Daher muss auf die Funktionalität des VizzAnalyzer™ in anderer Form zugegriffen werden. Es soll eine Kommandozeilenschnittstelle, die mit der Funktionsbibliothek des VizzAnalyzer™ verbunden ist, geschaffen werden.

Motivation

Diese Arbeit basiert auf den Entwicklungen, die im Rahmen des VizzAnalyzer™ Projektes erstellt wurden. Die Weiterführung des Projektes bringt nun einen neuen Weg hervor, um so die Produktpalette des Unternehmens Arisa AB zuerweitern.
Arisa AB vertreibt den VizzAnalyzer™ in verschiedenen Versionen. Mit der Entwicklung des Kommandozeilen-Interfaces wird ein Schritt eingeleitet, der die Softwarequalitätsanalyse in einer Serverumgebung, auch mit Fernzugriff, zulässt. Der Verzicht auf eine grafische Oberfläche und die Implementierung eines Script-Modus gestattet es dem Nutzer seine Analysen in einen automatisierten Prozess einzubinden.

Ziel

Gegenstand dieser Bachelorarbeit ist es, die zuvor bestimmten Schwierigkeiten mit dem VizzAnalyzer™ GUI zu beseitigen. Dazu muss ein Zugriff auf die Funktionsbibliothek gewährleistet sein. Die neu zu gestaltende Anwendung muss die Fähigkeit besitzen, die vom Nutzer eingegebenen Befehle zu erfassen und der entsprechenden Funktion der Bibliothek zuzuordnen. Bei den verfügbaren Befehlen handelt es sich um vordefinierte Kommandos, welche durchaus von einer Reihe von Pflicht- oder optionalen Parametern begleitet seinen können. Kommt es bei der Eingabe von Befehlen zu Problemen oder gar Fehlern, muss die Applikation dieses erkennen und dem Benutzer entsprechende Hilfetexte zur Verfügung stellen. Programmabstürze aufgrund von fehlerhaften Eingaben müssen auf jeden Fall verhindert werden.

Ebenso wichtig wie die Einzelkommandos ist das Ausführen von ganzen Kommandosequenzen. Diese sollten aus einer Datei heraus ausführbar sein. Diese Sequenzdatei kann dann in einem Scriptmodus als Startparameter der Anwendung mit übergeben werden oder mithilfe eines Befehls zur Laufzeit der Applikation ausgeführt werden. Um die erwähnten Sequenzdateien in korrekter Weise zu erzeugen, muss der Benutzer die Möglichkeit haben, seine Eingaben aufzuzeichnen.
Die Erweiterbarkeit der Anwendung soll so einfach wie möglich sein. Dabei sollte auch nur sehr wenig am eigentlichen Code der Applikation verändert werden müssen.

Inhalt

Kapitel 2 beschreibt grob die Zusammenhänge und die Idee, die hinter dem VizzAnalyzer™ steht. Dieser bildet die Grundlage für die Entwicklung des Kommandozeilen-Interfaces.
Kapitel 3 schafft die Grundlage für die Entwicklung der Applikation. Hier wird detailliert Bezug genommen auf die Anforderungen, die erfüllt werden müssen. Ebenso geht dieser Abschnitt auf die verschiedenen Anwendungsfälle ein mit denen die Applikation konfrontiert werden kann.
Im Kapitel 4 werden die Architektur und das Design vorgestellt. Entscheidungen, diese beiden Punkte betreffend, werden dabei erläutert.
In Kapitel 5 werden Aussagen zur Implementierung des Systems und der Kommandos gemacht. In dem Kapitel wird zu den wichtigsten Grundlagen der Applikation Stellung genommen und die Art und Weise der Implementierung dargelegt und begründet.
Kapitel 6 stellt dar, wie erfolgreich einzelne Teile der Applikation umgesetzt wurden, und zeigt Bereiche auf, deren Weiterentwicklung sinnvoll wäre.
Der Anhang liefert Verweise zu weiteren Dokumenten, die im Verlauf dieser Arbeit erstellt wurden.

2 Hintergrund

Das Programm, für welches das Kommandozeilen-Interface entwickelt werden soll, ist der zuvor schon erwähnte VizzAnalyzer™. In dem folgendem Abschnitt wird die Funktionsweise des VizzAnalyzer™ beschrieben. Bei dem an der Växjö Universität entwickelten VizzAnalyzer™ handelt es sich um ein „reverse engineering framework“ für die Analyse und die Visualisierung von Programmen [8]. Lincke und Löwe haben auf der Grundlage ihrer Arbeit im Bereich der Softwarequalität eine Applikation geschaffen, die Standardqualitäten und Softwaremetriken zusammenführt. Ihre Arbeit orientiert sich dabei an der ISO/IEC 9126 [9]. Diese Norm beschreibt ein Qualitätsmodell für Software und unterscheidet dabei in der Qualität sechs Hauptgruppen. Diese werden präzisiert durch die Einbeziehung weiterer Merkmale. Die Studien von Lincke und Löwe konzentrieren sich dabei auf die inneren Qualitätsmerkmale eines Softwareprodukts. Diese können am Quellcode des zu analysierenden Programmes bestimmt werden. Ihre Bemühungen sind in dem Dokument „Compendium of Software Quality Standards and Metrics” [9] dokumentiert. Aktuell sind dort 37 Softwarequalitätseigenschaften und 23 Softwarequalitätsmetriken beschrieben. Die in dem Werk enthaltenen Informationen können in einer Matrix dargestellt werden. Diese Matrix umfasst die verschiedenen Softwarequalitätseigenschaften und Metriken und kann daher auch als eine Darstellung des Softwarequalitätsmodells betrachtet werden.

Die Softwarequalitätsmatrix (SQX), Abbildung 2-1, zeigt die Beziehungen zwischen den verschiedenen Metriken und den Qualitätsmerkmalen. Die Beziehungen können dabei unterschiedliche Grade annehmen, so können die Metriken im Extremfall eine enge Beziehung (dunkles Grün) oder auch kaum eine Beziehung (dunkles Rot) zu den Qualitätsmerkmalen haben. Zwischen den Extremfällen existieren noch zwei Abstufungen, die in helleren Farben dargestellt sind. Ein Sonderfall besteht, wenn zwischen einer Metrik und einem Qualitätsmerkmal keine Beziehung besteht (Grau) [10].

		Main property	functionality					reliability				re-usability					efficiency			maintainability					portability						
		Sub Property	suitability	accuracy	interoperability	security	compliance	maturity	fault tolerance	recoverability	compliance	understandability	learnability	operability	attractiveness	compliance	time behavior	resource utilization	compliance	analyzability	changeability	stability	testability	compliance	adaptability	installability	co-existence	conformance	replaceability	compliance	VizzAnalyzer
Category	Sub-Category	Metric																													
Complexity	size	LOC			+	+		-	+	+		–	-	-	++		-	-		–	–	-	–		–				-		X
	interface C.	SIZE 2			+	+		-	+	+		–	–	–	++		-	-		–	–	-	–		–				–		X
		NOM			+	+		-	+	+		–	–	–	++		-	-		–	–	-	–		–				–		X
	structural C.	CC			+	+		-	+	+		–	–	–	++		-			–	–	-	–		–				–		X
		WMC			+	+		-	+	+		–	–	–	++		-	-		–	–	-	–		–				–		X
		RFC						-				–	-	-	++		-			–	–	-	–		–				-		X
Architecture & Structure	Inheritance	DIT			+	+		-	+	+		–	–	–	++		-	-		–	–	-	–		–				–		X
		NOC						+				++	++	++	++					-	–	+	-		-				–		X
	Coupling	Ca				+		–		-		–	-	-	-		-	-		-	-	-	-		–				-		X
		CBO				+		–		-		–	-	-	-		-	-		–	–	–	–		–				-		X
		CDBC				+		–		-		–	-	-	–		-	-		–	–	–	–		–				–		X
		CDOC				+		–		-		–	-	-	-		-	-		–	–	–	–		–				-		X
		Ce				+		–		-		–	-	-	-		-	-		–	–	–	–		–				-		X
		CF				+		–		-		–	-	-	-		-	-		–	–	–	–		–				-		X
		DAC				+		–		-		–	-	-	-		-	-		–	–	–	–		–				-		X
		I				+		–		-		–	-	-	-		-	-		–	–	–	–		–				-		X
		LD				-		++		+		++	+	+	+		-	-		++	++	++	++		++				+		X
		MPC				+		–		-		–	-	-	-		-	-		–	–	–	–		–				-		X
		PDAC				+		-				–	-	-	-		-	-		–	–	-	–		–				-		X
	Cohesion	LCOM						–				++	+	+	–		+			++	++	–	++		++				+		X
		ILCOM						–				++	+	+	–		+			++	++	–	++		++				+		X
		TCC						++				–	-	-	++		-			–	–	++	–		–				-		X
Design	Documentation	LOD						-				–	–	–	-					–	–	-	–		–				–		X

Abbildung 2-1: Softwarequalitätsmatrix (SQX)

Die Abbildung 2-2 zeigt den Ablauf einer Softwareanalyse und ihrer Visualisierung von der Extraktion der Daten über die Konvertierung in ein brauchbares Format und letzten Endes die Analyse und Darstellung [8].

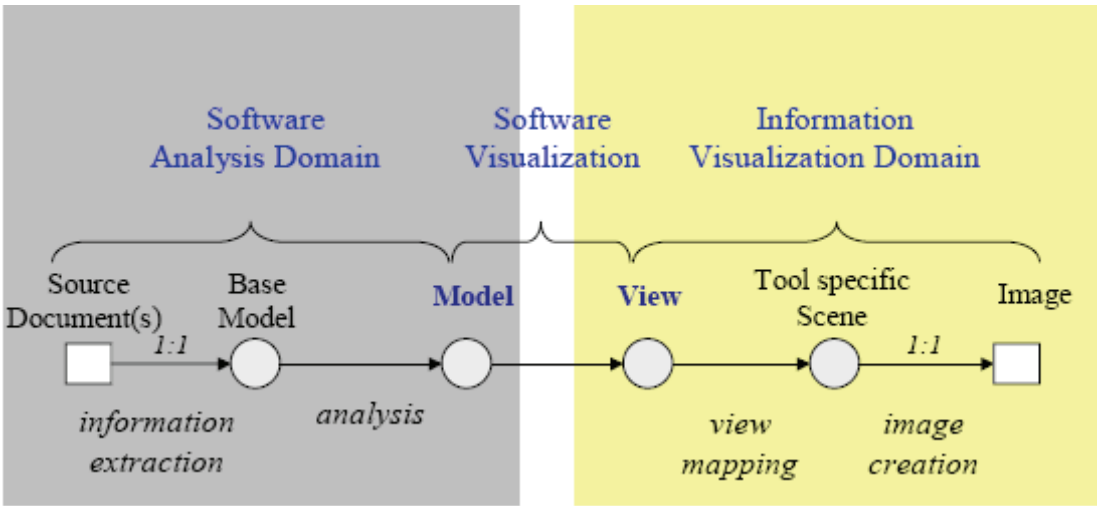

Abbildung 2-2: Ablauf der Analyse und Visualisierung [8]

3 Anforderungen

Dieses Kapitel befasst sich mit der Feststellung des Gegenstandes der Arbeit und definiert die Grundlagen, auf denen das Projekt beruht.

3.1 Problemanalyse

Wie schon in der Einleitung unter dem Punkt Problem erwähnt, verwendet der VizzAnalyzer™ zur Darstellung während der Laufzeit standardmäßig ein GUI. Für das GUI wurde das SWT benutzt. Dabei handelt es ich um eine Open Source-Lösung [3] welche von IBM entwickelt wurde und nun jedoch von der Eclipse Foundation gewartet und weiterentwickelt wird. Es stellt dabei eine Alternative zu anderen GUI Formen wie AWT und SWING dar. Das besondere an SWT ist, dass es so dargestellt werden kann, wie es der Benutzer von seinem Betriebssystem gewohnt ist. Ohne dabei Änderungen vornehmen zu müssen. Dies gelingt dadurch, das SWT auf die plattformspezifischen Funktionen und Methoden aus Java heraus zugreift. Dieses Verfahren wird „native“ genannt und bedeutet, dass die Funktion oder Methode aus der einheimischen Bibliothek stammt. [4]

Abbildung 3-1: SWT Darstellung unter verschiedenen Plattformen [3]

Diese grafischen Anwendungen sind jedoch nicht immer verfügbar, so z.B. in Serverumgebungen, die häufig ohne die für Desktop Computer typischen grafischen Benutzerschnittstellen betrieben werden. Gleichfalls gestaltet es sich schwierig, den VizzAnalyzer™ in einem Scriptmodus zu starten.

Der Scriptmodus bedeutet, dass das Programm automatisch, ohne Benutzereingabe, aufgezeichnete oder anderweitig festgelegte Befehlsfolgen abarbeitet und ein entsprechendes Ergebnis produziert. Dazu wird das Programm mit einem Parameter gestartet. Dieser Parameter könnte beispielsweise der Pfad zu einer Textdatei sein. Das Programm sollte dann in der Lage sein zu erkennen, dass es mit einem Startparameter aufgerufen wurde und dementsprechend handeln. In diesem Fall wäre es denkbar, dem Programm eine Datei mit Kommandosequenzen zu

übergeben. Diese würde geöffnet werden und anschließend Kommando für Kommando ausgeführt werden. So könnte die Effektivität gesteigert werden, da nicht alles per Hand ausgeführt werden müsste.

Einige Kommandos, die implementiert werden, benötigen Parameter, um deren Ausführung zu steuern. Um die Parameter aus der Nutzereingabe herauszufiltern, müssen diese jedoch zuvor analysiert werden. Das übernimmt der sogenannte Parser. Dabei wird die Eingabe aufgetrennt und nach Optionen durchsucht. Um diese Funktionalität zu ermöglichen, wurden verschiedene Parser-Projekte untersucht und bewertet. Vier Projekte wurden testhalber implementiert und nach verschiedenen Kritikpunkten verglichen. Getestete Kritikpunkte waren der Support, also wann wurde das Projekt zum letzten Mal aktualisiert. Die Einfachheit der Anwendung beschreibt wie viel Arbeit notwendig ist, um den Parser zu implementieren und wie kompliziert es ist, auf dessen Funktionalitäten zu zugreifen. Die Lizenz, unter der das Projekt veröffentlicht wurde, ist in sofern wichtig, weil der VaCli als kommerzielle Software gilt. Daher muss es möglich sein, das Projekt in einer solchen Anwendung zu verwenden und das am besten ohne Lizenzgebühren zu bezahlen. Das Kriterium Anwendung beschriebt, welche Besonderheiten bei den Tests auffielen. Der Parser sollte für Eingliederung, in das Kommandozeilen-Interface, am besten als eine Javaprogrammbibliothek zur Verführung stehen. Das Ergebnis der Untersuchung zeigt die Abbildung 3-2.

	Takeaway (CLI)	**args4j**	**JSAP**	**jargs**
Support	letztes Update August 14, 2008	letztes Update Juni 2008	letztes Update August 4, 2006	letztes Update April 12, 2005
Lizenz	Apache Lizenz für gewerbliche Nutzung freigegeben	MIT Lizenz für gewerbliche Nutzung freigegeben	LGPL für gewerbliche Nutzung freigegeben	BSD Lizenz für gewerbliche Nutzung freigegeben
Funktionalität	sehr gut	verwendet "Annotations"	ausreichend	unzureichend
Flexibilität	sehr gut; Eine Option kann den Status haben, ob ein Wert nach der Option steht.		gut; Eine Option kann den Status "benötigt" oder "optional" haben.	nicht gut; Eine Option kann nicht den Status "benötigt" oder "optional" haben; Eine Option kann den Status haben, ob ein Wert nach der Option steht.
Einfachheit				
Anwendbarkeit	einfach; Die Optionen können für jedes Kommando individuell gesetzt werden; Die Optionen können benutzt werden um einen Hilfetext zu generieren;	Funktioniert nur mit Optionen die beim Programmstart angegeben werden	einfach; Die Optionen können für jedes Kommando individuell gesetzt werden; Die Optionen können benutzt werden um einen Hilfetext zu generieren;	einfach; Die Optionen können für jedes Kommando individuell gesetzt werden; Es kann kein Hilfetext aus den Optionen erzeugt werden;
Bibliothek				
Urteil	Ausreichend Attribute für die Optionen stehen zur Verfügung; Aktives Projekt; Verfügt über die Abfrage, ob ein Wert nach der Option folgen muss; Verfügt über die Abfrage, ob eine Option in dem verarbeiteten Text vorkommt; Verfügt nicht über die Möglichkeit, einer Option den Status "benötigt" oder "optional" zu geben	Nicht weiter getestet, da nur mit Startparametern gearbeitet werden kann	Ausreichend Attribute für die Optionen stehen zur Verfügung; Die letzte Aktualisierung ist zu lange her --> möglicherweise kein Support mehr; Verfügt nicht über die Abfrage ob ein Wert nach der Option folgen muss; Verfügt nicht über die Abfrage ob eine Option in dem verarbeiteten Text vorkommt	Den Optionen stehen zu wenig Attribute zur Verfügung; Die letzte Aktualisierung ist zulange her --> möglicherweise kein Support mehr; Zu simpel in seiner Art und Weise

Abbildung 3-2: Parseranalyse

Das Projekt, welches alle Anforderungen am besten erfüllte, war das CLI des Apache Commons Projects [6]. Jedoch verlangt dieser einen schon vorverarbeiteten `String`, das bedeutet, die Nutzereingabe muss vorverarbeitet werden. Der zu implementierende Parser muss sich aus zwei Komponenten zusammensetzen: dem Zerlegen der Nutzereingabe und dem Durchsuchen der getrennten Eingabe. Das Durchsuchen übernimmt der CLI, das Zerlegen kann die Startklasse übernehmen. Diese benötigt den Kommandonamen aus der Eingabe für die Zuweisung an ein Kommando. Die Trennung müsste anhand eines eindeutigen Trennzeichens erfolgen. Die anderen nicht weiter erwähnten Projekte hatten zum Teil einen ganz anderen Lösungsansatz als der Apache CLI. So wurden Optionen anhand von „Annotations“ erkannt und der Parser arbeitete nur mit Optionen, die als Startparameter des Programms mit übergeben wurden. Des Weiteren waren die letzten Versionen der Parserprojekte bis zu drei Jahren alt. Der VizzAnalyzer™ wurde in der Programmiersprache Java entwickelt, deshalb werden alle Entwicklungen, die während dieser Arbeit gemacht werden, ebenfalls in Java programmiert.

3.2 Anwendungsfälle

Im folgenden Unterkapitel werden Aussagen über die verschiedenen UseCase-Szenarien der Kommandozeilenimplementierung des VizzAnalyzers™ gemacht. Ein UseCase oder Anwendungsfall genannt, beschreibt die Interaktion eines Akteurs mit einem System. Diese Fälle sollen in erster Linie einen groben Überblick über das System liefern, außerdem sind sie recht hilfreich, um die verschiedenen Nutzergruppen zu bestimmen. Die Darstellung der Anwendungsfälle erfolgt sowohl in textueller als auch grafischer Form. Bei der grafischen Umsetzung bedienten sich die Entwickler der Sprache UML – Unified Modeling Language. Die Abbildung 3-3 zeigt die grafische Umsetzung der Anwendungsfälle mit UML. Bei den Nutzern der Applikation handelt es sich um Entwickler von Javasoftware. Es kann sich dabei um professionelle als auch Hobbyentwickler handeln. Bei der textuellen Umsetzung werden die Anwendungsfälle näher erläutert. Die Erläuterung besteht aus einer Beschreibung, den beteiligten Akteuren, möglicherweise auftretenden Vor- und Nachbedingungen sowie einem normalen Ablauf und einem möglichen alternativen Ablauf.

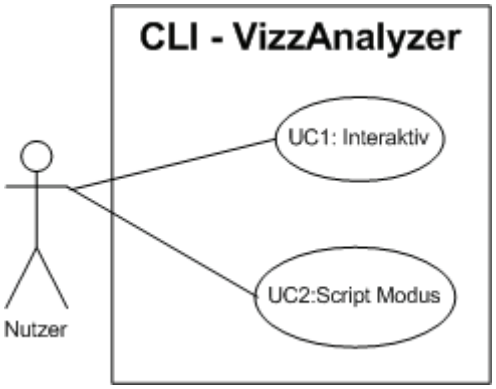

Abbildung 3-3: UseCase Diagramm

UseCase UC1 **Interaktiv**

Beschreibung: Nachdem das Programm gestartet wurde, wartet es auf Befehlseingaben durch den Nutzer. Dieser kann dabei aus der vollen Befehlsliste schöpfen. Diese Liste umfasst sowohl Befehle zur Analyse, als auch eine Reihe von Befehlen, die die Steuerung des Programmes ermöglichen, wie zum Beispiel Kommandos für die Anwendung von Befehlssequenzen, Verwalten von Graphen sowie Hilfe- und History-Anweisungen.

Anwender: Der Nutzer

Vorbedingung: Keine

Nachbedingung: Der VizzAnalyzer™ läuft in der Kommandozeilenebene und ist bereit für die Eingabe von Befehlen von der Nutzerseite.

Ablauf:

1. Nutzer startet das Programm
2. Programm lädt die benötigen Bibliotheken
3. Programmklassen, welche verfügbare Kommandos enthalten, werden vorgeladen.
4. Programm prüft, ob es mit einem Parameter gestartet wurde:
   a. Ja → weiter mit UC2
   b. Nein → weiter mit 5.
5. Öffnen der History-Datei für den aktuellen Tag
6. Auf Eingabe warten
7. Eingabe == „exit“-Befehl:
   a. Ja → Ende
   b. Nein → weiter mit 8.
8. Ausführen → weiter mit 6.

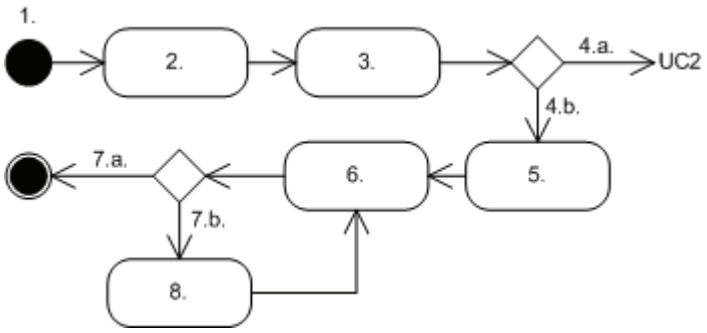

Abbildung 3-4: Ablaufdiagramm UC1

UseCase UC2 **Script Modus**

Beschreibung: Der Start des Programms erfolgt mit der Angabe eines Dateipfades. Diese Datei enthält eine Sequenz von Kommandos, dabei kann es sich auch um die vom Programm angelegten History-Dateien handeln. Die Applikation startet und beginnt die angegebene Datei zeilenweise einzulesen. Stößt das Programm dabei auf ein Kommando, wird es versuchen, dieses auszuführen. Gelingt die Ausführung und das Dateiende ist noch nicht erreicht so wird die nächste Zeile eingelesen. Misslingt die Ausführung oder ist das Dateiende erreicht, so wird die Applikation beendet.

Anwender: Der Nutzer

Vorbedingung: Keine

Nachbedingung: Der VizzAnalyzer™ wird nach Abarbeitung der Kommandosequenz beendet.

Ablauf:

1. Nutzer startet das Programm
2. Programm lädt die benötigen Bibliotheken
3. Programmklassen, welche verfügbare Kommandos enthalten, werden vorgeladen.
4. Programm prüft, ob es mit einem Parameter gestartet wurde:
   a. Ja → weiter mit 5.
   b. Nein → weiter mit UC1
5. Übergebene Datei exsistiert:
   a. Ja → weiter mit 6.
   b. Nein → Fehler ausgeben → Ende
6. Öffnen der Datei
7. Einlesen einer Zeile
8. Bestimmen des Typs der Zeile:
   a. Kommentar → weiter mit 10.
   b. Unixzeit → weiter mit 11.
   c. Kommando → weiter mit 9.
9. Ausführen
10. Ausgeben

11. Ende der Datei erreicht:

    a. Ja → Ende

    b. Nein → zurück zu 7.

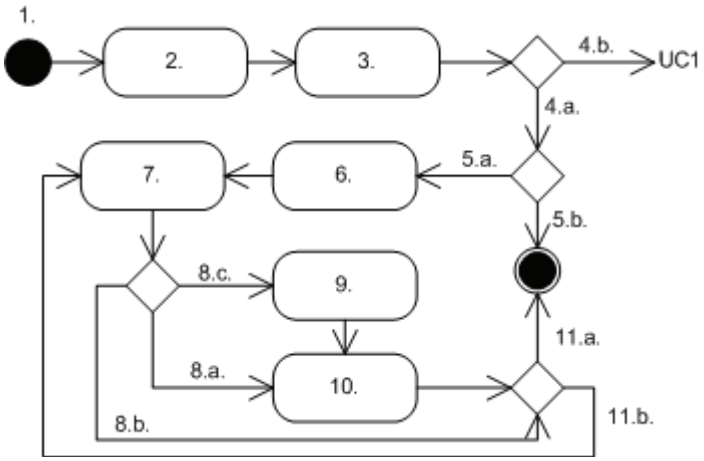

Abbildung 3-5: Ablaufdiagramm UC2

3.3 Anforderungen

Dieser Abschnitt befasst sich mit den funktionalen und nicht funktionalen Anforderungen an das zu entwickelnde Programm. Diese sagen aus, was mit dem Programm möglich sein muss und welche Richtlinien dabei eingehalten werden sollten.

Jeder funktionalen Anforderung sind, wie schon zuvor, eine eindeutige Referenznummer, ein Name und ein oder mehrere UseCases zugeordnet. Zudem bestehen sie aus einer Beschreibung, Begründung und einem Erfüllungskriterium, ob die Anforderung erfüllt ist.

Funktionale Anforderungen

Anforderung A01	
	Kommandozeilenprogramm
	UseCase#: UC1, UC2
Beschreibung:	Das Programm soll in einer Konsole laufen. Alle Aus- und Eingaben müssen in dieser besagten Konsole geschehen und dürfen nur textbasiert sein.
Begründung:	Die Verwendung von einem GUI, wie es aktuell besteht, ist nicht gewünscht, da es nicht auf allen möglichen Plattformen darstellbar ist. Des Weiteren ist die Ausführung des VizzAnalyzers™ in seiner jetzigen Form nicht in einem Scriptmodus möglich.
Erfüllungskriterium:	Die Anforderung gilt als erfüllt, wenn die Applikation keinerlei grafische Elemente verwendet wie sie in einem GUI üblich sind. Also wenn auch ein Einsatz auf einem Server über ein Terminal möglich ist.

Anforderung A02	
	Vordefinierte Kommandos
	UseCase#: UC1, UC2
Beschreibung:	Dabei handelt es sich um Befehle, die für die grundlegenden Funktionen des Systems verantwortlich sind. Diese Befehle sind unabhängig von den späteren Analysekommandos.
Begründung:	Das Vorhandensein von solchen Kommandos ist wichtig, da die Analyse einige Daten benötigt, um überhaupt einwandfrei

	arbeiten zu können. Des Weiteren sind diese Kommandos wichtig für die Benutzerfreundlichkeit des Programms und dessen Steuerung.
Erfüllungskriterium:	Sind alle untergeordneten Anforderungen erfüllt, gilt diese Anforderung auch als erfüllt.

Anforderung A02-A	
	Einen Graphen laden (loadgraph)
	UseCase#: UC1, UC2
Beschreibung:	Beim Ausführen dieses Befehls in Kombination mit einer Datei, die einen Graphen enthält, wird dieser in den Arbeitsspeicher geladen und mit einer internen Zuordnung versehen.
Begründung:	Eines der wichtigsten Elemente während der Analyse ist der Graph. Er bildet die Grundlage und ist das, was überhaupt auf seine Qualität analysiert werden soll. Dazu muss er jedoch vorher in den Arbeitsspeicher geladen werden. Die Vergabe einer internen Zuordnung soll dem Nutzer den Umgang mit dem System vereinfachen, da dadurch nicht mehr der komplette Name des Graphen verwendet werden muss. Der Name kann unter Umständen sehr lang werden. Die interne Zuordnung ist eine eindeutige Kennzeichnung des Graphen durch eine Zahl.
Erfüllungskriterium:	Die Anforderung wird als erfüllt angesehen, wenn nach dem Ausführen der „loadgraph“-Anweisung eine GML-Datei, welche die Struktur eines Graphen enthält, ohne Fehler von der Festplatte in den Arbeitsspeicher geladen wurde und danach weitere Operationen über dem Graphen ausgeführt korrekt ausgeführt werden können. Des Weiteren lässt sich am Speicher bedarf der Applikation feststellen, ob ein Graph tatsächlich in den Arbeitsspeicher geladen wurde.

Anforderung A02-B	
	Einen neuen Graph speichern (saveas)
	UseCase#: UC1, UC2
Beschreibung:	Speichert die Struktur eines generierten Graphen, der sich im Arbeitsspeicher befindet in eine GML-Datei.
Begründung:	Bei manchen Analysefunktionen werden während der Ausführung neue Graphen erstellt. Diese befinden sich dann im Arbeitsspeicher. Sollen sie auch nach Beendigung des Programms weiterhin zur Verfügung stehen, ist es notwendig die Graphen zu speichern.
Erfüllungskriterium:	Die korrekte Funktionsweise des Kommandos wird dadurch bestimmt, dass ein generierter Graph vom Arbeitsspeicher auf die Festplatte gespeichert wird. Existiert nach dem Ausführen des Kommandos „saveas“ eine GML-Datei, deren Größe und deren Inhalt ungleich null ist, so kann das Kommando als korrekt implementiert und funktionsfähig angesehen werden.

Anforderung A02-C	
	Einen Graph löschen (unloadgraph)
	UseCase#: UC1, UC2
Beschreibung:	Entfernt einen bereits im Arbeitsspeicher befindlichen Graphen aus besagtem Speicher.
Begründung:	Sollten mehrere Graphen geladen worden sein, besteht durchaus die Möglichkeit, dass der Arbeitsspeicher irgendwann nicht mehr ausreichend ist, um weitere Graphen zuladen oder durch Analysen zu erstellen. Das Entfernen von Graphen aus dem Arbeitsspeicher gibt wieder Speicherplatz frei.
Erfüllungskriterium:	Ein Graph kann als gelöscht angesehen werden, wenn der von ihm belegte Speicher wieder freigeben ist und eine Verwendung seiner internen Zuordnungen nicht mehr möglich ist. Ebenso ist zu erwarten, das der Graph bei einer Auflistung aller geladenen Graphen nicht mehr auftaucht.

Anforderung A02-D	
	Zeige alle Graphen (showgraphs)
	UseCase#: UC1, UC2
Beschreibung:	Listet alle sich im Arbeitsspeicher befindenden Graphen mit ihren vollen Namen und ihrer internen Zuordnung, auf.
Begründung:	Dieser Befehl soll verhindern, dass der Speicher ineffektiv genutzt wird, das heißt, hierdurch sollen doppelte Ladungen von Graphen verhindert werden. Des Weiteren benötigen alle Analysen eindeutige Graphenbezeichnungen. Diese kann man aus der Auflistung entnehmen. Jedem Graphen ist dort ein eindeutiger kurzer Ausdruck in Form einer Nummer zugeordnet.
Erfüllungskriterium:	Die Anforderung steht in einem engen Zusammenhang mit der korrekten Funktionsweise von A02-A und A02-C. Die Liste, die ausgegeben wird, sollte sich nach der Ausführung der Anweisungen A02-A verlängern und nach A02-C verkürzen. Ist das der Fall, ist die Anforderung erfüllt.

Anforderung A02-E	
	Befehlssequenz ausführen (openmacro)
	UseCase#: UC1, UC2
Beschreibung:	Öffnet eine Textdatei und führt die darin enthaltenen Kommandos der Reihe nach aus. Dabei soll es auch möglich sein vom Programm erstellte Verläufe (History) auszuführen.
Begründung:	Oft ist es der Fall, das eine Sequenz von Befehlen auf mehreren Graphen angewendet werden soll. Dazu müssten alle Kommandos wiederholt per Hand eingegeben werden. Um dies zu verhindern und somit Zeit zu sparen, können die in einer Datei gespeicherten Befehle ausgeführt werden.

Erfüllungskriterium:	Die Ausführung eines fest definierten Inhaltes einer Datei mit Befehlssequenz sollte immer als Ergebnis haben, dass der Inhalt der gelesenen Zeile an den Nutzer ausgegeben wird. Danach sollte die Bearbeitung folgen, welche eine Ergebnisausgabe liefern sollte. Eine korrekte Implementation zeichnet sich dadurch aus, dass das Makro nicht vorzeitig durch den „openmacro“-Befehl abgebrochen wird. Zeitstempel im UNIX-Format werden dabei ignoriert und nicht mit ausgegeben.

Anforderung A02-F	
	Befehlssequenz speichern (recordmacro)
	UseCase#: UC1
Beschreibung:	Startet mit einem entsprechenden Parameter die Aufzeichnung von einer Reihe von Befehlen. Diese werden dann im korrekten Format in einer Textdatei abgespeichert. Die Aufzeichnung wird beim Aufruf des Kommandos mit einem Stoppparameter wieder angehalten.
Begründung:	Das zuvor schon erwähnte Ausführen von Befehlssequenzen aus einer Datei setzt voraus, dass diese im richtigen Format vorliegen. Das Kommando sorgt dafür, dass während der Laufzeit solche Sequenzen im korrekten Format gespeichert werden können. Der Unterschied zur Verlaufsspeicherung ist, dass in dieser Aufzeichnung nur bestimmte Sequenzen enthalten sind. Der Nutzer bestimmt, welche Sequenz in der Datei gespeichert werden soll, indem er die Kommandos der Reihe nach durchführt.
Erfüllungskriterium:	Die Befehlssequenzen werden in einer Datei gespeichert, das heißt, eine korrekte Ausführung des Kommandos ist zum Einen daran zu erkennen, das eine Datei auf der Festplatte in dem angegebenen Verzeichnis angelegt wurde. Absolut korrekt ist die Funktionsweise, wenn auch der Inhalt der Datei mit dem was zur Laufzeit im Programm eingegeben wurde übereinstimmt.

Anforderung A02-G	
	Befehlshistory darstellen (history)
	UseCase#: UC1
Beschreibung:	Während der Programmlaufzeit speichert ein Hintergrundprozess alle eingegebenen Kommandos in einer Datei. Es soll dabei leicht nachzuvollziehen sein, an welchem Tag ein Kommando verwendet wurde. Darum legt das Programm für jeden Tag eine eigene Datei an und speichert jedes verwendete Kommando zusätzlich mit einem Zeitstempel im UNIX-Format. Dabei nimmt dieser Prozess jedoch keine Rücksicht auf eventuelle Fehler in der Syntax der Kommandos. Der „history“-Befehl kann diese Dateien öffnen und darstellen. Hierzu gibt es verschiedene Möglichkeiten und Parameter. Der Befehl kann ohne Parameter ausgeführt werden, dann wird der Verlauf vom aktuellen Tag dargestellt. Es besteht aber auch die Möglichkeit, dass ein Nutzer den Verlauf eines anderen Tages betrachten möchte. In dem Fall kann mithilfe eines Parameters der Tag bestimmt werden, dessen Verlauf dargestellt werden soll. Des Weiteren sollte es auch die Möglichkeit geben bestimmte Elemente des Verlaufs zu unterdrücken, also diese nicht darzustellen. Mit einem weiteren Parameter lässt sich der Verlauf ab einer bestimmten Uhrzeit darstellen.
Begründung:	Die Speicherung des Verlaufs soll in erster Linie die Rückverfolgung von Analysen ermöglichen. Die Rückverfolgung dient der Erkennung von Fehlern, die bei Analysen aufgetreten sein können.
Erfüllungskriterium:	Die Funktionsweise für das Speichern des Verlaufs gilt als korrekt, wenn eine Datei in einem Verzeichnis angelegt wird und diese Datei einen Zeitstempel im UNIX-Format und das im Programm eingegebene Kommando enthält. Die Funktionalität der Darstellung ist korrekt, wenn die Darstellung der Datei, die den Verlauf enthält, mit dem Inhalt der Datei übereinstimmt. Die Zeitstempel sind von der Darstellung ausgenommen, weil diese lediglich der internen Verwendung dienen.

Anforderung A02-H	
	Hilfe anzeigen (help)
	UseCase#: UC1
Beschreibung:	Bei der Ausführung des Befehls stellt das System zwei unterschiedliche Arten von Hilfetexten zur Verfügung. Bei der einfachen Verwendung, also ohne zusätzlich Angaben, wird eine Liste aller verfügbaren Kommandos generiert. Zu jedem der dort aufgeführten Kommandos werden eine kurze Beschreibung und Informationen über den Gebrauch ausgegeben. In Kombination mit einem Befehl wird eine erweiterte Hilfe zum entsprechenden Befehl dargestellt.
Begründung:	Sollte ein Nutzer nicht mit dem System zurechtkommen, besteht so immer die Möglichkeit, sich über die notwendige Handhabung entsprechend zu informieren, auch wenn das Handbuch gerade nicht zur Verfügung steht.
Erfüllungskriterium:	Die Funktion des „help“-Befehls, in seiner simplen Form, gilt als korrekt, wenn eine Liste aller verfügbaren Kommandos mit einer Benutzungshilfe oder eine entsprechende Mitteilung, dass keine Hilfe verfügbar sei, auf dem Bildschirm ausgegeben wird und die Anzahl der dort aufgelisteten Kommandos der Anzahl der zuvor geladenen Kommandos entspricht. Die Richtigkeit der Kombination des „help“-Befehls mit einem Kommandonamen ist dann gegeben, wenn zu jedem Kommando ein Hilfetext oder eine Mitteilung, dass keine Hilfe verfügbar sei, ausgegeben wird.

Anforderung A02-I	
	Beenden des Programms (exit)
	UseCase#: UC1
Beschreibung:	Dieses Kommando beendet die Applikation. Dabei wird keine Rücksicht auf geladene Graphen genommen.
Begründung:	Der „exit“-Befehl wird benötigt, um dem Nutzer ein ordnungsgemäßes Beenden des Programmes zu ermöglichen.
Erfüllungskriterium:	Das Kommando arbeitet korrekt, wenn die Applikation beendet wird und aller zuvor belegter Speicher freigegeben wurde.

Anforderung A02-J	
	Einen bestehenden Graph speichern (save)
	UseCase#: UC1, UC2
Beschreibung:	Während der Ausführung von Analysen kann es vorkommen, dass geladene Graphen verändert werden. Bei der Speicherung werden die Graphen automatisch mit der Datei, aus der sie geladen wurden, assoziiert.
Begründung:	Die Anforderung erleichtert dem Nutzer die Arbeit mit dem System.
Erfüllungskriterium:	Die Anforderung gilt als erfüllt, wenn ein zuvor geladener Graph nach der Speicherung mit dem „save“-Befehl in aktualisierter Form in der Datei vorliegt, das heißt, wenn der Inhalt der Datei überschrieben wurde.

Anforderung A03	
	Lesen von Tastatureingaben
	UseCase#: UC1
Beschreibung:	Eingaben des Nutzers müssen in das System gelangen können.
Begründung:	Die Arbeitsweise der Applikation beruht auf der Eingabe von Befehlen durch die Tastatur, da es sich hierbei um ein Kommandozeilenprogramm handelt (A01). Das bedeutet, nach der Abarbeitung eines Kommandos muss das System wieder in der Lage sein, einen neuen Befehl von der Tastatur zu lesen. Dabei handelt es sich um einen endlosen Kreislauf, der nur von einem Programmabsturz, einem harten Beenden des Programms oder durch den „exit“-Befehl (A02-I) unterbrochen werden kann. Das Einlesen von Befehl und Parameter erfolgt in einer Zeile, sodass es dadurch sowohl für den Nutzer, als auch für den Entwickler einfacher wird. Der Nutzer behält den Überblick über seine Eingaben und der Entwickler kann alle benötigten Daten auf einmal erfassen, ohne wieder auf eine Eingabe durch den Nutzer warten zu müssen. Auch gehört es zur gewohnten Arbeitsweise mit derartigen Programmen.
Erfüllungskriterium:	Diese Anforderung steht in einem sehr engen Zusammenhang mit allen ausführbaren Kommandos. Daher ist die Erfüllung dieser Anforderung nur mit einem Test eines Kommandos bestimmbar. Dabei sollte es sich um ein Kommando mit Optionen handeln, da daran erkennbar ist, ob die Trennung der Eingabe korrekt funktioniert. Sollte dieses Kommando dann in seiner Ausführung korrekt sein, so ist davon auszugehen, dass die Anforderung A03 korrekt funktioniert und somit erfüllt ist. Als Beispielkommando würde sich der „help“-Befehl anbieten. Da erstens zuvor beschrieben wurde, wann dieses Kommando als korrekt gilt und zweitens handelt es sich dabei um eine Grundfunktion, die immer zur Verfügung steht.

Anforderung A04	
	Script-Modus
	UseCase#: UC2
Beschreibung:	Hierbei handelt es sich um eine Funktion der Applikation, die in einem engen Zusammenhang mit der Anforderung A02-E steht. Die Anforderung ermöglicht dem Nutzer, dem Programm einen Startparameter mit zu übergeben. Bei dem Parameter handelt es sich um eine Datei, die eine Befehlssequenz enthält, welche vom Programm ausgeführt wird.
Begründung:	Diese Funktion erleichtert dem Nutzer den Umgang mit dem Programm. Wenn der Nutzer beispielsweise zuvor eine Sequenz erstellt hat und diese zu einem späteren Zeitpunkt nochmals ausführen möchte, so kann die Applikation mit der Datei als Parameter gestartet werden. Das Programm führt die enthaltenen Anweisungen aus und beendet sich dann selbstständig. Der Nutzer muss nicht mehr das Programm starten und ein Makro manuell zur Ausführung bringen. Die Anforderung ermöglicht es so, die Anwendung durch einen Zeitplaner, z.B. „crontab“ unter Linux, dann ausführen zulassen, wenn die Belastung des Rechners am geringsten ist. Die Applikation kann so auch in einen automatisierten Prozess integriert werden.
Erfüllungskriterium:	Die Erfüllung der Anforderung kann dadurch getestet werden, dass eine Makro-Datei geschrieben wird. Das kann mit dem „recordmacro“-Befehl erfolgen oder aber manuell. Wichtig ist nur, dass das richtige Format eingehalten wird. Die erstellte Datei wird als Parameter mit dem Startbefehl der Applikation verbunden. Die Applikation wird erkennen, dass sie mit einem Parameter gestartet wurde. Die übergebene Datei wird ausgeführt werden wie bei Anforderung A02-E. Daher ist diese Anforderung sehr eng mit A02-E verbunden. Ist A02-E funktionsfähig so ist es zu einem großen Teil auch diese Anforderung. Nach der Ausführung wird die Applikation beendet.

Anforderung A05	
	Ergebnisse darstellen
	UseCase#: UC1, UC2
Beschreibung:	Die Applikation soll ihren Benutzer immer darüber informieren, was das Ergebnis der durchgeführten Aktion war. Dies ist sowohl bei einem erfolgreich ausgeführten Befehl, als auch bei einem fehlgeschlagenen der Fall. Des Weiteren sollte der Nutzer über den Fortschritt seiner Analyse unterrichtet werden. Ebenso muss es möglich sein, Analysen abzubrechen.
Begründung:	Die Darstellung der Ergebnisse ist gerade für den Nutzer von größter Bedeutung. Wenn dieser nicht darüber informiert wird, welche Handlungen gerade im System ablaufen, ist es ihm nicht möglich, darauf zu reagieren. Der Nutzer kann ohne eine Fehlermeldung beispielsweise auch seine Fehler nicht korrigieren.
Erfüllungskriterium:	Da jedes Kommando eine Ausgabe liefert, muss nur jedes Kommando ausgeführt werden, um die Funktionsweise zu testen. Da es jedoch mehrere Möglichkeiten der Fertigstellung gibt, muss auch jedes Kommando mehrfach ausgeführt werden. Dazu gehört, dass ein Kommando erfolgreich und fehlerhaft beendet wird. Mit Hilfe von Test Fällen (Test Cases) und JUnit ist es möglich, vordefinierte Eingaben für jedes Kommando zu erstellen und diese automatisiert hintereinander ausführen zu lassen. So kann festgestellt werden, ob alle Möglichkeiten der Ausführung auch Ausgaben liefern.

Anforderung A06	
	Benutzen der VizzAnalyzer™ Bibliotheken
	UseCase#: UC1, UC2
Beschreibung:	Für jeden Befehl, der eine Analyse ausführt, muss ein entsprechendes Objekt erzeugt werden, welches sich auf den Algorithmus der Analyse bezieht. Das Kommandozeilen-Interface muss dazu mit der API der Bibliothek des VizzAnalyzer™ kommunizieren. Es wird eine Konfiguration vorgenommen, die an den eigentlichen Quellcode der Analyse weitergeleitet wird.
Begründung:	Die Applikation ist lediglich die Schnittstelle zwischen Nutzer und der API der Bibliothek. Diese Schnittstelle soll nicht nur feststellen, was der Nutzer möchte, sondern dann die Aufgabe auch ausführen. Um das tun zu können, müssen die Nutzereingaben korrekt analysiert und an die schon bestehenden Algorithmen der VizzAnalyzer™ Bibliothek weitergeleitet werden.
Erfüllungskriterium:	Das Interface benötigt die Algorithmen der VizzAnalyzer™ Bibliothek, daher können keine Analysen ausgeführt werden ohne das diese zur Verfügung stehen. Liefert die Ausführung einer Analyse bei der Verwendung korrekter Daten ein logisches Ergebnis, so ist die Verbindung hergestellt und die Anforderung erfüllt.

Erweiterbarkeit

Die Erweiterbarkeit spielt schon wie zuvor erwähnt bei Softwaresystemen eine wichtige Rolle. Sie kann durchaus die Lebensdauer einer Applikation verlängern. Ein wichtiges Kriterium dabei ist, dass dies möglichst einfach geschehen kann und gut dokumentiert ist. Bei späteren Entwicklungen kann es vorkommen, dass der ursprüngliche Entwickler nicht derjenige ist, der die Erweiterungen vornimmt. Daher muss es bei dieser Anwendung möglich sein, neu entwickelte Funktionen schnell und ohne großen Aufwand hinzuzufügen.

Zuverlässigkeit

Das System soll in jedem Fall immer so arbeiten, wie es erwartet wird. Das bedeutet, falls ein kritischer Fehler auftreten sollte, darf das Programm unter keinen Umständen abstürzen. Sollte es einen Fehler geben, muss das Programm diesen abfangen und eine entsprechende Bemerkung ausgeben, die dem Nutzer eine Hilfe bei der Lösung des Problems ist. Nach Abfangen des Fehlers und der Ausgabe eines Hilfetextes soll das Programm wieder in den Zustand zurückkehren, in dem es auf einen neuen Befehl wartet.

4 Architektur und Design

Dieses Kapitel befasst sich mit der Architektur und dem Design des Kommandozeileninterfaces des VizzAnalyzer™. Zuerst wird das Projekt als Ganzes betrachtet, im weiteren Verlauf werden die Hauptaspekte des Systems dargelegt und anschließend veranschaulicht.

Im Wesentlichen gibt es bei dem Projekt, den VizzAnalyzer™ auf der Kommandozeilenebene verfügbar zu machen, drei Hauptkomponenten, die miteinander agieren und kommunizieren müssen. Das eigentliche Problem ist dabei, dass die grafische Schnittstelle zwischen dem Nutzer und der VizzAnalyzer API nicht mehr verwendet werden soll. Diese Schnittstelle war das zuvor schon erwähnte GUI. Der Nutzer, also in diesem Falle der Mensch, ist nicht in der Lage, direkt mit der VizzAnalyzer API zu kommunizieren. Es wird eine Schnittstelle benötigt mit deren Hilfe der Mensch in der Lage ist mit der API zusprechen. Diese Aufgabe soll jetzt das Kommandozeilenprogramm übernehmen.

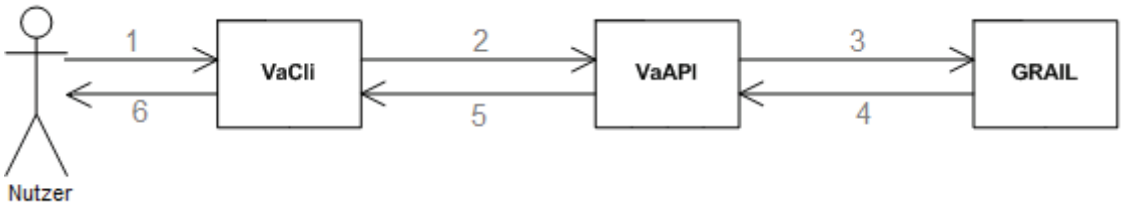

Abbildung 4-1: High Level Architektur

In der Abbildung 4-1 sind alle Komponenten dargestellt, die bei der Umsetzung des Projektes beteiligt sind. Die VaAPI ist die API, über die das Kommandozeilen-Interface mit der GRAIL-Bibliothek kommuniziert. Eine API ist also das Interface für die Kommunikation von Programmen. Die GRAIL-Bibliothek ist interne Grafik-Bibliothek, die an der Växjö-Universität entwickelt wurde. Sie ermöglicht es, dass über den Eigenschaften eines Graphen, z.B. Knoten und Kanten, Analysen ausgeführt werden können [7]. Sie verwaltet die Graphen und stellt verschiedene Algorithmen und Filter zur Verfügung, die an einem Graphen angewendet werden können [8]. Die nummerierten Pfeile der Abbildung 4-1 signalisieren den Ablauf einer normalen Anfrage an das System.

Im Ereignis 1 tippt der Nutzer eine Zeichenkette unter Verwendung seiner Tastatur in das laufende Kommandozeilenprogramm ein. Diese Zeichenkette wird vom VaCli entgegengenommen und analysiert. Diese Analyse erfolgt unter bestimmten Gesichtspunkten. Im ersten Schritt muss mithilfe der Analyse lediglich bestimmt

werden, ob es sich bei dem vom Nutzer eingegebenen Kommando um einen Teil des vom Programm zur Verfügung gestellten Kommandopools handelt. Wenn dem so ist, wird das bereits analysierte Kommando an die entsprechende Klasse weitergeleitet, welche für die Ausführung des Befehls zuständig ist. Die ausführende Klasse steht in Verbindung mit der VaAPI. Das führt im Ablauf der Abbildung 4-1 zu Ereignis 2. Aus dem analysierten Kommando extrahiert sich die Kommandoklasse des VaCli die Parameter, die die VaAPI benötigt um korrekt zu arbeiten. Mit diesen werden die entsprechenden Methoden der VaAPI konfiguriert. Wird VaAPI mit korrekten Daten „gefüttert“, so nimmt dieser Kontakt mit der GRAIL-Bibilothek auf.

Im Ereignis 3 kommunizieren also VaAPI und die GRAIL, es erfolgt eine Weiterleitung der Anfrage des Nutzers über korrekt konfigurierte Methoden der VaAPI. Die GRAIL führt die vom Nutzer verlangte Aktion aus und sendet die Ergebnisse, sei es in textueller oder anderer Form, zurück an die API.

Einer der wichtigsten Aspekte des Systems ist das Einlesen und das Verarbeiten/Analysieren der vom Nutzer eingegebenen Kommandos. Sowohl bei den Anforderung A02-G und A04 als auch im normalen nutzergesteuerten Ablauf wird auf dieses eigentliche Grundkonzept zurückgegriffen. Im Groben war der Ablauf schon in Abbildung 4-1 dargestellt und im Ereignis 1 und Ereignis 2 beschrieben, nun wird dies mit Abbildung 4-2 noch detaillierter betrachtet. Die erste Aktion, die ausgeführt wird, ist die Eingabe des Kommandos durch den Nutzer. Die Main-Methode des VaCli wartet so lange, bis der Nutzer seine Eingabe mit einem Druck der Enter-Taste abschließt. Gelesen wird in diesem Fall mithilfe eines Eingabepuffers, jener ermöglicht es von der Kommandokonsole gleich die komplette Zeile einzulesen. Durch die Festlegung, dass alle korrekten Kommandos der Applikation ein bestimmtes Format haben, wird die weitere Verarbeitung erleichtert. Dieses Format besagt, dass alle korrekten Kommandos immer mit dem Namen des Kommandos beginnen. Der Name wurde in der jeweiligen Klasse des Befehls definiert. Danach folgen die Parameter und Optionen, die zu diesem Kommando gehören. Getrennt wird alles von einem Leerzeichen zwischen den jeweiligen Spezifikationen, siehe Beispiel 4-1.

Beispiel 4-1: Allgemeines Kommando

```
Kommandoname -opt1 optValue1 --option2 optionValue2
```

Die Anzahl der Optionen für jedes Kommando zur Verfügung stehen, sind kommandospezifisch.

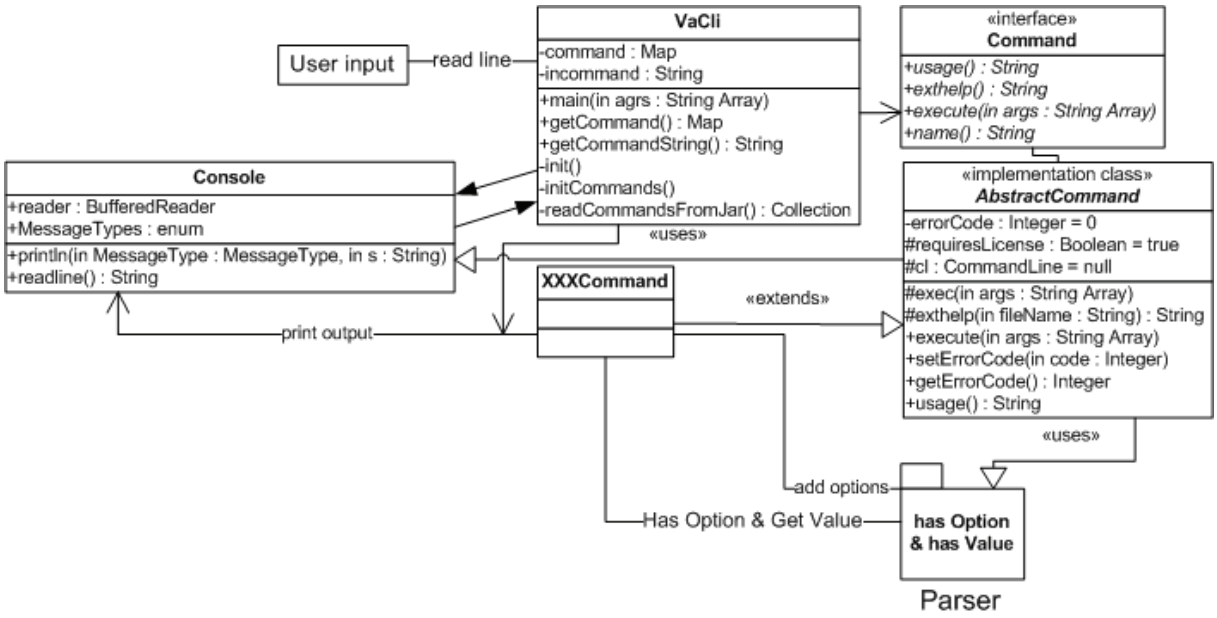

Abbildung 4-2: Kommando lesen & analysieren

Anhand des eindeutigen Trennungszeichens ist es jetzt möglich, den eingelesenen String aufzuteilen. Das erste Teilstück bezeichnet demzufolge immer das Kommando, welches der Nutzer ausführen möchte. Beim Start des Programms wurden alle verfügbaren Kommandos bestimmt und in einer Map, mit einem Indexschlüssel und einem dazugehörigen Wert gespeichert. Nach der Auftrennung des Nutzerstrings wird in der Map nachgesehen, ob ein Indexschlüssel existiert, der mit dem ersten Teilstück der Nutzereingabe identisch ist. Ist dies der Fall, geht die Main-Methode des VaCli davon aus, dass die besagte Nutzereingabe richtig ist. Die Nutzereingabe kann als richtig angesehen werden, weil die Main-Methode die Optionen und Parameter des Kommandos nicht kennt und auch nicht kennen muss, weil die weitere Fehlerbehandlung von den Klassen der Kommandos selbst durchgeführt wird. Die geteilte Eingabe wird an das entsprechende Kommando weitergeleitet. Dazu wird die Schnittstelle/Interface, Command, der Kommandoklassen angesteuert. Das Interface liefert Methoden, die für das Ansprechen jedes Kommandos gültig sind, gefolgt von einer abstrakten Klasse - dem AbstractCommand. Diese soll zukünftigen Kommandoklassen ihren Grundaufbau definieren. In dieser Klasse wird alles festgelegt, was unbedingt benötigt wird, so zum Beispiel einige Methoden, Variablen und einige Objekte die für das Analysieren der Nutzereingabe wichtig sind. Das Einlesen, Bestimmen was der Nutzer möchte

und das Weiterleiten ist somit abgehandelt. Im Folgenden bleibt jetzt noch für jede einzelne Kommandoklasse übrig, seine benötigen Parameter aus der Nutzereingabe zu filtern. Dies übernimmt ein sogenannter Parser, bei dem VaCli wird dafür die Apache Commons CLI Bibliothek verwendet. Einzelheiten zur Verwendung dieser Bibliothek erfolgen im Kapitel 5. Alle Basisoptionen, die der VaCli kennen muss, werden ebenfalls in der abstrakten Klasse der Kommandos festgelegt, damit die Optionen allen zur Verfügung stehen. Optionen, die für ein Kommando spezifisch sind, werden in der jeweiligen Klasse des Kommandos hinzugefügt. Die nach dem Leerzeichen aufgetrennte Eingabe des Nutzers wird dem Parser übergeben, dieser analysiert sie nun anhand der definierten Optionen.

Die eigentliche Klasse des Kommandos braucht im weiteren Verlauf dann nur noch zwei Arten von Anfragen an den Parser stellen. Zum einen ist das „Hat die Nutzereingabe die geforderte Option?“ und zum anderen „Gib mir den Wert, der zur geforderten Option gehört“. Mithilfe dieser Anfragen bestimmt die Kommandoklasse, ob sie mit den minimal geforderten Optionen aufgerufen wurde. Ist das der Fall so definiert die Klasse die Konfiguration für die VaAPI und schickt diese weiter.

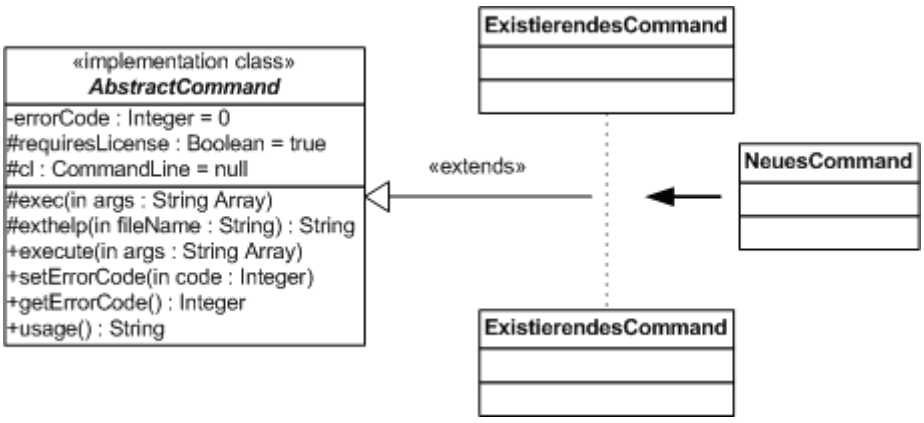

Abbildung 4-3: Kommando hinzufügen

Die Erweiterbarkeit der Applikation zeichnet sich dadurch aus, dass mit nur wenigen Änderungen neue Kommandos implementiert werden können. Die Abbildung 4-3 zeigt veranschaulicht, dass ein neues Kommando lediglich in die Liste der alten Kommandos eingehängt werden muss. Dafür muss die neue Kommandoklasse die bestehenden Funktionen aus der Klasse „AbstractCommand“ erben. Wie schon zuvor kurz erwähnt, werden alle in der Anwendung verfügbaren Kommandos beim Start des Programms vorgeladen und in einer Map gespeichert. Das vorladen geschieht automatisch. Der Entwickler muss an dieser Stelle nicht in den Quellcode eingreifen, um dort Veränderungen vorzunehmen. Das Hinzufügen von neuen Kommandos ist durch die Klasse „AbstractCommand“ ebenfalls erleichtert. Die

Klasse legt die minimalen Anforderungen fest, die ein Kommando erfüllen muss, damit es mit der Applikation korrekt arbeiten kann. Implementiert der Entwickler nun ein neues Kommando, so muss er lediglich die Spezifikationen für seine neue Entwicklung ergänzen. Sollte über die Vorgabe der abstrakten Klasse noch mehr für die erfolgreiche Abarbeitung benötigt werden, so kann es in der Klasse des neuen Kommandos hinzugefügt werden. Da die Optionen die der Parser verarbeiten kann, in der Klasse des Kommandos festgelegt werden, sind sie kommandospezifisch. Das bedeutet, dass die Optionen mit einer auf das Kommando angepassten Beschreibung versehen werden können. Zugleich kennt der Parser nur die zuvor registrierten Optionen, somit kann eine Benutzungshilfe für das jeweilige Kommando individuell erzeugt werden.

Die Ausgabe, die der VaCli macht, spielte schon bei dem Festlegen der Anforderungen in Kapitel 3 eine Rolle. Es ist wichtig, dass alle Ausgaben ein einheitliches Aussehen aufweisen. Das ist auch mit dem Gedanken verbunden, dass Änderungen nicht am Inhalt der Ausgaben, sondern am Aussehen, so einfach wie möglich vollzogen werden sollten. Das bedeutet, dass Änderungen nicht an jeder Stelle, wo es eine Ausgabe gibt, vorgenommen werden, sondern am besten zentral definiert werden sollten. Diese Aufgabe übernimmt die Klasse Console des VaCli. In der Klasse sind verschiedene Methoden implementiert, die eine einheitliche Ausgabe ermöglichen. Wenn eine andere Klasse eine Ausgabe vornehmen möchte, so ruft die Klasse dafür die entsprechende Methode der Klasse „Console“ auf und übergibt die geforderten Parameter. Die Methode übernimmt, wie schon geschrieben, die Formatierung und auch die eigentliche Ausgabe für den Nutzer. Unter anderem wird eine Instanz der Klasse „Console“ im AbstractCommand erzeugt, das heißt das alle Ausgaben, der einzelnen Kommandos, an die Klasse „Console“ weitergereicht werden müssen, um die optische Einheit zubewahren.

5 Implementierung

Dieses Kapitel beschreibt die Implementierungen, die während der Entwicklung des Kommandozeilen-Interfaces gemacht wurden. Das Kapitel geht dabei auf wichtige Komponenten, wie das Einlesen und Zuweisen, die Analyse der Eingabe mit dem Parser und die zentrale Ausgabensteuerung, ein.

5.1 Einlesen & Zuweisen des Kommandos

Das Analysieren der Nutzereingabe geschieht im Groben in zwei Schritten, zum Einen ist das die Zerlegung des Strings mit Hilfe des Leerzeichens als Trennzeichen. Zum Anderen ist das die Analyse der getrennten Eingabe auf angegebene und geforderte Optionen und deren Werte.

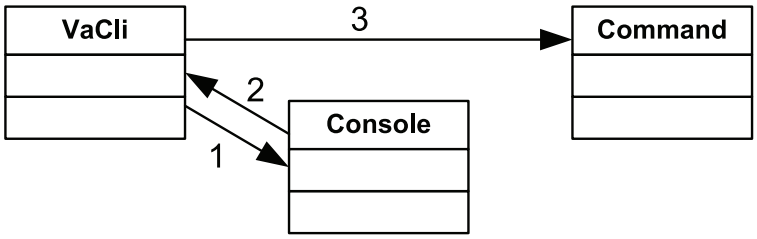

Abbildung 5-1: Ablauf des Einlesens

Da das Einlesen der Eingabe gepuffert erfolgt, dabei handelt es sich um eine Zeichenkette. Es handelt sich dabei um eine Eingabe auf der Konsolenebene. Es sollten wie vorher erwähnt alle Ausgaben, die auf die Konsole gehen, zentral gesteuert werden. Das bedeutet, dass dafür eine neue Klasse entsteht, mit der das ermöglicht werden kann. Wenn diese Klasse nun sowieso schon mit der Konsole zusammenarbeitet (Abbildung 5-1 – Punkt 1 & 2), kann die Eingabe auch von dort gelesen werden. Im Prinzip heißt das nichts anderes, als das die VaCli Main-Methode ein Objekt der Klasse „Console“ erzeugt, in welcher eine Methode festgelegt ist, die etwas gepuffert einliest und wieder zurück gibt. In der Klasse „Console“ müssen dafür zwei Dinge festgelegt werden, als Erstes ein Objekt, welches für das gepufferte Einlesen zuständig ist. Dieses Objekt muss dafür vom Typ BufferedReader sein. Dem Objekt stehen nun mehrere Funktionen aufgrund seines Typs zur Verfügung. Für diesen Fall ist die Funktion „readLine()“ von Interesse. Die Funktion liest einen String von der Konsolenebene erst, wenn dieser durch ein Enter abgeschlossen wurde. Die Funktionalität von readLine() muss noch einer Methode zugewiesen werden, damit darauf zu einem beliebigen Zeitpunkt von außen zugegriffen werden kann. Eine Aufteilung des gelesen Strings erfolgt jedoch nicht in der erstellten Methode der Klasse „Console“, da der String sowohl in seiner

ursprünglichen Form sowie auch in der getrennten benötigt wird. Das Ergebnis des eingelesenen Strings wird von der aufrufenden Klasse, in diesem Fall der VaCli, auf einer Variablen des Typs String zwischengespeichert. Die Variable wird nun verwendet, um die History des Programmes zu erstellen, in dem sie in einer Textdatei gespeichert wird. Ebenso wird die Variable auch verwendet, wenn es zur Anwendung des „recordmacro"-Kommandos kommt.
Für das Bestimmen des Kommandos, welches immer an der ersten Stelle zu finden sein muss, wird die Variable nach dem Leerzeichen aufgetrennt. Da die Variable vom Typ String ist, steht die Funktion „split()“ zur Verfügung. Diese teilt den String nach einem vom Entwickler bestimmten regulären Ausdruck auf und speichert die einzelnen Teilstücke der Trennung in einem Array. Array-Objekte sind eigentlich nichts anderes als eine Liste, auf deren Einträge beliebig zugegriffen werden kann. Zu bedenken ist nur, dass, wenn ein Eintrag geändert werden soll, dieser den vom gleichen Typ wie das Array sein muss. Die Nummerierung der Einträge in der List startet immer mit 0, dass bedeutet, das gesuchte Kommando steht immer an der Array-Position 0. Die vorgeladenen Kommandos befinden sich mit einem Indexschlüssel und einem Wert in einer Map. Der Indexschlüssel ist genauso wie das Array vom Typ String, das heißt, es ist einfach Vergleichsoperationen über dem Schlüssel und dem Kommando von Array-Position 0 vorzunehmen. Der Wert der Map ist vom Typ Command, welches auf die Klasse des auszuführenden Kommandos verweist. In einem weiteren Schritt wird nun jedes Indexschlüsselelement der Map mit dem Kommando aus der getrennten Eingabe verglichen. Sind zwei Werte identisch, so wird die getrennte Eingabe an die Klasse des dazugehörigen Kommandos weitergeleitet, Abbildung 5-1 – Punkt 3. Wird jedoch kein Treffer erzielt, so gibt es einen Fehler, der dem Nutzer mitgeteilt wird, damit dieser gegebenenfalls seine Eingabe verändern kann.

5.2 Analyse mit dem Parser

Die Analyse auf die geforderten Optionen übernimmt der zuvor schon genannte Parser. Im Verlauf des Projektes gab es mehrere Möglichkeiten die Analyse der Nutzereingabe zu realisieren. Diese waren die Entwicklung eines eigenen Parsers oder die Nutzung von durch Andere entwickelte Projekte. In der ersten Phase des Projektes wurde eine eigene Parserklasse entwickelt. Für wenige Kommandos war die Klasse auch sehr gut einsetzbar, nur mit der steigenden Anzahl implementierter Befehle wurde das Ganze immer unflexibler. In dieser eigenen Klasse wurde die zuvor getrennte Nutzereingabe auf die statischen Optionen hin untersucht. Es gab dabei keine Unterscheidung, ob eine Option einen Wert erforderte oder nicht. Ebenso existierte auch keine zweite Möglichkeit eine Option auszudrücken, zum Beispiel konnte für die Angabe eines Graphen nur der Ausdruck „-g“ verwendet werden und nicht auch noch „--graph“. Das wären in der Eigenentwicklung zwei unterschiedliche Optionen gewesen. Die Erweiterbarkeit des Parsers wurde auch dadurch erschwert, dass jede Option eine fest verankerte Methode der Parser Klasse war. Ein Vorteil der eigenen Klasse ist jedoch, das der Entwickler weiß, von welcher Option er, was erwartet. Ist eine Option für ein File zuständig, so lieferte der Parser gleich ein File-Objekt zurück. War es eine Map, so wurde eine Map zurückgegeben. Es wurde allerdings auch eine Analyse von anderen Parser Projekten vorgenommen, wie schon in Kapitel 3.1 erwähnt. Durch ein Hinzufügen der Programmbibliothek steht nun ein Parser zur Verfügung, der im Gegensatz zu der Eigenentwicklung auch bei einer großen Anzahl von Kommandos noch sehr flexibel erweiterbar ist. Die Instanzzierung des Parsers erfolgt in der Klasse „AbstractCommand“ (Abbildung 5-2).

```
33      // START Parser initiation
34      /**
35       * This object includes all operations for any option of the parser. Like
36       * add new Options, has Option or get a value for the option.
37       */
38      protected Options opt = new Options();
39      /**
40       * Use this object to print the help informations for all registered
41       * options.
42       */
43      protected HelpFormatter hf = new HelpFormatter();
44      /**
45       * This Object can parse a <code>String</code> Array by the registered
46       * options.
47       */
48      protected BasicParser parser = new BasicParser();
49      /**
50       * This object can check if an option is set in the user input and return
51       * it's value.
52       */
53      protected CommandLine cl = null;
54      // END Parser
```

Abbildung 5-2: Instanzzierung des Parsers

Die Registrierung von Standardoptionen in der abstrakten Klasse der Kommandos erwies sich nicht als praktikabel. Auch wenn es einige Optionen gibt, die für mehrere Kommandos verwendetet werden könnten, so würden diese auch in der Benutzungshilfe des Kommandos auftauchen, wenn es sie gar nicht verwendet werden. Die Hilfe wäre dadurch nicht mehr kommandospezifisch und würde den Nutzer eher verwirren als helfen. Bei der Hilfe zu den Optionen stehen zwei Möglichkeiten zur Verfügung. Die Erste wäre das Benutzen der vom Apache CLI gebotenen Funktionalitäten. Dafür ist es notwendig, ein Objekt des Typs „HelpFormatter“ zu erstellen. Anbieten würde sich wieder dies in der abstrakten Klasse zu tun. Dieses Objekt stellt eine Reihe von Möglichkeiten der Formatierung der Optionen zur Verfügung und arbeitet Hand in Hand mit den registrierten Optionen. Die zweite Möglichkeit wäre es eine eigene Ausgabe nachzubilden. Die wäre dann noch flexibler als die des Apache CLI, allerdings ist es etwas umständlicher, auf die einzelnen registrierten Optionen zuzugreifen. Die Wahl fiel auf die nachgebildete Ausgabe, da diese sich als String zurückgeben lässt und ein String im Help Kommando benötigt wird, um die Kommandoübersicht darzustellen. Auch die Nachbildung erwies sich nicht als besonders komplex, da alle Funktionalitäten die benötigt wurden vom Apache CLI mitgeliefert werden. Abbildung 5-3 zeigt, wie der Rückgabewert gebildet wird.

```
131     /**
132      * Displays usage information for the command.
133      * @return <code>String</code> with usage information.
134      */
135     public String usage() {
136         String usage = "usage: " + name() + "\n";
137         for (Object o : opt.getOptions()) {
138             usage = usage
139                     + (" -"
140                     + ((Option) o).getOpt()
141                     + (((Option) o).hasLongOpt() ? ", --" + ((Option) o).getLongOpt() : "")
142                     + (((Option) o).hasArg() ? " <" + ((Option) o).getArgName() + ">" : "")
143                     + "\t" + ((Option) o).getDescription() + "\n");
144         }
145         return usage;
146     }
```

Abbildung 5-3: Bilden der usage()-Informationen

Es werden dabei die von der Kommandoklasse registrierten Optionen abgefragt. Weil es sich dabei um eine für die Methode unbekannte Anzahl an Optionen mit unbestimmten Merkmalen handelt, muss jedes Objekt das aus der „Collection“ der „getOptions()“ Funktion kommt einzeln behandelt werden. Von jeder Option ist bekannt, dass sie mindestens eine Short-Option, kurz Darstellung, und eine Beschreibung enthält. Es kann jedoch auch noch eine Long-Option, lang Darstellung, und ein erwartetes Argument zu einer Option enthalten. Damit der Rückgabestring immer einheitlich aussieht, wird in Zeile 141 und 142 dynamisch geprüft ober der String noch erweitert werden muss. Wenn die Prüfung nicht durchgeführt werden würde, könnten in dem String NULL-Werte enthalten sein. Nun kann es aber auch vorkommen, dass einige Kommandos keine Optionen verwenden. Bei diesen Kommandos wird ebenfalls die usage()-Methode verwendet. Allerdings wird dabei nicht mit dem Befehl „super“ auf den Inhalt der AbstractCommand Klasse verwiesen, da im Parser keine Optionen registriert sind. Sondern ein spezifischer Rückgabestring wird in der Klasse des Kommandos erstellt, der dem des AbstractCommand jedoch ähnelt.

Um das Problem mit der kommandospezifischen Hilfe Darstellung zu vermeiden, werden alle Optionen, die ein Kommando verwendet, in der Klasse des Kommandos registriert. Ein Konstruktor, der beim Aufruf der Kommandoklasse ausgeführt wird, sorgt dafür (Abbildung 5-4).

```
 99     /**
100      * Define Parser options
101      */
102     public MetricsCommand(){
103         opt.addOption("g", "graph", true, "Internal name for the graph");
104         opt.addOption("n", "newname", true, "Optional: Internal name for the new generated graph");
105         opt.addOption("m", "metric", true, "Metrics: <metric[;metric;...]>");
106     }
```

Abbildung 5-4: Definierung der Parser-Optionen

Im Vergleich zur Eigenentwicklung kann jede Option nun mehrere Merkmale besitzen. Zuvor war es lediglich möglich einen kurzen oder langen Optionsnamen zu belegen. Mit dem Apache CLI kann nun beides für ein und dieselbe Option verwendet werden. Angesprochen wird die kurze Option dabei über ein einzelnes Minus (-) und die lange Version über zwei aufeinanderfolgende (--). Des Weiteren kann jetzt festgehalten werden ob eine Option einen Wert erwartet oder nicht. Das wird mit der Hilfe eines logischen Ausdrucks bewerkstelligt. Dieser ist „true“ wenn ein Wert erwartet wird und „false“ wenn nicht. Zuvor gab es in der Eigenentwicklung Probleme damit, dass der Parser versagte, wenn der Nutzer vergessen hatte einen geforderten Wert hinter die Option zu setzten.

Beispiel 5-1: Problem mit korrekter Schreibweise

korrekte Schreibweise	Kommando -g Argument -p Argument
falsche Schreibweise	Kommando -g -p Argument

In Beispiel 5-1, wurde falscher Schreibweise interpretiert das „-p“ ein Argument/Wert von „-g“ ist. Das ist aber offensichtlich falsch, daher ist das Merkmal ob die Option einen Wert fordert sehr sinnvoll und hilfreich. Es ist nun leicht möglich, diesen Fehler abzufangen. Ein weiterer Fehler der mit einführen des Apache CLI behoben wurde, ist das Argumente, die mit einem Minus beginnen nicht mehr als Option wahrgenommen werden. Da diese nicht beim Parser registriert sind, ein Wert wie „-1“ wird jetzt als gültiger Wert erkannt.

5.3 Zentrale Ausgabensteuerung

Ein Weiterer wichtiger Punkt, der noch näher erläutert werden soll, ist die zentrale Steuerung aller Ausgaben auf die Konsole durch die Klasse „Console“. Dadurch, dass allen Klassen die auf die Klasse „Console“ zugreifen, steht das gleiche Spektrum an Methoden zur Verfügung und es kann eine einheitliche Regelung erfolgen.

Die Breite der Konsole wird in der Klasse definiert. Aktuell beträgt sie 80 Zeichen, das heißt alles, was über die 80 Zeichen hinaus geht, wird vom System umgebrochen und in einer neuen Zeile weiter geführt. Zuständig dafür ist die Methode „split()“, die in einer Schleife zeichenweise untersucht, wann ein übergebener String die maximale Länge erreicht hat und an dieser Stelle eine neue Zeile beginnt. Dem Entwickler steht noch eine Reihe von weiteren Methoden, mit deren Hilfe die Ausgabe definiert werden kann, zur Verfügung. Es besteht durchaus

die Möglichkeit einer „println()“-Methode verschiedene Optionen mitzugeben. Für ein Einrücken der Ausgabe kann ein Präfix der „println()“-Methode mitgegeben werden. Allerdings muss dabei auch ein Suffix definiert werden, dieses kann jedoch leer sein. Dadurch, dass ein leeres Suffix, oder auch Präfix übergeben werden kann, muss die Methode nicht mehrmals implementiert werden. Statt drei Methoden, um alle Eventualitäten abzudecken, existiert nur eine, die mit allen drei Fällen klarkommt. Bei der Verwendung der „println()“-Methode kann es jedoch zu Verschiebungen kommen, da das Präfix und Suffix bei einem Aufruf jedes Mal von Neuem festgelegt werden müssen. Um das Problem auszuschließen, wird die Ausgabe über „println()“ mit einem „MessageType“ und einem String vorgenommen. „MessageType“ wurde zuvor in der Klasse „Console“ definiert. Es handelt sich dabei um eine Aufzählung, in der die Elemente festgelegt werden, die vorkommen können.

```
24      private int intend = 0;
25      private char intendChar = ' ';
26      private final int columns;
27
28      public static enum MessageTypes {
29          NORMAL, INFORMATION, WARNING, ERROR, COMMANDPRINT
30      }
105     /**
106      * Prints a String by using the method <code>formatString()</code> and <code>MessageTypes</code>
107      * The output is generated by the <code>MessageType</code>.
108      * @param s
109      *      Print String
110      *@param type
111      *      <code>enum MessageType</code>
112      */
113     public void println(MessageTypes type, String s) {
114         String prefix = "";
115         String postfix = "";
116         switch (type) {
117         case NORMAL:
118             break;
119         case COMMANDPRINT:
120             prefix = "    ";
121             postfix = "";
122             break;
123         case INFORMATION:
124         case WARNING:
125         case ERROR:
126             prefix = "    * " + type.toString() + ": ";
127             postfix = " *";
128             break;
129         default:
130         }
131         println(prefix, postfix, s);
132     }
```

Abbildung 5-5: Ausgabe mit MessageTypes

Bei den Elementen „INFORMATION, WARNING und ERROR“, wird eine einheitliche Ausgabe generiert. Die enthält sowohl ein Präfix (Abbildung 5-5 Zeile 126) und ein Suffix (Abbildung 5-5 Zeile 127), auf die Konsole geschrieben wird das mit der zuvor erwähnten Methode „println()“ (Abbildung 5-5 Zeile 131). Die drei Elemente weisen den Nutzer drauf hin, dass etwas bei der Ausführung nicht in Ordnung war.

Das Element „COMMANDPRINT“ ist dafür zuständig, dass alles, was eine gültige Ausgabe ist, einen einheitlichen Standard hat. Dazu wird, wie bei den drei Elementen zuvor, ein Präfix und ein Suffix definiert und an „println()“ weitergereicht. Immer, wenn ein Kommando eine Ausgabe erzeugt, die nicht auf einen Fehler hinweist, wird dieser „MessageType“ verwendet.
Bei dem Element „NORMAL“ wird keine Formatierung des Strings vorgenommen.
Generell ist die Festlegung so, dass Kommandos immer ganz links in der Konsole stehen und dass alle Ausgaben von Kommandos immer um vier Leerzeichen eingerückt sind. Das hat den Vorteil, dass bei einer längeren Anwendungszeit Informationen optisch schneller herausgefiltert werden können. Zu sehen ist das an Abbildung 5-6, es kann eindeutig zwischen dem Kommando und der Ausgabe des Kommandos unterschieden werden. Bei dem Kommando „loadgraph“ wird ab einer bestimmten Größe der übergebenen Datei ein „Fortschrittsbalken“ eingefügt. Allerdings nicht in der bekannten Form, wie es bei GUI-Anwendungen üblich ist. Bei diesem Balken werden die gelesenen Zeilen und Eigenschaften des Graphen aktualisiert.

```
VizzAnalyzer Command-Line Interface 2.0.0
------------------------------------------
Loading GraphProperties
 Loaded Common Meta-Model 2.0.7 successfully.
 Loaded Eclipse Java Front-end 1.0 successfully.
 Loaded Software Quality Model 1.1 successfully.
 Loaded Dynamic Analysis Model 1.0 successfully.
 Loaded Metric Suite 1.0 successfully.
Loaded 41 commands.

Type 'help' for viewing a list of available commands.
Type 'exit' for quitting the VizzAnalyzer Command-Line Interface.

> loadgraph test_large.gml
Read 54000 lines, 3747 nodes, 0 edges. Using 5507 KByte heap memory.
(2528 ms)

    Loaded graph: 1 - se.vxu.msi.grail-AST_defined
    (2582 ms)

> showgraphs
    Loaded graphs
    1   se.vxu.msi.grail-AST_defined    C:\Users\Daniel\Hochschule\Bachelorarbeit\wor
    -> kspace\se.arisa.vizzanalyzer.cli\test_large.gml
    (4 ms)

> unloadgraph
    * ERROR: Wrong number of parameters                                          *
    usage: unloadgraph <internal graph name>
    (2 ms)

>
```

Abbildung 5-6: Konsolenausgabe

6 Schlussfolgerung und zukünftige Arbeit

In dem folgenden Kapitel wird auf die erreichten Punkte und deren Ergebnis Bezug genommen. Auch wird im zweiten Teil ein Problem dargestellt, welches bei einer Weiterentwicklung mit in Betracht gezogen werden sollte. Für dessen Lösung fehlte in dieser Arbeit jedoch leider die Zeit.

6.1 Schlussfolgerung

Die Hauptaufgabe, mit der sich die Bachelorarbeit befasste, bestand darin, eine Verbindung zwischen einem Nutzer und den Analysealgorithmen der Anwendung des VizzAnalyzer™ einzuführen. Diese Verbindung existierte schon zuvor in der Form eines „Graphical User Interface“ (GUI). Das Problem bei dem GUI war jedoch das wie es der Name schon besagt, grafische Elemente enthält. Diese Elemente sind nicht auf allen Systemen einsetzbar. Der Betrieb auf einem Server kann daher nicht garantiert werden. Ebenso ist es bei dem GUI nicht möglich die Applikation in einem Script-Modus zu starten, das heißt der Applikation beim Start eine Datei mitzugeben, die eine Reihe von Befehlen enthält, welche dann zur Ausführung gebracht werden. Die Lösung ist ein Interface auf der Kommandozeilenebene. Mit der Entwicklung des Interfaces und der Beschreibung der Lösungsansätze befasst sich diese Arbeit. Die im Kapitel 3.2 bestimmten Anwendungsfälle stellen die beiden Szenarien dar. Der zuvor erwähnte Script-Modus wird dem „normalen“ Anwendungsfall gegenübergestellt. Der Script-Modus ist sehr eng mit der Ausführung von Makros verbunden. Im Prinzip ist es auch nichts anderes, als dass dem Interface zum Start eine Macro-Datei mitgegeben wurde. Von daher funktioniert das Ganze zu fast 100 Prozent wie der „openmacro“-Befehl der Applikation. Es ist im Script-Modus möglich, sowohl eine eigens erstellte Marco-Datei auszuführen als auch die vom Programm selbst erzeugte Verlaufsdokumention (History) zu benutzen. Das Prinzip, das beide Dateien (Makro und History) den gleichen Grundaufbau haben, macht den Script-Modus zu einem sehr flexiblen Werkzeug. Dass dem Nutzer jederzeit eine unkomplizierte Durchführung von Analysen ermöglicht. In Kapitel 3.3 wurden alle Mindestanforderungen an das Interface dargelegt. Die Anforderungen sind alle insgesamt umgesetzt worden und können somit als erfüllt betrachtet werden. Das Interface ist so implementiert, dass es auf der Kommandozeilenebene funktioniert und keine grafischen Elemente eines üblichen GUIs aufweist. Es steht somit mit der Hauptaufgabe der Bachelorarbeit im Einklang. Durch die Verwendung eines Parsers, ‚beschrieben in Kapitel 5.2, ist es ermöglicht worden eine gute und automatisierte

Hilfe für den Nutzer zu erzeugen. Das geschieht anhand der vom Entwickler definierten Optionen, es handelt sich somit um eine kommandospezifische Benutzungshilfe. Generell ist der benutzte Parser des Apache Commons Projects ein überzeugendes Werkzeug zur Handhabung multipler Optionen für Kommandos. Neben der Generierung der Benutzungshilfe bietet der Parser auch geeignete Fehlermeldungen an, falls Probleme mit den Optionen auftreten sollten. Die im Kapitel 4 vorgestellten Klassen „AbstractCommand“ und „Console“ bieten dem Entwickler einen guten Ansatz für zukünftige Erweiterungen des Interface. Die Klasse „AbstractCommand“ sorgt für den Grundaufbau aller zukünftigen Kommandos und ist so eingerichtet, dass die Hilfe, die der Parser liefern kann, nur vom Entwickler angesprochen werden muss. Sollte der Entwickler nicht das verwenden wollen, was in der „AbstractCommand“ verankert ist, so stellt das auch kein Problem dar. Kurz gesagt ist die Klasse eine geeignete Hilfe für weitere Entwicklungen. Die Klasse „Console“ übernimmt, wie in Kapitel 4 angedeutet und in Kapitel 5.3 näher beschrieben, die zentrale Verwaltung der Ausgaben, die auf der Kommandozeilenebene dargestellt werden. Durch die Verwendung der dort definierten Methoden ist es also möglich die Form der Ausgaben aller Kommandos auf einen Schlag zuändern und so mit das komplette Erscheinungsbild der Applikation. Bei den zur Zeit 41 implementierten Kommandos nimmt die Klasse dem Entwickler enorm viel Arbeit ab.

Die Applikation stellt in ihrer Gesamtheit und Funktionalität ein durchaus leicht erweiterbares Interface dar. Das Einfinden in die Programmierweise sollte mit dieser Dokumentation ohne größere Probleme vonstattengehen.

Die Abbildung 6-1 listet die Anforderungen noch mal auf und zeigt den Status ihrer Implementierung. Die Anforderung A06 bezieht sich auf die implementierten Kommandos. Es konnten nicht alle Implementierungen getestet werden, deshalb erreicht die Anforderung A06 nicht den Status „ok“.

A01	ok
A03	ok
A04	ok
A05	ok
A06	X

A02	ok			
A02-A	ok		A02-F	ok
A02-B	ok		A02-G	ok
A02-C	ok		A02-H	ok
A02-D	ok		A02-I	ok
A02-E	ok		A02-J	ok

Abbildung 6-1: Anforderungsstatus

6.2 zukünftige Arbeiten

In zukünftigen Entwicklungen sollten die hier folgenden Punkte mit berücksichtigt werden. Es können Probleme bei der Ausführung von Makros auftauchen. Die Probleme ergeben sich aus der Vergabe der internen Zuordnung einer Nummer zu einem Graphen. Beim mehrfachen Ausführen von Makro Dateien, die den „loadgraph“-Befehl enthalten und danach wieder auf Graph zugreifen wollen, treten beim zweiten Mal Ausführen Probleme auf. Das folgende Beispiel erläutert das Problem.

Beispiel 6-1: Makro Problem

Die Voraussetzung für dieses Beispiel ist, dass zuvor kein Graph geladen wurde. Nun wird ein Makro ausgeführt. Beim ersten Durchgang wird ein Graph geladen und mit der der internen Nummer 1 versehen. Danach werden Analysen über den Graphen ausgeführt. Die Nummer des Graphen ist in diesem Fall ein fester Bestandteil der Syntax des Analysekommandos. Alles verläuft noch korrekt. Nun wird ein Zweites Makro zur Ausführung gebracht. Dieses lädt ebenfalls einen Graphen und setzt in seinen Analysekommandos voraus, dass der Graph die interne Nummer 1 hat. Die Nummer kann der Graph jedoch nicht haben, weil diese schon durch das Erste Makro belegt wurde. Das Zweite Makro arbeitet nun mit einem falschen Graphen, oder auch gar nicht, weil der Graph aus dem ersten Makro möglicherweise schon entfernt wurde, seine interne Nummer aber erst nach einem Programmneustart wieder vergeben wird.

Das Problem könnte gelöst werden, wenn die internen Nummern der Graphen in den Makros dynamisch wären. Also in den Makros selbst nur Platzhalter verwendet werden würden. Dabei würde der Platzhalter dann durch die letzte verwendete interne Nummer ersetzt.

Des Weiteren stehen einer Erweiterung der Applikation um neue Kommandos und Analysen keine größeren Hürden im Weg.

Anhang

Glossar

API	Application Program Interface Einfach ausgedrückt: Was für den Menschen das GUI, ist für ein Computerprogramm die API. Sie wird von Betriebssystemen, Services oder Bibliotheken bereitgestellt um Anfragen von Computerprogrammen zu ermöglichen. [5]
GUI	Graphical User Interface Ist die Schnittstelle zwischen dem Benutzer und dem Programm. Dies geschieht mit Hilfe von Grafiken, z.B. Bildern, Buttons und Tabellen.
String	Ist eine Zeichenkette bestehend aus einer beliebigen Anzahl von Zeichen z. B. „Hallo Welt!“
SWT	Standard Widget Tool

Quellenverzeichnis

[1] Requirements Engineering – Grundlagen, Prinzipien, Techniken;
Klaus Pohl; Heidelberg 2008

[2] Hauptseite der Arisa AB, http://www.arisa.se (13.08.2008)

[3] Hauptseite des SWT-Projekts, http://www.eclipse.org/swt (13.08.2008)

[4] Wikipediabeitrag zu SWT,
http://en.wikipedia.org/wiki/Standard_Widget_Toolkit (13.08.2008)

[5] Wikipediabeitrag zu API,
http://en.wikipedia.org/wiki/Api (29.08.2008)

[6] Internetpräsentation des Apache Commons CLI,
http://commons.apache.org/cli (05.09.2008)

[7] The VizzAnalyzer Handbook – Technical Report;
Thomas Panas, Rüdiger Lincke, Welf Löwe; Växjö University 2005

[8] UML 2.0 with VizzAnalyser;
Francisco Modesto; Växjö Universität 2007

[9] Compendium of Software Quality Standards and Metrics – Version 1.0
Lincke, Löwe, http://www.arisa.se/compendium (24.09.2008)

[10] VizzAnalyzer CLI - Command Manual;
Daniel Rohde; Växjö Universität 2008

[11] VizzAnalyzer CLI - Developer Manual;
Daniel Rohde; Växjö Universität 2008

Abbildungsverzeichnis

Beispielverzeichnis