

Studiengang: Softwaretechnik

Prüfer: Prof. Dr. rer. nat. Jochen Ludewig

Betreuer: Dipl. Ing. Rainer Schmidberger

begonnen am: 21. Oktober 2002

beendet am: 21. April 2003

CR-Klassifikation: D.2.9, H.5.2, J.4, K.3.2, K.6.1, K.6.3

Diplomarbeit Nr. 2050

**Entwurf und Implementierung einer
für Projektleiter realitätsnahen
Simulationsoberfläche für SESAM**

Markus Badstöber

Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
D-70565 Stuttgart

Inhaltsverzeichnis

Inhaltsverzeichnis	3
Abbildungsverzeichnis	7
1 Einleitung	9
1.1 Das Ausbildungswerkzeug SESAM	9
1.1.1 Idee	9
1.1.2 Motivation von SESAM	9
1.1.3 Umsetzung	10
1.2 Aufgabenstellung der Diplomarbeit	10
1.3 Aufbau der Diplomarbeit	11
2 SESAM	13
2.1 Benutzerrollen	13
2.2 Lernansatz	14
2.3 Teilsysteme	15
2.4 SESAM-Modelle	16
2.4.1 Informationsfluss	17
2.5 Das Qualitätssicherungsmodell	19
2.5.1 Kommandos des QS-Modells	20
2.5.2 Nachrichten des QS-Modells	21
2.5.3 Entitäten des QS-Modells	21
3 Motivation für eine neue Spieleroberfläche	23
3.1 Erfahrungen im Einsatz von SESAM	23
3.2 Erhoffte Verbesserungen	24
4 Konzepte für die Oberfläche	25
4.1 Graphische Benutzeroberflächen	25
4.1.1 Definition und Geschichte	25
4.1.2 Gründe für eine graphische Oberfläche	26
4.1.3 Entwicklungsziele einer Oberfläche nach Shneiderman	26
4.1.4 Risiken	27
4.2 Bisherige Bedienkonzepte	27
4.2.1 Darstellung der Nachrichten	28
4.2.2 Interaktionsmöglichkeiten	30
4.2.3 Stärken und Schwächen	30

4.3	Neue Ansätze zur Gestaltung der Oberfläche	33
4.3.1	Graphisches Konzept des GPI	33
4.3.2	Raumkonzept und Mitarbeiter	33
4.3.3	Darstellung der Nachrichten	34
4.3.4	Interaktionsmöglichkeiten	35
4.3.5	Stärken und Schwächen	36
4.4	Alternative Ansätze	36
4.4.1	Kontextabhängige Menüs	37
4.4.2	Befehlsorientierte Eingabe	37
4.4.3	Schwächen der alternativen Ansätze	38
5	Realisierung der Oberfläche	41
5.1	Entwicklungsansätze	41
5.1.1	Objektorientierter Ansatz	41
5.1.2	Einfachheit	42
5.1.3	Styleguide und Lesbarkeit	42
5.1.4	Document-View Muster	42
5.1.5	Plug-in Konzept	45
5.1.6	Auslagerung spielweiter Optionen	45
5.1.7	Versionskontrolle	46
5.1.8	Verwendung einer IDE	46
5.2	Spezifikation	46
5.2.1	Informelle Beschreibung der Anforderungen	46
5.2.2	Nichtfunktionale Anforderungen	48
5.3	Entwurf	50
5.3.1	Bisheriges SESAM-System	50
5.3.2	SESAM-System mit neuer Spieleroberfläche	51
5.3.3	Modellunabhängige Teile der Spieleroberfläche	52
5.3.4	Plug-ins	53
5.4	Implementierung	55
5.4.1	Schnittstelle zur Basismaschine	55
5.4.2	Schnittstelle für verteiltes Arbeiten	58
5.4.3	Entwicklung der Spieleroberfläche	58
5.4.4	Plug-ins	59
5.4.5	Graphische Umsetzung	59
5.5	Entwicklung weiterer Plug-ins	60
5.5.1	Änderungen an Plug-in unabhängigen Teilen	60
5.5.2	Erstellen einer Plug-in Klasse	61
5.5.3	Beispiel: Ein Plug-in für das QS-Modell	62
5.6	Test	64
5.7	Beispiel eines Spielverlaufs	65
5.7.1	Start des Programms	65
5.7.2	Spieldurchführung	65

6	Konzepte für verteiltes Arbeiten	69
6.1	Verteilte Systeme	69
6.2	Gründe für verteiltes SESAM	69
6.3	Möglichkeiten zur Umsetzung	70
6.3.1	Sockets	71
6.3.2	RPC	71
6.3.3	RMI	71
6.3.4	CORBA	72
6.4	Realisierung in der Spieleroberfläche	73
6.5	Risiken	74
6.6	Empfehlung	75
7	Zusammenfassung und Ausblick	77
7.1	Zusammenfassung	77
7.1.1	Vorgehensweise	77
7.1.2	Wahl eines Entwicklungsprozesses	78
7.1.3	Entwicklungsphasen	80
7.1.4	Aufwand und Ergebnisse der Arbeit	82
7.1.5	Untersuchung des Codes	84
7.1.6	Bewertung der Ergebnisse	85
7.1.7	Offene Punkte	86
7.2	Ausblick	86
7.2.1	Verbesserung der interaktiven Möglichkeiten	86
7.2.2	Entwicklung neuer Plug-ins	87
7.2.3	Halbautomatische Generierung von Plug-ins	87
7.2.4	Verteilte Ausführung und echte Mehrbenutzerfähigkeit	87
7.2.5	Feedbackunterstützung der Basismaschine	87
	Literaturverzeichnis	89
A	Gesamtübersicht des Entwurfs	91
A.1	Verzeichnisstruktur der Spieleroberfläche	91
A.2	Übersicht über die Klassen und Pakete	92
B	Begriffslexikon	97
C	Erklärung	103

Abbildungsverzeichnis

1.1	Überblick über das SESAM-System	10
2.1	Teilsysteme von SESAM	13
2.2	Lernerfolge durch positive Selbstverstärkung	15
2.3	Teile eines SESAM-Modells	17
4.1	Die aktuelle Spieleroberfläche von SESAM	28
4.2	Der Organizer von SESAM	31
4.3	Screenshot der Spieleroberfläche: Mitarbeiterraum	34
4.4	Spieleroberfläche eines typischen Abenteuerspiels	37
4.5	Alternative Spieleroberfläche mit Befehlsbuttons	38
5.1	Das Document-View Entwurfsmuster	44
5.2	Screenshot der Spieleroberfläche: Cafeteria	47
5.3	Screenshot der Spieleroberfläche: Nachrichtenfenster im Projektleiterraum	49
5.4	Architektur von SESAM heute	50
5.5	Neue Architektur von SESAM	52
5.6	Plug-in Konzept der GPI	53
5.7	Document-View Muster der Plug-ins	54
5.8	Erstellen und Einbinden des Shared-Objects	57
5.9	Überblick über den Entwurf der Plug-in unabhängigen Teile der Spieleroberfläche	59
6.1	Verteilung des SESAM-Spiels im Idealfall	70
6.2	Remote Method Invocation (RMI)	72
6.3	CORBA	73
6.4	Entwurf der Schnittstellen bei verteiltem SESAM	74
7.1	Definition von Softwareentwicklungsprozessen, IEEE Std. 610.12-1990	78
7.2	Definition des Spiralmodells, IEEE Std. 610.12-1990	79
7.3	Die Phasen der Diplomarbeit	81
7.4	Zeitplan geplant	82
7.5	Zeitplan real	83
7.6	Aufwand und Ergebnisse der Phasen	83
7.7	Soll- und Ist-Aufwandsverteilung	83
7.8	Analyse des Codes	84
A.1	Überblick über den Rahmen der Spieleroberfläche (ohne Plug-ins)	93

A.2	Überblick über das QS-Modell-Plug-in	94
A.3	Überblick über die Raumklassen	95
A.4	Überblick über die Konfigurationsdateien des QS-Modell-Plug-ins	96

Kapitel 1

Einleitung

Diese Diplomarbeit behandelt den Entwurf und die Implementierung einer graphischen Benutzeroberfläche für SESAM. Dieses Kapitel soll die notwendigen Grundlagen schaffen, mit denen die Diplomarbeit in den Zusammenhang mit dem Ausbildungswerkzeug SESAM eingeordnet werden kann. Im Anschluss wird das Thema der Arbeit genauer vorgestellt. Abschließend wird erläutert, wie die Vorgehensweise zur Lösung der Aufgabe aussieht.

1.1 Das Ausbildungswerkzeug SESAM

1.1.1 Idee

SESAM ist ein Ausbildungswerkzeug für angehende Projektleiter, in der Regel Studenten, die Erfahrungen mit der Durchführung eines Softwareentwicklungsprojektes erlangen wollen. Die Abkürzung SESAM steht für *Software Engineering Simulation by Animated Models*. SESAM ist ein Spiel, welches einen Software-Entwicklungsprozess simulieren kann. Der Spieler steuert den Verlauf eines Projekts und wird aufgrund bestimmter Eigenschaften seiner Vorgehensweise bewertet (vergleiche [Lud94]). Ziel eines Spielers ist es, ein Projekt zum Erfolg zu führen und aus den Fehlern vergangener Spiele zu lernen. SESAM simuliert das gesamte Projekt, also sämtliche Eigenschaften, Regeln und Zusammenhänge, die ein solches Projekt ausmachen. Einzig die Person des Projektleiters wird vom Spieler selbst übernommen. Von seinen Entscheidungen hängt die erfolgreiche Durchführung des Projekts ab.

1.1.2 Motivation von SESAM

Die Ausbildung von Projektleitern gestaltet sich schwierig. Ein angehender Projektleiter benötigt zum Lösen von Problemen in seinem späteren Arbeitsumfeld drei wichtige Eigenschaften: Intelligenz, Wissen und Kenntnisse. Ziel einer Ausbildung an einer Universität ist es, den Studenten Wissen und Kenntnisse über die Tätigkeiten und Aufgaben eines Projektleiters zu vermitteln. Wissen kann hierzu auf mehrere Arten erworben werden, durch Aneignung von Wissen anderer (z.B. durch Bücher oder Vorlesungen) und durch eigene Erfahrungen, vor allem Erfahrungen, die durch Fehlhandlungen entstehen. Leider ergibt sich aus dem Umstand, dass Fehler in realen Projekten oft erhebliche finanzielle Folgen haben, dass ein realer „Probier-Ansatz“ zwar wünschenswert, in der Praxis jedoch nicht durchführbar ist. Rein theoretisch erlangtes Wissen wird der realen Situation in einem Projekt jedoch auch nicht gerecht, und oftmals fehlt es den Lernenden am Verständnis für bestimmte Handlungen in entsprechenden Situationen, weil der reale Bezug weit weg und damit schwer vorstellbar ist.

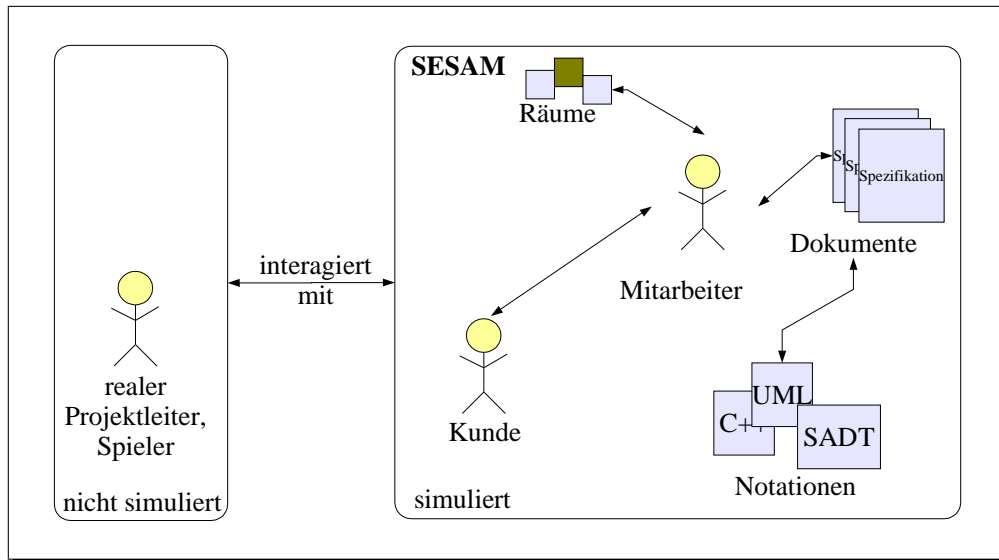


Abbildung 1.1: Überblick über das SESAM-System

Daher kam die Idee auf, ein Werkzeug zu erstellen, welches reale Situationen simuliert und mit dem ein Projektleiter gefahrlos üben kann. Der Spieler soll aus seinen Fehlern lernen können und begreifen, welche Auswirkungen sein Handeln und das Anwenden von Vorgehensweisen, wie sie in der Softwaretechnik gelehrt werden, hat. Im Gegensatz zu zeit- und eventuell kostenintensiven realen Projekten kann hierbei dem Spieler wertvolle Erfahrung ohne Risiko und mit geringem Kostenaufwand vermittelt werden, die er später in echten Projekten einsetzen kann.

1.1.3 Umsetzung

Wenn heute von SESAM gesprochen wird, ist damit im Allgemeinen das SESAM-2 System gemeint. Die erste Version des Simulators war als Prototyp gedacht, der schließlich durch die zweite, heutige Version ersetzt wurde. Das SESAM-System ist in Ada programmiert und wird mittels einer graphischen Benutzeroberfläche (GUI) gesteuert, die in Tcl/Tk implementiert wurde. Diese, im Folgenden Spieleroberfläche genannte Benutzerschnittstelle ist zur Zeit größtenteils textbasiert.

1.2 Aufgabenstellung der Diplomarbeit

Ziel der Diplomarbeit ist eine graphische Umsetzung des Modellzustands von SESAM und die Möglichkeit zur Interaktion mit dem Modell mittels dieser graphischen Benutzeroberfläche. Im Vordergrund steht die Erarbeitung von Konzepten, die eine möglichst realitätsnahe Bedienung für den Spieler darstellen. Realitätsnah bedeutet in diesem Zusammenhang, dass die Möglichkeiten der Eingabe, Speicherung und Transformation von Informationen so weit wie möglich dem entspricht, was auch in einem wirklichen Projekt zu erwarten ist. Der Spieler soll also keine Hilfe und Unterstützung bekommen, die er in einem realen Projekt nicht ebenso erfahren kann. Andererseits soll er in der Ausführung seines Spiels auch nicht unnötig eingeschränkt oder gar behindert werden. Anschließend soll die Oberfläche entworfen und prototypenhaft implementiert werden, wobei besonders darauf geachtet werden

soll, den Entwurf für eine mögliche verteilte Ausführung von SESAM offen zu halten. Die Implementierung soll in der Programmiersprache Java durchgeführt werden. Die offizielle Ausschreibung der Diplomarbeit ist bei [Sch02a] beschrieben.

1.3 Aufbau der Diplomarbeit

Kapitel 2 gibt einen Überblick über das SESAM-System und das für diese Diplomarbeit verwendete Qualitätssicherungs-Modell. Kapitel 3 erklärt die Gründe, weshalb eine neue Spieleroberfläche gewünscht wird und welche Erwartungen in eine neue Spieleroberfläche gesteckt werden. Kapitel 4 erläutert die Konzepte für eine graphische Benutzeroberfläche für SESAM. Anschließend wird in Kapitel 5 die Umsetzung einer graphischen Oberfläche beschrieben und ein Beispiel für ein Spiel gegeben. Kapitel 6 gibt einen Überblick über ein mögliches verteiltes Ausführen des SESAM-Simulators. Im letzten Teil der Diplomarbeit, Kapitel 7, wird eine Zusammenfassung der Diplomarbeit gegeben, gefolgt von einem Ausblick auf zukünftige Erweiterungen basierend auf dieser Arbeit.

Kapitel 2

SESAM

Dieses Kapitel gibt Aufschluss über die Teilsysteme von SESAM. Es soll erklären, welche Aufgaben die Teilsysteme abdecken und wie sie zusammenspielen. Zunächst werden dazu die einzelnen Benutzerrollen des SESAM-Systems näher erläutert. Spezielle Teile des SESAM-Systems werden erklärt, sofern diese Relevanz für die Diplomarbeit haben.

2.1 Benutzerrollen

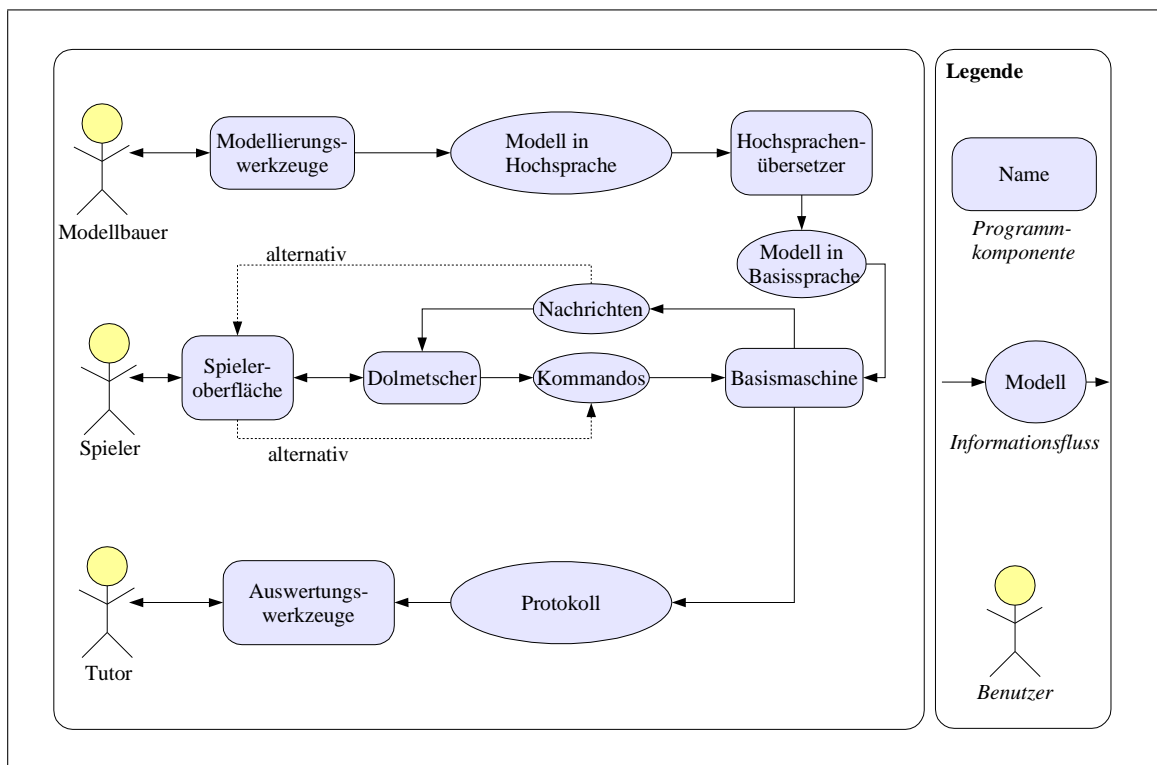


Abbildung 2.1: Teilsysteme von SESAM

Das SESAM-System wird von drei verschiedenen Benutzertypen verwendet. Der bereits erwähnte *Spieler* interagiert mit dem SESAM-System und steuert den Verlauf einer Simulation. Das SESAM-System ist in ein Schulungskonzept eingebunden, in dem ein *Tutor* die Durchführung der Simulationen analysiert. Das SESAM-System bietet hierfür eine Protokollschnittstelle an, mit dem der Tutor durchgeführte Spiele analysieren kann. Er kann damit den Spielern Feedback geben, so dass diese ihre zukünftigen Spiele gezielt verbessern können. Die dritte Benutzergruppe umfasst die *Modellbauer*. Sie entwickeln Modelle von Projekten, die die Grundlage der SESAM-Spiele bilden. Das Schaubild 2.1 auf der vorherigen Seite zeigt, wie die Gruppen jeweils mit dem SESAM-System arbeiten und in welchem Zusammenhang sie stehen.

2.2 Lernansatz

Der SESAM-Simulator soll nicht nur den Studenten eine Unterstützung bei der Ausbildung zum Projektleiter bieten, es soll auch den Entwicklern von Projektmodellen helfen, die Prozesse in einem Softwareentwicklungsprojekt besser zu verstehen. Beide Benutzergruppen lernen unterschiedliche Sachverhalte, beide auf unterschiedliche Art und Weise.

Man kann zwei Ansätze beim Lernen unterscheiden ([Aus74], [Bru73]):

- **Entdeckendes Lernen**
- **Rezeptives Lernen**

Beim *entdeckenden* Lernen erlernt man Sachverhalte auf spielerische Weise, also durch Ausprobieren. Man versucht, sich die Vorgänge zu erklären und Schlussfolgerungen daraus abzuleiten. Die Gefahr besteht darin, dass durch falsche Rückschlüsse ein falsches Wissen angeeignet wird. Es ist daher nur dann empfehlenswert, wenn der Lernende schon über genügend Hintergrundwissen verfügt, in das er seine Schlussfolgerungen sinnvoll einbetten kann. Eine gewisse Leitung bei der Erlernung ist daher sinnvoll. Zeitlich gesehen ist das entdeckende Lernen aufwändiger als das rezeptive. Das *rezeptive* Lernen hat einen größeren Einfluss auf die Wissensaneignung in einem Lehrumfeld wie der Universität als das entdeckende Lernen. Hier wird das Wissen der Lernenden durch Lesen von Publikationen oder das Lehren der Dozenten im Studium angeeignet. Es wird also vorhandenes Wissen anderer Menschen aufgenommen. Auch hier besteht die Gefahr, dass durch falsches Verstehen dieses Wissens falsch gelernt wird. Auch fehlt hier die eigene Erfahrung, die das Wissen in Kenntnisse umsetzt und somit zu einem vollständigen Verständnis des Problems führt. Das rezeptive Lernen ist zeitlich ökonomischer als das entdeckende. Vorwissen ist auch hier von Vorteil, ist aber je nach Themengebiet nicht unbedingt erforderlich. Rezeptiv bedeutet nicht, dass das Lernen passiv wäre. Auch hier muss der Lernende sich aktiv mit dem Lernmaterial befassen.

Ideal ist daher ein Lernansatz, der beide Lernweisen in sich vereint. Genau dieser Ansatz wird mit SESAM verfolgt. Das theoretisch vermittelte Wissen der Vorlesung zur Softwaretechnik soll nun vom Studenten aktiv erprobt werden. Dazu soll er mit dem SESAM-Simulator spielen und die Auswirkungen seines Handelns auf verschiedene Aspekte der Softwareentwicklung erfahren, z.B. die Auswirkungen auf die Qualität der Software oder die Motivation der Mitarbeiter. Um ein bestmögliches Ergebnis zu erzielen, wird die Simulation in ein umfangreiches Schulungskonzept eingebettet. Die Studenten bekommen in der Schulung Feedback über den Verlauf ihrer Spiele und Fehler, die sie gemacht haben und können so ihre zukünftigen Spiele verbessern.

Wie bereits erwähnt, dient SESAM aber auch dem Lernen der Modellbauer. Bei der Erstellung von Projektmodellen müssen Zusammenhänge im Projekt untersucht und modelliert werden. Der Modellbauer lernt somit bereits während der Umsetzung des Modells die Zusammenhänge eines Projektes zu

verstehen. Diese können durch Vergleichen der Ergebnisse in der Simulation und den Ergebnissen von realen Projekten Rückschlüsse auf die Qualität und Realitätsnähe der Modelle geben und somit dem Modellbauer Wissen über die Softwareprojekte vermitteln. So kann der komplexe Prozess einer Softwareentwicklung besser verstanden werden und in Zukunft dazu beitragen, dass Prozesse verbessert und gezielter eingesetzt werden können.

In [Lud94] ist beschrieben, dass viele Bücher zum Thema Software-Engineering zwar gute Ratschläge erteilen, wie man Software möglichst nicht entwickeln soll, dass aber positive Anleitungen rar sind, da die Auswirkungen noch wenig untersucht sind. Die Modellierung von Softwareprozessen soll dabei helfen, die realen Auswirkungen besser zu verstehen und damit die Qualität der Prozesse zu verbessern.

Für beide Lerngruppen stellt sich besonders dann ein Lernerfolg ein, wenn die Motivation zur Bedienung des Systems oder der Modellierung eines Prozesses möglichst lange anhält. Durch positive Erlebnisse bei der Durchführung einer Simulation zeigen sich positive Selbstverstärkungseffekte ([Sch02b], siehe Bild 2.2), durch die ein anhaltender Lernerfolg erst ermöglicht wird. Im Falle von SESAM soll das Lernen durch Spielen und die damit verbundenen Erfolgserlebnisse die Motivation der Studenten fördern und somit das Lernen des Softwareentwicklungsprozesses auf spielerische Weise unterstützen.

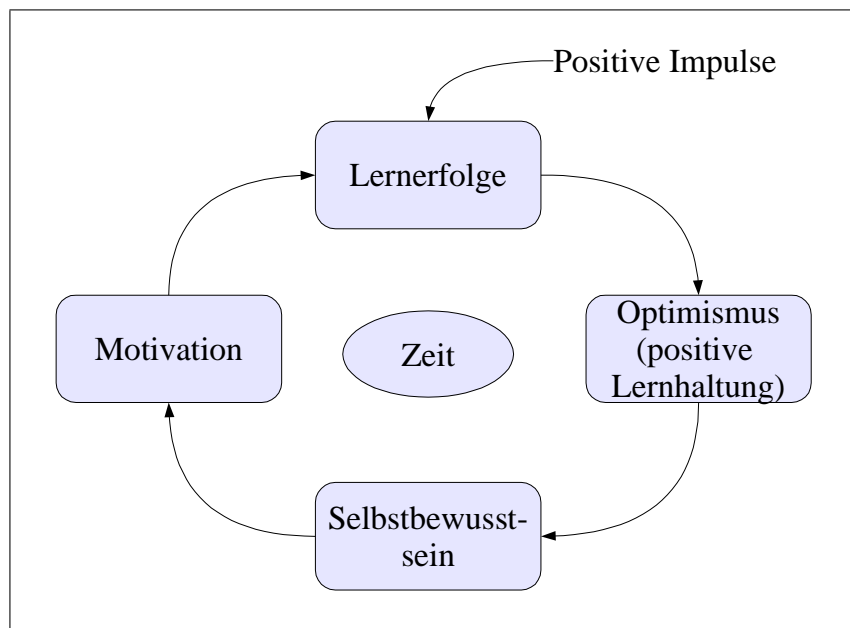


Abbildung 2.2: Lernerfolge durch positive Selbstverstärkung

2.3 Teilsysteme

SESAM besteht aus verschiedenen Teilsystemen, die über Schnittstellen miteinander kommunizieren. Abbildung 2.1 auf Seite 13 zeigt die verschiedenen Teilsysteme und Benutzerrollen des Systems. Der Modellbauer kann mittels spezieller Werkzeuge Hochsprachenmodelle erstellen, welche die Zusammenhänge in einem Projekt modellieren und diese mit einem Hochsprachencompiler in Code

übersetzen, der von der Basismaschine eingelesen und interpretiert werden kann. Dem Tutor stehen Auswertungswerkzeuge zur Verfügung, mit denen er den Verlauf eines Spiels analysieren kann. Der Spieler kann das SESAM-System über eine Spieleroberfläche bedienen, in welcher er Kommandos eingibt und Nachrichten als Antworten darauf erhält. Der Spieler sendet Kommandos, die er in (eingeschränkter) natürlicher Sprache eingeben kann. Die Modelle der Projekte beinhalten die entsprechenden Kommandos in formalisierter Sprache. Daher übersetzt ein Dolmetscher die Kommandos des Spielers in die Kommandos, die im Modell definiert sind und auf umgekehrtem Weg die Nachrichten, die von einer formalen Darstellung im Modell in eine natürlichsprachliche Variante für den Spieler gebracht werden müssen. Für bestimmte Zwecke, beispielsweise die automatische Ausführung von Spielen zu Analysezwecken, kann der Dolmetscher umgangen werden. Der Dolmetscher übersetzt die Spielereingaben und Nachrichten mit Hilfe eines Wörterbuchs. Die Teilsysteme Spieleroberfläche, Dolmetscher und Basismaschine sind die für diese Diplomarbeit relevanten Teile. Diese Teilsysteme stellen die Schnittstelle für den Spieler dar, der ein virtuelles Projekt als realer Projektleiter leiten möchte.

Zentrales Herzstück des SESAM-Systems ist die Basismaschine. Sie interpretiert die Projektmodellbeschreibungen und führt die Simulation aus. Die Basismaschine bietet hierzu eine Reihe von Schnittstellenfunktionen an, mit denen der Spieler beispielsweise Kommandos der Modelle aufrufen kann.

2.4 SESAM-Modelle

SESAM-Modelle sind Simulationsmodelle, die ein Software-Projekt modellieren. Die Modelle werden zur Zeit in der speziell für diesen Zweck entwickelten Hochsprache SEMOS-2 geschrieben. Im Folgenden ist mit *Hochsprache* daher SEMOS-2 gemeint. Da die Hochsprache sehr komplex ist, werden in dieser Diplomarbeit nur einzelne Ausschnitte vorgestellt. Eine genaue Beschreibung der Hochsprache findet sich bei [SMM99].

Die Hochsprachenmodelle haben eine große Bedeutung für die Aufgaben dieser Diplomarbeit, da sie unter anderem die Entitäten beschreiben, die in einer graphischen Umsetzung der Spieleroberfläche angezeigt werden müssen.

Die Hochsprache umfasst drei Teilmodelle, deren Beziehung in Bild 2.3 auf der nächsten Seite dargestellt ist:

- **Das Schemamodell**

Das Schemamodell beschreibt das Umfeld, in dem die Projektsimulation stattfindet. Es beinhaltet vor allem die Entitätstypen, die auftreten können, die Relationen zwischen den Entitäten und Attributstypen, ähnlich einem Entity-Relationship-Modell. Zusätzlich werden noch Nachrichtenarten definiert, die vom System ausgegeben werden.

- **Das Effektemodell**

Das Effektemodell definiert Regeln, Aktivitäten und Benutzerkommandos, die die Projektsimulation steuern. Dazu sind verschiedene Arten von Effekten definierbar. Da sie für die Spieleroberfläche nicht von Bedeutung sind, werden sie in dieser Diplomarbeit nicht weiter behandelt. Eine genaue Beschreibung der Basismaschine und des Effektemodells findet sich bei [Rei96]¹.

- **Das Situationsmodell**

Das Situationsmodell beinhaltet den Modellzustand. Konkret sind hier Instanzen der im Schemamodell definierten Entitäts- und Relationstypen enthalten. Das Situationsmodell wird während der Spielausführung durch das Effektemodell geändert. Ein besonderes Situationsmodell stellt die Startsituation dar. Sie beschreibt den Anfangszustand des SESAM-Modells. Hier wird zum Beispiel festgelegt, wie viele Mitarbeiter anfangs zur Verfügung stehen oder welchen Umfang das simulierte Projekt haben soll.

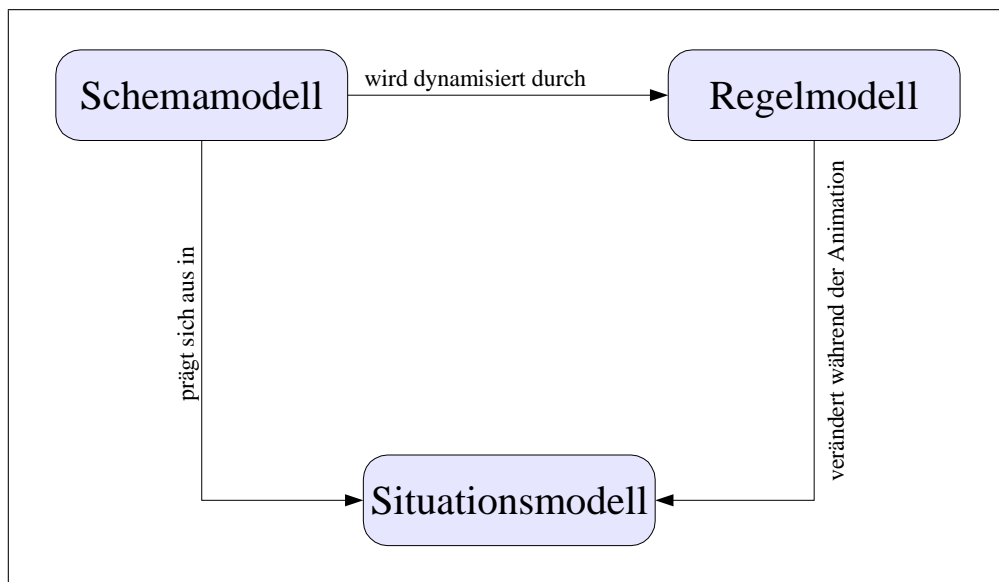


Abbildung 2.3: Teile eines SESAM-Modells

2.4.1 Informationsfluss

Der Informationsfluss sieht für den Spieler wie folgt aus.

- Startet der Spieler das SESAM-Spiel, indem er ein Modell lädt, so wird dieses Modell initialisiert. Gängigerweise wird eine Startsituation vorgegeben, die dem Spieler nach dem ersten Simulationsschritt im Spiel dargestellt wird. In dieser Startsituation sind meist der Projektumfang, die Ziele des Projekts, Anzahl der Mitarbeiter und ähnliche Eigenschaften definiert, über die der Spieler am Beginn der Simulation Informationen erhält.

Ein Beispiel für die Ausgabe einer Startsituation:²

¹In dieser Ausarbeitung wird statt des Begriffs Effektemodell der Begriff Regelmodell verwendet.

²Die Startsituation bezieht sich auf das QS-Modell, das im Folgenden noch beschrieben wird. Ausschmückende Sätze der Originalausgabe wurden gekürzt, um das Beispiel kompakt zu halten.

1999/08/02/08:00:

Good morning. [...]

The system comprises about 200.88 Adjusted Function Points (AFP). Since the system should be implemented using Ada95 this corresponds to approximately 9800 lines of code (LOC).

Since the system must reflect the change-overs planned in your customer's company, you must end the project at last on 2000/04/28/08:00. This means you must complete the project in 9 months. [...]

Because of the narrow time plan, the customer will agree to a compromise and gives you a budget of 450000.0 DM.
[...]

Since the accountancy is very important, the customer requires that the Code contains at least 95.0 percent of all his requirements.

Furthermore, the customer demands that the code contains at most 12.0 error(s) per 1000 lines of code. He will control that by performing an acceptance test.
[...]

Since the accountancy is very important, the customer requires that the Manuals contains at least 95.0 percent of all his requirements.

Since the system will be used by different, partly unexperienced users, the customer requires that the Manuals hardly contains any error. Each page of the Manuals should contain at most 0.5 errors per page.

1999/08/03/08:00

- Der Spieler kann mit dem Simulator spielen, indem er Kommandos an die Basismaschine sendet. Die Spielerkommandos werden in einer eingeschränkt natürlich-sprachlichen Form eingegeben, welche nicht direkt vom Modell verarbeitet werden können. Daher übernimmt ein Dolmetscher die Übersetzung der Spielereingaben in Kommandos, die das Modell ausführen kann.

Ein Beispiel einer Spielereingabe:

ask Bernd to talk to the customer

Die Umsetzung der Eingabe in ein Kommando durch den Dolmetscher sieht wie folgt aus:

```
lasse_mit_Kunden_sprechen(Bernd)
```

Die Definitionen für die Übersetzungen stehen im Wörterbuch, welches vom Dolmetscher verwendet wird.

- Die Basismaschine führt nun Effekte aus, die im SESAM-Modell beschrieben sind und ändert gegebenenfalls den Zustand des Modells.
- Die Nachrichten des Modells werden anschließend von der Basismaschine an den Spieler zurückgegeben. Ähnlich wie bei den Kommandos muss auch hier der Dolmetscher die formalen Nachrichten in eine natürlich-sprachliche Form umwandeln.

Eine formale Nachricht sieht beispielsweise folgendermaßen aus:

```
begonnen_mit_Abnahmetest()
```

Eine dazugehörige übersetzte Nachricht für den Spieler:

```
The customer takes the product in his company to make  
the acceptance test.  
This will take some time, because he wants to be sure  
about the product quality.
```

- Die Ausgabe der Nachrichten übernimmt eine Spieleroberfläche, die eine Schnittstelle zur Basismaschine besitzt. Über sie kann der Spieler auch die Kommandos absenden.

2.5 Das Qualitätssicherungsmodell

Das Qualitätssicherungsmodell, kurz QS-Modell³, ist ein Modell eines Softwareentwicklungsprozesses, das im Rahmen einer Dissertation entstanden ist [Dra99]. Das Hauptaugenmerk des Modells liegt auf Aspekten, welche die Qualität der Software beeinflussen. Der Spieler soll die Vorteile eines guten Prozesses und eines guten Managements erkennen. Die Qualität der Projektleitung wird vom Spieler definiert. Macht der Spieler Fehler, versäumt er beispielsweise, wichtige Qualitätssicherungsmaßnahmen zu treffen, so hat dies negative Auswirkungen auf die Produktqualität. Nutzt er die Möglichkeiten zur Sicherung der Qualität, so wird sein Produkt, und damit das Spielergebnis, verbessert.

Das QS-Modell verfolgt mehrere Ziele:

- Der Spieler soll das gesamte Projekt steuern. Er soll nicht nur in das Spielgeschehen eingreifen, sondern vielmehr das Spiel selbst bestimmen. Es hängt von ihm ab, wie sich das Spiel entwickelt. Das QS-Modell macht also keine Vorgaben, was der Spieler als nächstes durchführen muss.
- Der Spieler soll mit Schwierigkeiten in Berührung kommen, die auch für reale Projekte typisch sind. Beispielsweise sollen Informationen nur ungenau beschrieben werden, sofern keine Aktionen durchgeführt wurden, die eine genauere Analyse zulassen.

³Das QS-Modell wird im Englischen auch QA-Model genannt. QA steht für Quality Assurance.

- Der Spieler soll die Rolle des Projektleiters vollständig einnehmen. Er muss dazu einen ähnlich großen Handlungsspielraum haben wie ein realer Projektleiter.
- Der Spieler soll die Zusammenhänge zwischen seinem Handeln und den Auswirkungen auf den Projektverlauf und die Produktqualität lernen. Dazu gehört, dass der Spieler die Möglichkeit haben muss, Fehler zu begehen. Er soll also die Konsequenzen seines Handelns in der Simulation erfahren.
- Der Spieler soll realistische Werte für quantitative Aussagen in der Softwareentwicklung sehen, so dass er ein Gefühl für die Größenordnung in solchen Projekten bekommt.

Wie in jedem Modell, sind auch hier die Eigenschaften des Originals verkürzt umgesetzt [Sta73]. Das bedeutet, dass nur die für den Verwendungszweck relevanten Teile des Prozesses dargestellt werden. Das QS-Modell beinhaltet daher im Wesentlichen Kommandos für die Entwicklung, Prüfung und Überarbeitung von Software. Grundlage des Modells bildet eine Auswahl von Metriken, mit der die Zusammenhänge zwischen Aktivitäten und deren Auswirkung auf die Produktqualität beschrieben sind [Dra99].

Das QS-Modell verfügt über eine Schnittstelle, mit der ein Spieler mit dem Modell kommunizieren kann. Ihm stehen etwa vierzig Kommandos zur Verfügung, welche größtenteils dazu dienen, einem Mitarbeiter Aufgaben zuzuweisen oder Informationen zum Zustand einzelner Entitäten des Modells einzuholen. Das Modell soll Effekte abbilden, die in der realen Welt beobachtet werden können. Dazu werden zwei Annahmen getroffen, die für die Ausführung der Kommandos wichtig sind:

- Die Kommandos eines Spielers werden immer ausgeführt, auch, wenn der Zeitpunkt der Ausführung ungünstig erscheint oder keinen Sinn ergibt. Der Spieler darf also Fehler machen.
- Zudem wird jeder Entwickler die Ergebnisse seiner Arbeit immer dokumentieren. Mit diesen Dokumenten kann daraufhin weitergearbeitet werden.

Ich habe die Kommandos im Folgenden verschiedenen Aufgabenfeldern der Softwareentwicklung zugeordnet, ohne die konkreten Kommandos zu nennen, da diese zwar aussagekräftig, aber stark formalisiert sind.

2.5.1 Kommandos des QS-Modells

Die Kommandos des QS-Modells sind logisch gruppiert, obwohl dies für die Ausführung keine tragende Rolle spielt. Wichtig ist dieser Aspekt nur für die Wartung des Modells und war für die Entwicklung dieser Diplomarbeit hilfreich. Die Kommandos gliedern sich wie folgt:

- **Erstellen von Dokumenten**

Folgende Dokumente können während der Entwicklung erstellt werden:

- Analysedokument
- Spezifikation
- Systementwurf
- Modulspezifikation
- Code
- Handbuch

Zudem fällt in diese Kategorie das Zuweisen von geeigneten Notationen zu den Dokumenten, wie beispielsweise UML für Spezifikation und Entwurf oder C++ als Programmiersprache. Ebenso gibt es ein Kommando zur Integration der Codeteile in das Gesamtsystem.

- **Test**

Hierbei stehen Kommandos für Tests der Module, der Integration der Module in das System sowie ein Test des gesamten Systems zur Verfügung. Am Ende der Entwicklung kann der Spieler einen Akzeptanztest vom Kunden durchführen lassen.

- **Korrektur**

Der Spieler hat die Möglichkeit, sämtliche Dokumente des Spiels korrigieren zu lassen.

- **Review**

Ebenso kann er Reviews für einzelne Dokumente vorschreiben. Hierzu zählen Reviews mit zwei oder drei Mitarbeitern sowie Reviews mit dem Kunden.

- **Inspektion**

Neben der Inspektion von Dokumenten können in dieser Kategorie auch die Ergebnisse der Reviews geprüft werden sowie verbrauchte Ressourcen, d.h. Zeit und Budget, festgestellt werden.

- **Personalaufgaben**

Hier stehen Kommandos zur Verfügung, die das Personal betreffen. Dies sind neben dem Einstellen und Entlassen der Mitarbeiter vor allem Kommandos, die Informationen zu den Entwicklern liefern.

- **Projektsteuerung**

Die Projektsteuerung des Modells umfasst Kommandos zum Beenden des Projekts und Ausliefern des Produkts an den Kunden sowie Kommandos zum Abrechnen oder Beenden von Aktivitäten.

2.5.2 Nachrichten des QS-Modells

Das QS-Modell beinhaltet neben den Kommandos auch die Definitionen der Nachrichten. Diese informieren den Spieler des SESAM-Systems über den Zustand des Modells und den Zustand der Basismaschine. Für die Spieler sind hauptsächlich die Nachrichten von Bedeutung, die den Zustand der Projektsimulation näher erläutern. Diese Spielernachrichten sind entweder Antworten auf Kommandos des Spielers, Meldungen beim Beginn oder Ende einer Tätigkeit sowie Meldungen zu Beginn oder Ende des Spiels. Neben den Spielernachrichten gibt das System weitere Informationen aus. Dazu gehören Nachrichten für den Tutor, Debug- und Fehlernachrichten. Insgesamt gibt es im QS-Modell etwa 110 Spielerausgaben, die in der Oberfläche graphisch umgesetzt werden müssen.

2.5.3 Entitäten des QS-Modells

Zu den Entitäten des QS-Modells zählen alle Dokumente, Notationen und Mitarbeiter. Die Startsituation des in dieser Diplomarbeit verwendeten QS-Modells sieht sieben Mitarbeiter für die Durchführung des Projekts vor.

Die Dokumente und Notationen umfassen:

- Dokumente

- Analysedokument
- Spezifikation
- Systementwurf
- Modulspezifikation
- Code
- Handbücher
- Testberichte
- Reviewberichte

- Notationen
 - Programmiersprachen
 - * Ada 95
 - * C
 - * C++
 - Natürliche Sprachen
 - * Englisch
 - * Deutsch für bestimmte Dokumente
 - Formale Methoden
 - * UML (Unified Modelling Language)
 - * SA (Structured Analysis)
 - * Z (eine formale Spezifikationsmethode)

Kapitel 3

Motivation für eine neue Spieleroberfläche

Dieses Kapitel erläutert, weshalb eine neue Spieleroberfläche für SESAM entwickelt werden sollte und welche Auswirkungen man sich von dieser Oberfläche erhofft.

3.1 Erfahrungen im Einsatz von SESAM

Die Erfahrungen verschiedener Testspiele haben gezeigt, dass ein großer Teil der Spieler mit der Bedienung des Systems unzufrieden ist [Opf02]. Für Unzufriedenheit im Umgang mit Computerprogrammen spielt die Zielerreichung bei der Bedienung einer Oberfläche eine entscheidende Rolle. In diesem Fall stellt das erfolgreiche Durchführen eines Spiels das Ziel eines Spielers dar. Die Motivation des Spielers und die Möglichkeit, ein gesetztes Ziel zu erreichen, ist von zwei Faktoren geprägt [BCL⁺02]:

- Die Wichtigkeit der Aufgabe.
- Die Überzeugung, dass der Spieler das Ziel erreichen kann.

Werden diese Faktoren durch das Spiel negativ beeinflusst, sinkt die Zufriedenheit. Die bisherige Spieleroberfläche lässt sich den *Graphical Character-based User-Interfaces* zuordnen: Benutzeroberflächen, deren Grundprinzip eine Textkonsole ist, die um ein Menü erweitert wurde. Die komplexen Zusammenhänge des SESAM-Modells können damit nur bedingt übersichtlich dargestellt werden. Tatsächlich geschieht es häufig, dass ein Spieler entweder nicht weiß, wie er weiter vorgehen soll, oder er kann seine Idee nicht umsetzen, weil das Programm dies behindert. Beispielsweise können viele Kommandos des Spielers nicht interpretiert werden, weil die Spracherkennung der Eingabebe fehle nicht mächtig genug ist.

Die neue Spieleroberfläche soll nun versuchen, den Modellzustand, soweit dies sinnvoll ist, graphisch darzustellen, um dem Spieler die wichtigen Informationen übersichtlich zu präsentieren. Der Spieler soll so einen besseren Überblick über das Projekt bekommen. Die bisherige Umsetzung war zudem nicht in allen Punkten nah an der realen Projektwelt. So ist beispielsweise die Historyfunktionalität der Textkonsole in einem realen Projekt nicht vorhanden. Wenn ein Projektleiter seine Informationen nicht speichert, sind sie nach einer gewissen Zeit verloren. Die neue Spieleroberfläche soll solche Situationen verhindern und eine möglichst wirklichkeitsnahe Umsetzung des QS-Modells darstellen.

Außerdem soll die neue Spieleroberfläche die Eingabemöglichkeiten erweitern und vereinfachen, so dass der Spieler mehr Möglichkeiten hat, seine Ideen im Spiel umzusetzen.

3.2 Erhoffte Verbesserungen

Die neue Spieleroberfläche soll zunächst den Lernerfolg der Studenten im Umgang mit SESAM erhöhen. Dabei soll eine höhere Motivation der Spieler zu besseren Ergebnissen führen, ohne jedoch das Spiel zu weit von einer wirklichkeitsnahen Simulation eines Projektes zu entfernen. Eine einfache Bedienung, die die Vorgänge des Projektleiters intuitiv umsetzen lässt, und die Visualisierung des Projekts mit allen Objekten, Mitarbeitern und deren Räume soll helfen, den Spieler stärker in die Simulationswelt einzugliedern. Man hofft, dass dadurch die Simulation nicht nur ein trockenes Lernmittel ist, sondern auch eines, das Spaß machen kann.

Zudem soll das Spiel allgemein attraktiver gestaltet werden, da eine Textkonsole heute nicht mehr zeitgemäß erscheint. Eine graphische Oberfläche, die die Möglichkeiten heutiger Computergraphik zumindest mehr ausschöpft als die bisherige Spieleroberfläche, sollte das Programm interessanter erscheinen lassen, so dass zukünftig mehr Studenten ein SESAM-Spiel durchführen wollen. Auch wenn das Spiel nicht kommerziell ist, muss man es dennoch (im übertragenen Sinn) verkaufen. Eine Simulation und der Einsatz in der Ausbildung ist nur dann erfolgreich, wenn eine breite Masse damit übt. Ein Spiel, das gerne gespielt wird, hat damit eine größere Chance auf Verbreitung und Akzeptanz bei den Studenten, was wiederum zu besseren Resultaten in der statistischen Auswertung der Spiele führt.

Kapitel 4

Konzepte für die Oberfläche

Benutzeroberflächen sollen für die meisten Programme einfach und intuitiv zu bedienen sein und eine große Unterstützung für den Anwender bieten. Dies gilt für SESAM nur eingeschränkt, da die realitätsnahe Simulation eventuell im Widerspruch mit der Bedienbarkeit des Systems steht. Daher soll in dieser Diplomarbeit ein Konzept für eine Spieleroberfläche erarbeitet werden, das die Bedienmöglichkeiten des SESAM-Simulators nah an denen eines realen Projektleiters anlehnt. Dazu wird beschrieben, wie eine solche Oberfläche aussehen kann und welche Auswirkungen die Konzepte auf die Bedienmöglichkeiten haben.

Zunächst wird erläutert, welche Eigenschaften allgemeine graphische Benutzeroberflächen haben sollen und wie diese in SESAM integriert werden können. Anschließend soll das bisherige Konzept und die Oberfläche, wie sie vor Abschluss der Diplomarbeit aussieht, beschrieben und analysiert werden. Die neuen Konzepte werden vorgestellt und hinsichtlich ihres Sinnes für die Bedienung und Realitätsnähe begründet. Hierzu werden jeweils Stärken und Schwächen der alten und neuen Konzepte erläutert.

4.1 Graphische Benutzeroberflächen

Bevor nun die Ansätze für eine graphische Oberfläche beschrieben werden, soll dieser Abschnitt zunächst einen Überblick über Benutzeroberflächen im Allgemeinen geben und die Gründe erläutern, die zur Entwicklung solcher Oberflächen geführt haben. Anschließend wird erklärt, welche Grundsätze bei der Erstellung von modernen Oberflächen gelten.

4.1.1 Definition und Geschichte

Im Gegensatz zu rein textuellen Darstellungen werden bei graphischen Benutzeroberflächen (GUI) Bildelemente wie Menüs, Icons, Pushbuttons und Fenster dargestellt. Die Eingabe einer solchen Oberfläche geschieht mittels verschiedener Eingabegeräte, darunter am häufigsten mit einer Maus oder einer Tastatur. Die graphische Benutzeroberfläche soll Informationen kompakt und übersichtlich darstellen und die Interaktion mit dem Programm vereinfachen. Ein wichtiger Aspekt stellt die intuitive Bedienbarkeit und Erlernbarkeit eines Systems mit graphischer Benutzeroberfläche dar. Eine Zwischenstufe zwischen Textkonsole und graphischer Benutzeroberfläche stellt das *Graphical Character-based User Interface* dar. Diese Darstellung verwendet Menüs, ist ansonsten jedoch textbasiert.

Die Art der Eingabe und das Aussehen der Bildelemente bezeichnet man häufig auch als *Look&Feel* eines Programms. Dieses Look&Feel ist in den letzten Jahren recht einheitlich geworden. Dadurch

erreicht man eine gewisse Standardisierung, was die Entwicklung von Schnittstellen für die Programmierung von graphischen Oberflächen (so genannte Toolkits) erleichtert (beispielsweise Java Swing). Ebenso verkürzt sich der Einarbeitungsaufwand für die Anwender. Die Geschichte der graphischen Benutzeroberflächen ist verhältnismäßig jung. Obwohl erste Ansätze für graphische Benutzeroberflächen bereits in den 40er Jahren gezeigt wurden, dauerte es bis Mitte der 70er Jahre, ehe mit der Alto von Xerox PARC eine echte graphische Benutzeroberfläche auf den Markt kam. Oberflächen, wie sie heute allgemein verwendet werden, setzten sich erst Mitte der 90er Jahre, vor allem durch die Verbreitung von PCs, durch.

4.1.2 Gründe für eine graphische Oberfläche

Die Einführung von graphischen Oberflächen hatte mehrere Gründe. Zum einen ließen die technischen Möglichkeiten moderner Bildschirme mehr Spielraum für graphische Gestaltung, zum anderen wurden die Programme immer komplexer. Diese Komplexität war mit reinen Textkonsolen kaum zu erfassen. Der Anwender kann bei Textkonsolen nur schwer erkennen, welche Eingaben er tätigen muss, um ein Problem zu lösen. Graphische Konzepte, beispielsweise Icons, haben hingegen die Möglichkeit, viel Information auf wenig Raum unterzubringen und die komplexen Eigenschaften und Daten eines Programms geordnet und übersichtlich darzustellen. Sicher war auch die Einführung der Maus als Zeigegerät ausschlaggebend für graphische Benutzeroberflächen, da hier eine einfache und einheitliche Eingabeart die Bedienung vor allem für ungeübte Anwender vereinfachte.

In der rechnergestützten Lehre spielen graphische Benutzeroberflächen eine besondere Rolle. Die Fähigkeit, Informationen aufzunehmen und zu speichern, ist beim Menschen vor allem visuell stark ausgebildet [Sch02b]. Daher sind graphische Oberflächen besonders geeignet, komplexe und vielfältige Informationen darzustellen. Den Anwendern solcher Lernsysteme können die wichtigen Punkte der Lehre dadurch nicht nur einfach vermittelt werden, sie haben auch später eine bessere Chance, das Gelernte im Gedächtnis wieder abzurufen, weil sie beim Benutzen des Systems eine visuelle Assoziation zum Lehrinhalt aufbauen können.

4.1.3 Entwicklungsziele einer Oberfläche nach Shneiderman

Oberstes Ziel einer graphischen Benutzeroberfläche ist es, den Anwender so weit wie möglich bei der Bewältigung einer Aufgabe mittels eines Programms zu unterstützen. Dazu muss eine Benutzeroberfläche alle relevanten Informationen sinnvoll geordnet und übersichtlich darstellen. In [Shn97] sind Regeln definiert, die für graphische Benutzeroberflächen angewendet werden sollen, um dem Anwender eine möglichst gut bedienbare Oberfläche zu präsentieren. Diese Regeln umfassen die folgenden Punkte:

1. **Streben nach Konsistenz**

Konsistente Oberflächen haben ein durchgängig gleiches Bedienkonzept. Für moderne Oberflächen bedeutet dies beispielsweise, dass eine Datei immer über einen Dateiauswahl-Dialog gewählt wird, nicht aber einmal über den Dialog und einmal per Texteingabe.

2. **Abkürzungen für erfahrene Anwender**

Erfahrene Anwender wissen oft, welche Funktion sie ausführen wollen und brauchen keine Menüstruktur, in der sie die Funktion suchen können. Für diese Anwender soll es Abkürzungen, z.B. Tastenkombinationen, für bestimmte Befehle geben.

3. **Informatives Feedback anbieten**

Dieser Punkt betrifft vor allem Operationen, die länger dauern oder den Zustand des Systems

verändern. Hier muss dem Anwender eine Rückmeldung gegeben werden, aus der er Rückschlüsse auf den Zustand des Systems ziehen kann.

4. Sinnvolle und abgeschlossene Gliederung von Dialogen

Hierbei spielt die Konsistenz der Oberfläche wieder eine wichtige Rolle. Dialoge, bei denen der Anwender nicht weiß, welche Eingaben das System erwartet, um eine Aufgabe zu lösen, beeinflussen die Bedienbarkeit negativ.

5. Einfache Fehlerbehandlung

Jeder Fehler, Systemfehler oder ein Anwenderfehler, benötigt eine Rückmeldung an den Anwender.

6. Reversibilität

Eine Oberfläche soll die Umkehrbarkeit von Bedienhandlungen zulassen. Begeht der Anwender einen Fehler oder möchte er eine Aktion rückgängig machen, muss ihm eine Undo-Funktion zur Verfügung gestellt werden.

7. Den Anwender als „Herren des Systems“ unterstützen

Das System soll den Anwender nicht lenken und ihm einen Lösungsweg aufzwingen, sondern Werkzeug für den Anwender sein. Der Anwender soll also dem Programm sagen, was es tun soll und nicht umgekehrt.

8. Kurzfristige Gedächtnisbelastung reduzieren

Menschen können in aller Regel nur etwa sieben Einheiten als individuelle Dinge unterscheiden (siehe [Mil56]). Werden mehr Einheiten wahrgenommen, so sieht man nur „Masse“. Für Menüs und Darstellungen von Optionen etc. muss daher beachtet werden, dass man dem Anwender nicht zu viele Punkte gleichzeitig anbietet. Eventuell muss man die Strukturen hierarchisch gliedern, um die Bedienung übersichtlich zu gestalten.

4.1.4 Risiken

Graphische Benutzeroberflächen sind für moderne Programme den Textkonsolen meist überlegen. Dennoch ist eine graphische Oberfläche nicht per Definition die beste Lösung. Oftmals werden Fehler in der Gestaltung solcher Oberflächen begangen, die dazu führen, dass der Anwender seine Aufgabe nicht oder zumindest nicht optimal zu Ende führen kann. Die Missachtung der Punkte von Shneiderman führen oft zu solchen Oberflächen, vor allem die Überladung von Oberflächen mit Information (Punkt 8), unnötige Verschönerungen und sinnlose Funktionalität behindern die Aufgabenbewältigung. Es gibt aber auch auf Entwicklungsseite einige Effekte, die man bei graphischen Benutzeroberflächen beachten muss. Die Entwicklung ist im Vergleich zu Textkonsolen aufwändiger. Die Illusion, man nehme ein gängiges Oberflächendesign und baue seine Oberfläche darauf auf, führt oft zu schlecht bedienbaren Programmen. Die Konzepte, beispielsweise auch bei der Gestaltung sinnvoller Icons, nehmen einen durchaus beträchtlichen Teil der Entwicklungszeit ein.

4.2 Bisherige Bedienkonzepte

Abbildung 4.1 auf der nächsten Seite zeigt einen Screenshot der aktuellen Version der Spieleroberfläche von SESAM. Man kann drei Bildteile erkennen: eine Menüleiste, ein Ausgabefenster und eine Eingabemaske. Bei der Ausführung eines Modells kann der Spieler Kommandos eingeben, die vom

System interpretiert und umgesetzt werden. Das System selbst gibt seinen Zustand mittels Textausgaben an den Spieler weiter. Das Spiel wird gestartet, in dem der Spieler im Menü ein Modell auswählt, welches vom System geladen wird.

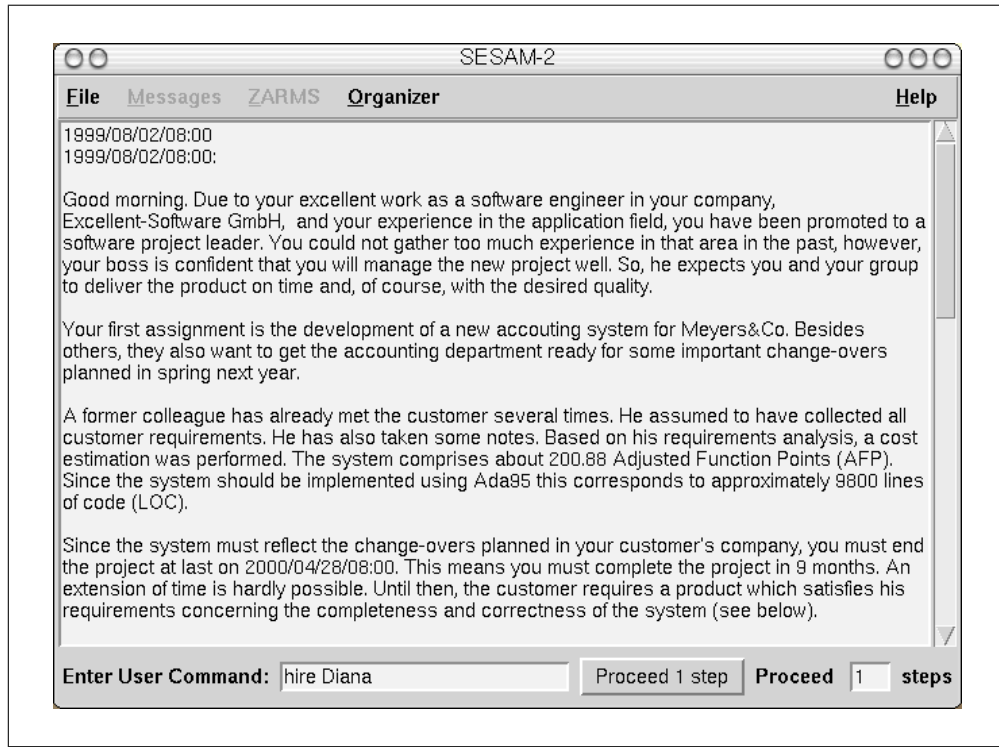


Abbildung 4.1: Die aktuelle Spieleroberfläche von SESAM

4.2.1 Darstellung der Nachrichten

Die Nachrichten gliedern sich in mehrere Arten auf. Die für das Spielen relevanten Informationen ergeben sich aus:

- **Spielernachrichten**

Spielernachrichten enthalten alle Informationen, die ein Spieler benötigt, um den Zustand des Modells zu erkennen. Konkret werden hier Informationen wie z.B. die aktuellen Tätigkeiten von Mitarbeitern oder die Zustände der Dokumente ausgegeben. Ein Beispiel für eine Spielernachricht sieht folgendermaßen aus (Die Nachricht wurde vom Dolmetscher bereits übersetzt):

```
1999/08/03/08:00 Bernd: The module specification
is done. It was fun.
Just give me the next job, I have some energy left.
```

Ohne den Dolmetscher wird die Nachricht ähnlich wie ein Methodenaufruf dargestellt, welcher von der Spieleroberfläche interpretiert werden kann. Spielernachrichten werden mit einem P

eingeleitet. Das vorangegangene Beispiel sieht ohne Verwendung des Dolmetschers wie folgt aus¹:

```
@PAUFGEHOERT_MODULE_ZU_SPEZIFIZIEREN(Bernd)
```

- **Tutornachrichten**

Tutornachrichten sind, wie der Name andeutet, Ausgaben, die der Tutor des Spielers nutzen kann, um zusätzliche Informationen zu den Aktionen des Spielers zu bekommen. Hier werden in kurzer Form alle Daten gezeigt, die Aktionen des Spielers und Reaktionen des Systems umfassen. Ähnlich den Spielernachrichten gibt es hier ebenfalls Ausgaben mit und ohne Dolmetscher. Die Tutornachrichten werden in der unübersetzten Methode mit einem T eingeleitet. Ein Beispiel:

```
@TPRINT_DATE (1999/08/04/08:00)
@TPRINT_STRING (Today is Wednesday)
@TPRINT_STRING (Project coaching done2)
```

- **Debugnachrichten und Fehlermeldungen**

Debug- sowie Fehlermeldungen sind für den Spieler uninteressant. Sie dienen den Entwicklern, vor allem den Modellbauern, während ihren Implementierungsaufgaben. Diese Fehlermeldungen umfassen lediglich Fehler des Modells oder der Basismaschine, nicht Eingabefehler des Spielers. Diese werden gesondert von der Spieleroberfläche oder dem Dolmetscher behandelt und ausgegeben. Debug- und Fehlermeldungen werden analog zu den anderen Nachrichten mit einem D beziehungsweise einem E eingeleitet. Typische Debugmeldungen umfassen die einzelnen Aufrufe im Modell. Nachfolgend ist ein kleiner Auszug aus den Meldungen bei der Einstellung eines Mitarbeiters mit anschließendem Simulationsschritt dargestellt.

```
@DDEBUG: TEILE_DEM_PROJEKT_ZU
@dDEBUG: DELTA_T_BERECHNEN
@dDEBUG: KOSTEN_FUER_MITARBEITER_BEI_5_TAGE_WOCHE_8H_TAG
@dDEBUG: KOSTEN_FUER_MITARBEITER_BEI_5_TAGE_WOCHE_8H_TAG
@dDEBUG: NACHT_UEBERSPRINGEN
@dDEBUG: REGELMENGE_LOESCHEN
@dDEBUG: WOCHENTAG_AUSGEBEN
@dDEBUG: Active_INITIALIZE
@dDEBUG: ABGELEITETE_ATTRIBUTE_DER_MODULSPEZ_BERECHNEN
```

Alle Nachrichten werden auf der integrierten Textkonsole ausgegeben. Die neuen Nachrichten werden einfach an das untere Ende der Ausgabe angehängt, so dass die neuesten Nachrichten ganz unten zu finden sind. Durch den Umstand, dass bei neuen Nachrichten die Textkonsole nicht geleert wird, ergibt sich automatisch eine Historyfunktionalität. Der Spieler kann also alte Nachrichten jederzeit anschauen, indem er mittels der Bildlaufleiste in der Konsole zur entsprechenden Nachricht fährt.

¹Die Motivation des Mitarbeiters, die in der natürlich-sprachlichen Variante der Ausgabe beschrieben wird, ist in diesem Fall zufallsgesteuert.

4.2.2 Interaktionsmöglichkeiten

Die Interaktion des Spielers mit dem System findet durch Eingabe von Kommandos statt, die das System interpretiert und ausführt. Dazu steht dem Spieler ein Eingabefeld zur Verfügung, in welches er Freitext eingeben kann. Möchte er das Kommando ausführen lassen, bestätigt er seine Eingabe mit *Enter*. Hier gibt es zwei Arten von Kommandos: Solche, die keine Simulationszeit in Anspruch nehmen und weitere, die mindestens einen Simulationsschritt benötigen, um ausgeführt werden zu können. So wird beispielsweise beim Kommando

```
inspect job of Bernd
```

sofort nach Bestätigen mit *Enter* ein entsprechender Text ausgegeben, in dem der Mitarbeiter seine aktuelle Beschäftigung mitteilt. Beim Kommando

```
ask Bernd to create the specification
```

wartet das System mit der Ausführung hingegen, bis der Spieler einen Simulationsschritt fortschreitet. Damit kann der Spieler mehrere Kommandos nacheinander eingeben und die Ausführung parallelisieren. Dieses Fortschreiten erreicht der Spieler durch Drücken des „Proceed 1 step“-Buttons. Hierbei wird genau ein Simulationsschritt vorangeschaltet, wobei die Simulationszeit für einen Schritt vom Modell abhängt. Im verwendeten QS-Modell entspricht ein Simulationsschritt genau 24 Stunden. Alternativ kann der Spieler auch eine höhere Schrittweite angeben, beispielsweise 5 Tage, indem er im Schrittweiten-Eingabefeld die entsprechende Zahl eingibt.

Die Spielereingaben werden mittels des Dolmetschers in Kommandos umgewandelt, die von der Basismaschine interpretiert werden können. Der Dolmetscher analysiert die Eingaben des Spielers und vergleicht sie mit vorgegebenen Sätzen, die in einer speziellen Form definiert sind. Bei der Erkennung unterscheidet der Dolmetscher Stichworte, die für ein bestimmtes Kommando vorkommen müssen, und Worte, die vorkommen können aber nicht zwangsläufig müssen. Eine Definition eines Kommandosatzes sieht damit beispielsweise wie folgt aus:

```
"[ask] [to] (talk|speak) [to|with] [the] customer",
ask "Who should talk to the customer?" => ein_Entwickler;
```

Die in eckige Klammern gesetzten Worte sind optional. Worte in runden Klammern sind Alternativen, von denen mindestens eine eingegeben werden muss, normal gedruckte Worte sind in jedem Fall erforderlich. Im obigen Beispiel steht im Anschluss an das Kommando eine Nachfrage des Dolmetschers, da das Kommando in der beschriebenen Form unzureichend spezifiziert ist. In diesem Fall fehlt als Argument der Name des Entwicklers, so dass der Dolmetscher beim Spieler den Namen des Mitarbeiters erfragt. Die Reihenfolge der Worte muss bei der Eingabe der Kommandos eingehalten werden.

Dem Spieler stehen weiterhin Zusatzprogramme zur Verfügung, die in die Spieleroberfläche von SESAM in Form des *Organizers* integriert sind (siehe Bild 4.2 auf der nächsten Seite). So kann der Spieler ein kleines Werkzeug zur Erstellung von Projektplänen, einen Kalender und ein Werkzeug für kleine Denkhilfen in Form von Klebezetteln verwenden, mit denen er Notizen erfassen und speichern kann.

4.2.3 Stärken und Schwächen

Eingabe der Kommandos Die Spieleroberfläche soll bewusst ohne graphische Darstellung auskommen. Die Eingabe geschieht über Freitext, um den Spieler in seiner Wahl der Kommandos nicht



Abbildung 4.2: Der Organizer von SESAM

einzu­schränken oder gar zu lenken. In [Sch94] ist beschrieben, dass die Implementierung eines Interpreters für Freitextererkennung zunächst nicht realisierbar erschien. Daher entschied man sich in der Implementierung von SESAM-1 für bewusst überladene und unübersichtliche Menüs. Diese Menüs sollten einerseits dem Spieler die Möglichkeit geben, die korrekten Kommandos einzugeben, andererseits sollten sie den Spieler nicht leiten. Das bedeutet, der Spieler sollte im Menü keinen Hinweis auf die richtige Bedienung der Simulation bekommen und allein durch sein Wissen entscheiden, welche Kommandos er als nächstes auszuführen hat. Er muss also *vor* der Benutzung des Menüs erkennen, welche Kommandos er einzugeben hat und diese anschließend im Menü suchen. Die Erfahrung im Einsatz des Simulators ergaben jedoch, dass die Leitung durch das Menü immer noch stark ausgeprägt war. Die Spieler betrachten das Menü und entscheiden danach, welche Kommandos in der aktuellen Spielsituation sinnvoll erscheinen. Dadurch ist das Spielverhalten stark geprägt vom Aufbau des Menüs und somit nicht realitätsnah. Das Prinzip der überladenen Kommandomenüs wurde daher zugunsten der reinen Freitexteingabe in SESAM-2 aufgegeben.

Textausgabe Bei der Ausgabe der Nachrichten spricht sich [Sch94] gegen graphische Oberflächen aus, wie sie in gängigen Abenteuerspielen zu finden sind, da der Spieler die Situation im Projekt zu schnell erfassen und überschauen kann. Er erkennt die relevanten Details zu schnell und wird in der Wahl seiner Kommandos beeinflusst. Hier muss beachtet werden, dass diese Aussagen zu einer Zeit gemacht wurden, als die graphischen Fähigkeiten der Rechner noch recht eingeschränkt waren. Für Abenteuerspiele aus dieser Zeit war die Darstellung hochauflösender und damit detailreicher Graphik nicht möglich. Eine reine Textausgabe fordert zwar die Phantasie der Spieler und sie müssen sich zwangsläufig mit den komplexen Informationen länger auseinandersetzen als bei graphischen Darstellungen; Allerdings wird der Spieler hier in eine Situation gebracht, die er in der Realität so nicht vorfindet. Die Aufbereitung der Textinformationen in eine bildliche Vorstellung des Zustands kann

dazu führen, dass der Spieler den Überblick über Dinge verliert, die er in einem wirklichen Projekt problemlos überschauen kann.

Dolmetscher Die zweite Version von SESAM, also die aktuelle Entwicklungsstufe, verwendet keine Menüs mehr, da sich zeigte, dass der Spieler zu stark gelenkt wird. Festgehalten wurde an der Freitexteingabe, deren Möglichkeiten durch einen Dolmetscher erweitert wurden. Der Dolmetscher ist ausführlich beschrieben in [Spi99]. Da der Dolmetscher, wie oben erwähnt, Sätze mittels Vergleich mit bekannten Sätzen analysieren kann, ist die Erkennung korrekter Kommandos eingeschränkt. Der Spieler muss bei seinen Sätzen nicht nur die korrekten Fachausdrücke erwähnen, was sinnvoll ist; Er muss ebenso den Satzbau einhalten, den sich ein Modellbauer bei der Erstellung der Erkennungssätze ausgedacht hat. Tatsächlich hat sich in den SESAM-Spielen mit Studenten gezeigt, dass auch sinnvolle Eingaben der Spieler nicht erkannt wurden, so dass die Spieler nicht unterscheiden können, ob ihre Handlung generell falsch ist oder ob nur die Formulierung falsch ist. Beispielsweise würde der Satz

Please tell Bernd that he should write the specification

nicht erkannt, obwohl er gleichbedeutend wäre mit dem (vom Modell erkannten) Kommando

Ask Bernd to create the specification

Daher wird den Spielern ein Handbuch zur Bedienung des Simulators gegeben, in denen die möglichen Kommandos aufgelistet sind. Dadurch werden die Spieler jedoch wieder stark geführt. Das Handbuch ist so aufgebaut, dass die Kommandos in einer für die Projektdurchführung sinnvollen Reihenfolge geordnet sind, so dass der Spieler schnell erkennen kann, welche Kommandos im aktuellen Simulationszustand sinnvoll sein könnten.

Fortschrittskontrolle Die Kontrolle des Spielfortschritts mittels eines Fortschrittsbuttons gibt dem Spieler zusätzlich Kontrolle über den Ablauf des Projekts. Der Spieler muss selbst entscheiden, wann er das Spiel fortführen will. Ein Konzept, wie man es bei Abenteuerspielen häufig findet, das die Entwicklung der Simulationszeit mittels Erreichen von Zielen umsetzt, wäre für SESAM nicht sinnvoll. Der Spieler kann nicht unmittelbar nach Absetzen eines Kommandos wissen, ob seine Handlungen zum Erfolg führen. Die Realität sieht natürlich keine Möglichkeit vor, die Zeit voranschreiten zu lassen. Daher ist zukünftig vorgesehen, die Simulationszeit automatisch voranschreiten zu lassen, um den Spieler zusätzlich unter Druck zu setzen und ihn dazu zu bringen, seine Entscheidungen schnell zu treffen. Das Konzept mit dem Fortschrittsbutton erlaubt, ein solches Konzept einfach zu realisieren.

Feedback Die Feedbackeigenschaften der Spieleroberfläche und der Basismaschine sind zum Teil wenig hilfreich für die Spieler. Das System wartet beispielsweise auf Eingaben oder hat lange Ladezeiten, ohne dass entsprechende Dialoge angezeigt werden. Die Spieler wissen oft nicht, ob sie einen Fehler gemacht haben, ob das System auf eine Eingabe wartet oder gar abgestürzt ist.

Die Ausgabe der Nachrichten enthält hingegen wieder zu viel Information. In den Textausgaben stehen zum Teil Details, die in einem realen Projekt nicht so offensichtlich zum Vorschein kommen würden. So ist beispielsweise die Information, dass ein Mitarbeiter seine Arbeit gerade fertiggestellt hat, nicht sofort erkennbar, wird in der Nachricht aber sofort angezeigt und gibt dem Spieler damit die Chance, schnell zu reagieren.

Auch die Tatsache, dass man alte Nachrichten einsehen kann, da der Bildschirminhalt nicht geleert wird, sondern mittels Bildlaufleisten einsehbar ist, wird der Wirklichkeit nicht gerecht. Will der Spieler solche Nachrichten lesen, beispielsweise die Aufgabenstellung des Projekts, so muss er in

einem realen Projekt auch selbst dafür Sorge tragen, dass er die Informationen gespeichert hat und verwaltet, so dass er zu einem späteren Zeitpunkt des Projekts darauf zugreifen kann.

Letztendlich betrifft die Wirklichkeitsnähe auch die Hilfsprogramme der Spieleroberfläche. Diese sind durchaus nützlich, sind aber für die Simulation eines Projekts nicht wirklichkeitsnah. Möchte der Spieler solche Programme verwenden, ist dies seine (mit Sicherheit auch sinnvolle) Entscheidung. Die Simulation soll ihm dies aber nicht aufdrängen, indem die Programme in die Oberfläche integriert werden und die Benutzung dadurch nahelegen.

4.3 Neue Ansätze zur Gestaltung der Oberfläche

4.3.1 Graphisches Konzept des GPI

Die neue Spieleroberfläche soll nun im Wesentlichen um eine graphische Ausgabe des Systemzustands erweitert werden. Dazu wird statt der bisherigen Textkonsole ein Bildteil angezeigt, in dem der Zustand visualisiert wird. Da sie hauptsächlich dem Spieler Vorteile bringen soll, wurde sie GPI getauft - **Graphical Player Interface**.

4.3.2 Raumkonzept und Mitarbeiter

Die Oberfläche wird mit dem Raumkonzept um einen wichtigen Punkt erweitert. Die Darstellung wird nicht auf ein Bild beschränkt, vielmehr wird es für die unterschiedlichen Rollen des Projekts und unterschiedliche Ausgaben jeweils eigene Räume geben. So gibt es Räume für den Projektleiter, eine Cafeteria und jeweils Büros für die Mitarbeiter. Der Spieler muss von Raum zu Raum gehen, um seine Kommandos an die entsprechenden Personen zu richten. Ebenso muss er die richtigen Räume aufsuchen, um bestimmte Informationen zu bekommen. Will er wissen, was Entwickler Bernd gerade arbeitet, muss der Spieler sich in dessen Raum begeben und nachschauen oder nachfragen. Ein solcher Raum ist in Bild 4.3 auf der nächsten Seite dargestellt.

Das Raumkonzept stellt einen wichtigen Schritt in Richtung realitätsnahe Umsetzung dar. Der Spieler bekommt einen besseren Bezug zur Projektumgebung und kann die Mitarbeiter durch die Zuordnung zu den Räumen besser unterscheiden. Die reine Textdarstellung hatte den Nachteil, dass die Spieler oft den Überblick darüber verloren hat, was die Mitarbeiter gerade machen und wer überhaupt eingestellt ist. Die Zuordnung zu Räumen hat zwei Vorteile: Erstens wird auf den ersten Blick klar, wer dem Projekt zugeteilt ist und wer nicht (nur eingestellte Mitarbeiter bekommen einen eigenen Raum), was das Verwalten der Angestellten erleichtert. Zweitens helfen die graphischen Darstellungen und Zuordnungen der Räume zu Mitarbeitern dabei, verschiedene Informationen mit den Mitarbeitern zu verknüpfen. Dies kann dazu führen, dass es leichter fällt, sich zu merken, welcher Mitarbeiter welche Aufgaben erledigt. Diese Assoziationen gibt es auch (sogar stärker als in der Simulation) in realen Projekten.

Ebenfalls verstärkt wird dieser Effekt durch die Verwendung unterschiedlicher Graphiken für die Mitarbeiter, so dass man nicht nur unterschiedliche Namen sieht, sondern tatsächlich Individuen, die man auf den ersten Blick unterscheiden kann. Die visuelle Assoziation zwischen Graphik des Mitarbeiters und seinen Fähigkeiten oder Vorlieben ist einfacher zu erreichen als eine Assoziation mit seinem Namen, weil Menschen in der Regel solche Informationen visuell besser verarbeiten können. Ein Beleg dafür sind z.B. die Erfolge der Mnemotechniken, einem Lernkonzept, bei dem komplexe Informationen mit Bildern verknüpft werden, um die Gedächtnisleistung zu erhöhen.



Abbildung 4.3: Screenshot der Spieleroberfläche: Mitarbeiterraum

4.3.3 Darstellung der Nachrichten

Für die Darstellung der Nachrichten werden in der neuen Spieleroberfläche zwei Wege begangen. Zum einen werden die Meldungen nach wie vor in Textform präsentiert. Allerdings werden dazu für jede Meldung kleine Fenster geöffnet. Hat der Spieler die Nachricht gelesen, so kann er den Text mit anderen Programmen mittels Copy&Paste speichern. Verlässt der Spieler den Raum, in dem die Nachricht angezeigt wurde und betritt ihn erst später wieder, so wird nur dann die Nachricht erneut angezeigt, wenn keine neue Nachricht für diesen Raum eingetroffen ist. Ansonsten wird die neue Nachricht angezeigt, die Informationen der alten sind verloren. Der Spieler darf also nicht vergessen, wichtige Informationen selbst zu speichern. In realen Projekten muss er sich darum ebenso kümmern. Die zweite Variante, bestimmte Nachrichten darzustellen, ist rein graphischer Natur. Beispielsweise wird eine Spezifikation als graphisches Element in der Bildszene dargestellt, wenn ein Entwickler gerade damit arbeitet. Ähnlich verhält es sich bei anderen Dokumenten und Nachrichtenarten. Die Darstellungen in Textform und mit Bildern ist kombinierbar und wird in den meisten Fällen auch zusammen angezeigt.

Eine graphische Darstellung hat in vielen Fällen Vorteile. Viele der Vorgänge in einem Software-Projekt können unmittelbar gesehen werden. Arbeitet ein Mitarbeiter gerade, so genügt ein kurzer Blick in sein Büro. Solche Informationen können graphisch einfach dargestellt werden und kommen der Realität näher als eine Beschreibung der Situation mittels Text. Will man hingegen genauere Informationen zu den Vorgängen, Dokumenten oder Mitarbeitern, wird dies in kleinen Berichten zusammengefasst, so dass eine Textdarstellung dieser Informationen auch in der Spieleroberfläche Sinn ergibt. Die Spieleroberfläche soll also Informationen des Projekts dann graphisch darstellen, wenn ein Projektleiter in der realen Situation diese Informationen ebenso auf visuellem Weg erfährt. Andere Informationen, die er beispielsweise durch Berichte bekommt, sollen weiterhin als Textnachrichten ausgegeben werden.

4.3.4 Interaktionsmöglichkeiten

Die Interaktion findet nun auf verschiedene Arten statt. Die bewährte Freitexteingabe wird übernommen. Sie stellt die Haupteingabemöglichkeit der Oberfläche dar. Allerdings wird der Interpreter, der die Eingaben übernimmt, abgeändert, so dass mehr Eingaben verstanden und vom System korrekt umgesetzt werden können. Die Eingabe erfolgt wie bisher mittels eines Eingabefelds, die Bestätigung eines Kommandos wiederum mit *Enter*. Die Kommandos sind nun zusätzlich Räumen zugeordnet. Will der Spieler einem Mitarbeiter zum Beispiel den Auftrag erteilen, die Spezifikation zu erstellen, muss er sich in dessen Raum befinden, um das Kommando einzugeben.

Zusätzlich kann der Spieler jetzt auch Bildelemente der dargestellten Szene in die Eingabe einbinden. So kann er bestimmte Objekte und Personen anklicken. Deren Namen werden daraufhin automatisch in das Eingabefeld eingetragen. Somit muss der Spieler sich nicht mehr darum kümmern, wie beispielsweise ein Entwickler heißt, er kann ihn einfach anklicken und so in das Kommando integrieren. Objekte, die in dieser Form verwendbar sind, werden in der Oberfläche durch eine Veränderung des Mauszeigers kenntlich gemacht, sobald der Spieler die Maus über ein solches Objekt bewegt.

Die dritte Möglichkeit schließlich ist für Anfänger im Umgang mit dem SESAM-System gedacht. Die Spieleroberfläche bietet ein Menü an, welches hierarchisch gegliedert alle Kommandos des QS-Modells anzeigt. Hier kann der Spieler ein Kommando auswählen, das anschließend im Eingabefeld angezeigt wird. Die Gliederungsstufen sollen dem Spieler die Möglichkeit geben, durch kleine Hinweise selbst das richtige Kommando herauszufinden. Beispielsweise wird auf Ebene 1 des Menüs nur das Kommando *Create* angezeigt. Auf der nächsten Ebene werden die Dokumente angezeigt, auf die man *Create* anwenden kann. Wenn der Spieler bereits weiß, welche Aktionen er mit *Create* ausführen kann, muss er somit nicht durch das gesamte Menü gehen, sondern kann den Dialog vorzeitig abbrechen und das Kommando in das Eingabefeld eintragen. Manche Kommandos sind nicht vollständig im Menü verfügbar. So müssen bei einigen Kommandos die Namen der beteiligten Mitarbeiter ergänzt werden. Der Spieler muss auf jeden Fall die Eingabe mit *Enter* abschließen, eine automatische Ausführung nach Drücken des Menüpunktes soll nicht erfolgen. Dadurch hat der Spieler die Chance, das Kommando nochmals genau zu betrachten, um beim nächsten Anwenden ohne Menü auszukommen. Allerdings soll diese einfache Variante der Bedienung nicht kostenlos sein. Wählt der Spieler einen Punkt aus dem Menü aus, so wird automatisch der Fortschrittsbutton aktiviert und die Simulationszeit schreitet einen Schritt voran. In wirklichen Projekten entspricht dieses Nachschauen im Menü einem Nachschauen in Büchern, Webseiten usw. Wer nachlesen muss, was als nächstes zu tun ist, vergeudet damit wertvolle Zeit.

Die Freitexteingabe wurde konzeptionell komplett überarbeitet. Der neue Dolmetscher soll nicht mehr Satzteile erkennen und zuordnen, sondern auf Stichworte reagieren. Bei einer bestimmten Kombination von Stichworten kann das System davon ausgehen, dass eine bestimmte Handlung ausgeführt

werden soll. Ein Beispiel:

```
Ask Bernd to talk with the customer
```

Um den Entwickler Bernd mit dem Kunden sprechen zu lassen, genügt dem System die Angabe der Stichworte *Bernd*, *talk*, *customer*, und zwar auch dann, wenn die Reihenfolge beliebig verdreht ist. Der syntaktisch korrekte Satz

```
Ask the customer to talk with Bernd
```

würde zum selben Ergebnis wie der erste Satz führen. Ob dieses Verhalten des Systems in den Spielen tatsächlich kritisch ist, muss sich in Testspielen zeigen. Ein Nachfragen, ob das vom System erkannte Kommando korrekt sei, ist hierbei eine Kompromisslösung, über die nachgedacht werden kann. Die Stichworterkennung hat den Vorteil, dass wesentlich mehr Sätze auf einfache Weise erkannt werden können als dies bisher der Fall war.

4.3.5 Stärken und Schwächen

Die Stärken und Schwächen der neuen Spieleroberfläche werden sich natürlich erst im Laufe der Zeit bei der Benutzung durch Studenten zeigen. Die folgenden Punkte sind also rein spekulativ und stellen Vermutungen darüber an, was mit der neuen Oberfläche zu erwarten ist. Die graphische Darstellung des Systemzustands hilft, den Spieler mehr in das Projekt einzubinden. Durch die Darstellung kann er sich den Projektzustand und -verlauf besser vorstellen. Damit hat er einen besseren Überblick über die Gesamtsituation, wie er sie auch in einem realen Projekt hat. Durch den verbesserten Interpreter werden mehr Eingabemöglichkeiten für die Modellbefehle gegeben. Damit erhöht sich die Chance, dass eine Eingabe des Spielers, sofern sie sinnvoll ist, auch erkannt und umgesetzt wird. Die Verringerung von Fehlermeldungen bei Nicht-Erkennen eines Kommandos führen dazu, dass der Spieler mehr positive Erlebnisse beim Absetzen der Kommandos erfährt. Ebenso soll die intuitive Bedienung und die allgemein zugänglichere Oberfläche die Motivation des Spielers steigern.

Die neue Oberfläche weist aber auch Gefahren und Nachteile auf. So wird der Spieler wieder stärker in seinem Handeln geleitet. Speziell durch das Kommandomenü werden verschiedene Handlungsmöglichkeiten vorgegeben. Dennoch denke ich, dass ein Spieler, dem keine Hilfe beim Lösen seiner Aufgaben angeboten wird, schnell die Lust an der Simulation verliert, wenn er bei einer Aufgabe nicht weiterkommt und wenig positive Erlebnisse erfahren kann.

Ein ganz anderes Problem ergibt sich für die Entwickler von Modellen. Im Gegensatz zur reinen Textausgabe der bisherigen Spieleroberfläche muss nun ein erheblicher Entwicklungsaufwand für die graphische Umsetzung der Modelldaten geleistet werden. Die Entwicklung wird zwar durch den Entwurf der neuen Spieleroberfläche erleichtert, dennoch wird die Zeit für die Erstellung der Graphiken, Menüs, Räume usw. durchaus erheblich sein. Der Aufwand zur Anpassung der bisherigen Spieleroberfläche an neue Modelle war praktisch nicht vorhanden, so dass nun damit gerechnet werden muss, einen gewissen Aufwand zusätzlich aufzubringen, um die Modelle graphisch umzusetzen.

4.4 Alternative Ansätze

Im Laufe dieser Diplomarbeit wurden zwei alternative Konzepte für die Spieleroberfläche erarbeitet, die mit den anderen Konzepten kombiniert werden können.



Abbildung 4.4: Spieleroberfläche eines typischen Abenteuerspiels

4.4.1 Kontextabhängige Menüs

Zum einen handelt es sich um kontextabhängige Eingabemöglichkeiten, beispielsweise mittels situationsabhängigen Popupmenüs. Dieses Konzept wird bei Oberflächen oft angewendet, wenn sich bestimmte Funktionen nur auf bestimmte Objekte anwenden lassen. Ein bekanntes Beispiel hierfür sind die *kontextsensitiven Menüs* in Dateibrowsern, die Kommandos abhängig vom Dateityp anbieten. Da die Kommandos abhängig von Bildobjekten gesteuert ist, wird diese Art der Eingabe auch *objektorientierte Bedienung* [Bal96] genannt.

4.4.2 Befehlsorientierte Eingabe

Die zweite Möglichkeit für Kommandoingaben in SESAM ergab die Analyse von bestehenden Abenteuerspielen. Diese bieten oft *Buttonleisten* an, mit deren Hilfe man Befehlswords und Objekte verketten kann, um eine Aktion auszuführen. Dieses Prinzip wird häufig bei Abenteuerspielen angewendet, da es eine einfache Möglichkeit darstellt, Befehle einzugeben, ohne dem Spieler die Lösungswege offensichtlich vorzulegen. Ein Beispiel für eine solche Oberfläche ist in Bild 4.4 dargestellt². Für SESAM wäre denkbar gewesen, Befehlswords wie *Create* und *Review* anzubieten, die mit Objekten wie einer *Spezifikation*, welche in der Bildszene graphisch dargestellt werden, kombiniert werden können. Dazu hätte allerdings ein weiterer Bereich in der Bildszene eingerichtet werden müssen, der alle verwendbaren Dokumente und Notationen graphisch dargestellt hätte, um dem Spieler eine Auswahl an Objekten zu liefern, die noch nicht existieren aber erzeugt und verwendet werden können. Eine solche Alternative ist in Bild 4.5 auf der nächsten Seite dargestellt.

²Monkey Island 2, © by LucasArts Entertainment Company LLC



Abbildung 4.5: Alternative Spieleroberfläche mit Befehlsbuttons

4.4.3 Schwächen der alternativen Ansätze

Beide Konzepte haben aber ein grundsätzliches Problem, was schließlich dazu führte, dass die Konzepte verworfen wurden. Sie können dem Spieler nur die Möglichkeiten anbieten, die das Spiel auch verarbeiten kann. Dadurch ist die Auswahl der Einträge eingeschränkt, was Auswirkungen auf den Gebrauch der Konzepte macht. Der Spieler kann, vor allem bei kontextabhängigen Eingaben, leicht erkennen, welches der richtige Weg ist. Er kann also ein Spiel erfolgreich beenden, auch wenn er die Hintergründe seines Handelns nur unzureichend verstanden hat. Damit kann nicht mehr unterschieden werden, ob der Spieler tatsächlich etwas gelernt hat und sein Wissen im Spiel durch Nachdenken eingebracht hat, oder ob er sich situationsabhängig überlegt hat, welches Kommando sinnvoll erscheinen könnte und halb durch Kombinieren, halb durch Zufall richtige Entscheidungen getroffen hat. Ein Lösungsansatz hätte darin bestanden, die Szene komplex zu gestalten und viele Objekte anzubieten, die der Spieler benutzen kann, damit er nicht a priori weiß, welche Objekte für die Entwicklung sinnvoll sind. Diese Umsetzung entspricht in etwa dem Konzept der überladenen Menüs der ersten SESAM-Version. Es sprechen allerdings mehrere Gründe gegen eine solche Oberfläche. Zum einen

werden die Modelle sehr groß und aufwändig zu implementieren, da viele Entitäten modelliert werden müssen. Zum anderen müssten Aktionen für die Objekte angeboten werden, wobei eine reine *Dummyfunktion*, die lediglich Simulationszeit kostet aber sonst keine Wirkung zeigt, nicht realitätsnah erscheint. Es besteht außerdem die Gefahr, dass die Spieler wieder stark von den Menüs und Graphiken in ihrer Handlungsweise geleitet werden, ähnlich wie dies bei den überladenen Menüs der Fall war.

Kapitel 5

Realisierung der Oberfläche

Dieses Kapitel beschreibt, wie die Konzepte in der graphischen Benutzeroberfläche umgesetzt wurden.

5.1 Entwicklungsansätze

Während der Entwicklung wurden einige so genannte *Best Practices* angewendet. Diese Entwicklungsansätze dienen dazu, strukturiert und mit bewährten Methoden vorzugehen, um die Wahrscheinlichkeit von Entwurfsfehlern zu minimieren.

5.1.1 Objektorientierter Ansatz

Da die Aufgabenstellung die Sprache Java für die Implementierung der Oberfläche vorsieht, wurde dem Sprachkonzept entsprechend ein objektorientierter Ansatz gewählt. Objektorientierung ist ein Programmierparadigma, bei dem Dinge in Klassen durch ihre Eigenschaften und Methoden beschrieben werden. Die Sprache muss dabei nicht zwangsläufig als objektorientiert gelten, wie dies bei Java oder Ada der Fall ist. Auch in konventionellen Sprachen ist es möglich, objektorientierte Software zu entwickeln, wie das Beispiel von [Fie91] für die Sprache C zeigt. Allerdings bieten Sprachen, die objektorientiert ausgelegt sind, eine einfachere Möglichkeit, dieses Programmierparadigma umzusetzen.

Java Das Prinzip der Objektorientierung bietet sich bei der Programmiersprache Java an. Java ist auf eine objektorientierte Lösung ausgelegt und bietet entsprechende Werkzeuge an, mit denen eine solche Programmierung besonders einfach gemacht wird.

Natürliche Modellierung Die Spieleroberfläche soll die Simulation einer realen Situation visualisieren. Die objektorientierte Modellierung ist „natürlich“, da reale Dinge in Klassen beschrieben werden können, indem man ihre Eigenschaften und Funktionen beschreibt. So kann man beispielsweise Mitarbeiter durch die Vergabe eines Namens und ihrer Fähigkeiten beschreiben.

Objektorientierte Basismaschine Die Objektorientierung bot sich auch deshalb an, weil die Basismaschine, die die Modelle und Systemzustände verwaltet, ebenfalls nach diesem Prinzip implementiert wurde. Die Verwendung eines gemeinsamen Programmierprinzips ist für Wartungsarbeiten einfacher zu verstehen, da ähnliche Teile, zum Beispiel die Schnittstellen zwischen der Basismaschine und der Spieleroberfläche ähnlich implementiert sind.

Nachteile Es gibt aber nicht nur Vorteile. Der wohl gravierendste Nachteil besteht darin, dass objektorientierte Programme schwieriger als konventionell geschriebene Programme zu testen sind. Der Grund hierfür liegt in der Komplexität der Aufrufbeziehungen der Objekte und deren Zustände. Bei konventionellen Programmen genügt es, eine Funktion zu testen, indem man Äquivalenzklassen für die Eingabe der Parameter findet und damit Testfälle aufstellt und ausführt. Bei Objekten gilt dies für Methoden nicht, obwohl sie nicht viel anders aussehen als konventionelle Funktionen. Jedes Objekt kann unterschiedlichste Zustände annehmen. Das heißt, die Klassenvariablen des Objekts können unterschiedliche Werte haben, so dass die Methoden nicht nur über ihre Parameter getestet werden müssen, sondern zusätzlich auch noch für jeden Zustand des Objekts. Betrachtet man dann noch den Umstand, dass Objekte sich gegenseitig Nachrichten schicken und dass die Aufrufe beliebige Aufrufgraphen erzeugen können, wird schnell klar, dass die Testfälle für OO-Programme wesentlich umfangreicher ausfallen und damit mehr Aufwand verursachen. Das Finden der Testfälle wird durch die schwer überschaubaren Objektbeziehungen ebenfalls erschwert. Daher ist es bei objektorientierter Software unerlässlich, schon während der Spezifikations- und Entwurfsphase gründlich zu arbeiten, um Implementierungsfehler zu vermeiden.

Fazit Die Vorteile der objektorientierten Modellierung überwiegen die Nachteile deutlich, so dass ich mich für diese Art der Modellierung entschieden habe. Eine konventionelle Programmierung wäre alleine durch die Verwendung von Java schon schwieriger geworden als die objektorientierte Umsetzung.

5.1.2 Einfachheit

Da die Wartung nicht in den Bereich der Diplomarbeit fällt, wurde besonders darauf geachtet, dass Entwickler, die das System zukünftig pflegen und erweitern, eine kurze Einarbeitungszeit zum Verständnis des Codes benötigen. Daher wurden keine komplizierten Algorithmen oder Muster verwendet. Es wurde, sofern dies möglich war, die einfachste Lösung für ein Problem verwendet, gemäß des Prinzips „Keep it small and simple“ (KISS).

Bei Lösungen, die nicht einfach gehalten werden konnten, wurde besonders viel Wert auf eine saubere und verständliche Kommentierung gelegt.

5.1.3 Styleguide und Lesbarkeit

Neben der Einfachheit des Codes ist auch das Aussehen und die Kommentierung des Codes für die Wartung von größter Bedeutung. Der Code, d.h. alle Bezeichner, Namen, Kommentare usw. halten sich daher an die Vorgaben eines Styleguides. Dieser wurde von Sun für die Programmiersprache Java herausgegeben (siehe [KND⁺02]). Die Vorgaben betreffen vor allem die Namensvergebung an Bezeichner, Klassen und Methoden. Ferner legen sie fest, wie die Schachtelung vom Code erfolgen soll und wie die Kommentierung auszusehen hat.

5.1.4 Document-View Muster

Während der Entwurfsphase war es wichtig, Strukturen zu verwenden, die zum einen leicht erweiterbar sind, zum anderen schon in anderen Softwareprojekten mit Erfolg eingesetzt wurden, so dass die Vorteile der Erfahrung mit solchen Strukturen für die Spieleroberfläche genutzt werden konnten. Dazu wurden Entwurfsmuster eingesetzt. Sie stellen abstrakte Lösungen für typische Probleme des

Entwurfs dar. Vor allem in der objektorientierten Welt der Programmierung werden Entwurfsmuster genutzt, um Fehler im Entwurf zu vermeiden und damit die Qualität der Software zu erhöhen.

Das Document-View Muster ist ein solches Entwurfsmuster. Die Idee besteht darin, die Daten und deren Darstellung zu entkoppeln. Das Datenmodell, `Document` genannt, ist neben der Datenhaltung auch für das Transformieren der Daten zuständig. Hier werden also alle Daten verändert, gespeichert und geladen. Ein solches Datenmodell kann nun mehrere Darstellungen der Daten besitzen, `Views` genannt. Je nach Anwendungszweck können beliebig viele Ansichten für ein `Document` erzeugt werden. Diese Ansichten sind für die Darstellung der Daten sowie für Eingaben des Anwenders zuständig. Die Ansicht selbst hält keine Informationen zu den Daten, lediglich eine Referenz auf das `Document`, über die es bei Bedarf die aktuellen Werte der Daten erhalten kann. Bei der Ausführung eines Programmes, das mit dem Document-View Muster arbeitet, sieht der Informationsfluss wie folgt aus. Das Datenmodell ändert seine Daten und schickt an alle Ansichten eine Nachricht, dass sie sich neu darstellen sollen. Die Ansichten holen sich die aktuellen Daten beim `Document` ab und stellen sie entsprechend dar. Dabei ist es unerheblich, ob die Daten durch ein Kommando des Anwenders geändert wurden, oder ob die Änderung auf anderem Wege erfolgt ist. Das Beispiel in [Abbildung 5.1](#) auf der nächsten Seite zeigt ein typisches Beispiel für ein Document-View Muster. Die Klasse `Potato`, die die Eigenschaften einer Kartoffel beschreiben soll, kann auf zwei Weisen dargestellt werden. Einmal durch eine graphische Repräsentation, d.h. die Kartoffel soll am Bildschirm gezeichnet werden, das andere Mal sollen die Informationen über die Kartoffel in einer Tabelle eingetragen werden. Die graphische Darstellung wird von der Klasse `PotatoGraphic` übernommen, die Tabellendarstellung durch die Klasse `PotatoTable`. In unserem Beispiel wird aus Gründen der Einfachheit lediglich die Größe der Kartoffel betrachtet. Dieses Attribut ist `private` deklariert, um ungewollte Zugriffe und Veränderungen auszuschließen.

Die Funktionen

```
letGrow
```

und

```
getSize
```

sind die zugehörigen Schnittstellenfunktionen. Erstere dient dazu, die Kartoffel wachsen zu lassen, also den Wert ihrer Größe zu erhöhen. Die zweite Funktion gibt den aktuellen Wert der Größe zurück.

Wird die Größe der Kartoffel mittels `letGrow` angepasst, ruft die `Potato` Klasse die `Update` methode jeder ihr zugeordneten `View` auf. Dazu muss das `Document` natürlich wissen, welche `Views` ihr zugeordnet sind. Bei der Instanzierung registriert sich eine `View` deshalb bei *ihrem* `Document`, welches alle `Views` in einer Liste hält. Um das Schaubild nicht unnötig kompliziert zu gestalten, wurde auf dieses Detail jedoch verzichtet. Die `Views` sind, wie erwähnt, in eigenen Klassen implementiert. Die `Update` methoden dieser Klassen werden nun vom `Document` aufgerufen. Da die `Views` selbst keine Daten über die Kartoffel halten, müssen sie diese erst beim `Document` mittels

```
getSize
```

erfragen. Die `Views` erhalten die Objektreferenz auf das `Document` im Normalfall bei der Instanzierung, was im Schaubild ebenfalls aus Gründen der Übersichtlichkeit nicht dargestellt ist. Haben die `Views` die Daten erhalten, in diesem Fall die Größe der Kartoffel, können sie diese darstellen, jede auf ihre Weise. Die `Views` können auch Anwenderbefehle erhalten. Beispielsweise könnte der Anwender in die Tabellenansicht gehen und dort auf einen Button drücken, der die Kartoffel wachsen lassen soll. Die `View` würde dann die Methode

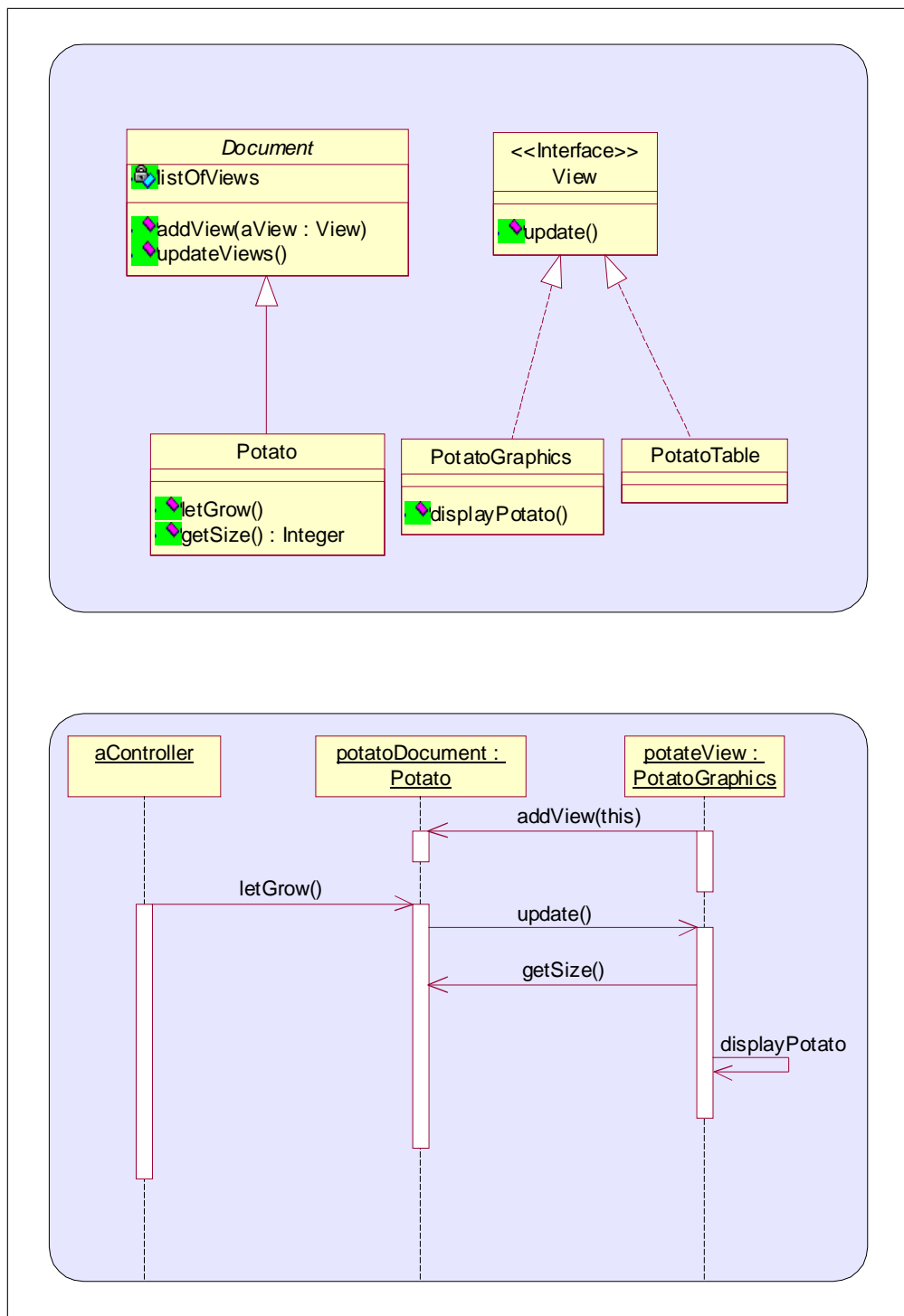


Abbildung 5.1: Das Document-View Entwurfsmuster

`letGrow`

des Documents aufrufen. Die Größe wird geändert, die Views bekommen wieder ein Update und

stellen die geänderten Werte der Kartoffel entsprechend dar. Eine Änderung der Daten im Document bewirkt somit immer eine neue Darstellung in den zugeordneten Views. Dabei spielt es keine Rolle, ob die Daten durch die Views selbst oder durch andere Aufrufe geändert wurden.

Man kann somit beliebig viele Views erstellen, die den Zustand des Documents konsistent abbilden und automatisch auf eine Änderung der Daten reagieren. Dabei können je nach Zweck auch nur Teile der Daten dargestellt werden. Das Document-View Muster ähnelt dem bekannteren Model-View-Controller Konzept, mit dem Unterschied, dass bei dem ersten Muster die Views auch die Controller für die Eingabeverarbeitung beherbergen. Im Falle dieser Diplomarbeit bot sich das Document-View Muster an, da die Controller einfach und klein gehalten und somit leicht zu den Views hinzugefügt werden konnten. Das Model-View-Controller Muster ist detailliert in [GHJV94] beschrieben.

Ein Beispiel für den Vorteil, den das Document-View Muster bietet, findet sich in der neuen Spieleroberfläche im Plug-in für das Qualitätssicherungsmodell. Dort gibt es zwei Views, welche die Räume der Mitarbeiter einmal direkt und einmal indirekt repräsentieren. Eine View stellt eine Liste der Räume dar, die andere stellt die Mitarbeiter in der Cafeteria dar, die noch keinen eigenen Raum haben und auf Anstellung warten. Wird nun in einer der Views das Kommando gegeben, einen Mitarbeiter einzustellen oder zu entlassen, stellen sich beide Views neu dar. Wird also ein Mitarbeiter angestellt, so wird sein Raum in die Liste eingetragen und der Mitarbeiter aus der Cafeteria entfernt. Durch das Document-View Muster könnten noch beliebig viele Views dieser Daten erzeugt werden, ohne dass das Datenmodell oder die anderen Darstellungen geändert werden müssten, was beispielsweise für Zusatzfenster für Tutoren etc. interessant ist.

5.1.5 Plug-in Konzept

Die Oberfläche für SESAM besteht aus zwei Teilen: der Oberfläche selbst, die das Fenster und Optionen bereitstellt, die für alle Modelle von SESAM gelten sollen, und die Darstellung des eigentlichen SESAM-Modells. Dementsprechend werden für die Modelle Plug-ins erstellt, d.h. kleine Programme, die über eine definierte Schnittstelle in die Spieleroberfläche integriert werden können. Diese Plug-ins können beliebig ausgetauscht werden, so dass der Entwicklungsaufwand für neue Modellvarianten gering gehalten wird.

5.1.6 Auslagerung spielweiter Optionen

Graphische Benutzeroberflächen sind nicht für jeden Anwender gleichermaßen geeignet. Deshalb sind Oberflächen im Normalfall konfigurierbar. Ein wichtiger Aspekt ist beispielsweise die Sprache der Menüs oder des Spiels selbst. Werden Beschriftungen, Ausgaben usw. fest im Code integriert, ist die Anpassung von Sprachvarianten schwierig. Daher wird in der Implementierung der Diplomarbeit darauf geachtet, dass alle Einstellungen dynamisch aus einer Datei gelesen und in diese auch zurück geschrieben werden können, so dass ein Austausch der Optionen, zum Beispiel die Umstellung der Menüsprache von englisch auf deutsch, schnell durch Austausch der entsprechenden Konfigurationsdatei erfolgen kann.

Die Konfigurationsdateien werden in einem Verzeichnis im Homeverzeichnis des Spielers angelegt. Dadurch ist gewährleistet, dass das Programm auf den meisten Betriebssystemen lauffähig ist und nicht versucht, in geschützte Bereiche des Datei-Systems zu schreiben. Die Methoden für das Lesen und Schreiben der Konfigurationsdateien werden von Klassen der Java-Standardbibliotheken geliefert.

5.1.7 Versionskontrolle

Während der Diplomarbeit wurden sämtliche Dokumente mittels des Versionskontrollsystems CVS verwaltet. Dieses Vorgehen hatte mehrere Gründe. Der für mich wichtigste Grund war die Sicherheit. Alle Dokumente waren stets auf mindestens zwei Rechnern gesichert. Somit war gewährleistet, dass bei einem Ausfall einer Festplatte oder Ähnlichem keine Daten verloren gehen. Zudem waren alle Dokumente immer konsistent gespeichert und der Zugriff auf alte Versionen gewährleistet. Die Versionskontrolle umfasste nicht nur den Code der Oberfläche, sondern bezog sich auf alle Dokumente gleichermaßen. Der Code und dazugehörige Dokumente können dadurch auch leicht in die bestehende Versionsverwaltung von SESAM eingepflegt werden und bieten die Möglichkeit, auf alte Versionen zuzugreifen, um Änderungen zurückverfolgen zu können.

5.1.8 Verwendung einer IDE

Um möglichst viel Unterstützung bei der Implementierung zu bekommen, wurde eine IDE (integrierte Entwicklungsumgebung) eingesetzt, die gegenüber herkömmlichen Editoren Vorteile bei der Codierung hat. Neben Syntaxhighlighting ist vor allem die Syntaxergänzung hilfreich, da ein langwieriges Suchen von Methoden in Klassen damit entfällt. Für diese Diplomarbeit wurde Eclipse von IBM eingesetzt.

5.2 Spezifikation

Die Spezifikation enthält Informationen über die Spieleroberfläche, die unabhängig von der Sprache der Implementierung die Umsetzung der Anforderungen in Software beschreibt. Dieser Abschnitt soll die wichtigen Teile der Spezifikation natürlichsprachlich zusammenfassen. Die gesamte Spezifikation der Spieleroberfläche wurde während der Diplomarbeit in teils formaler Form (mit UML) geschrieben und der Software bei der Auslieferung an den Betreuer beigelegt. Die formale Beschreibung ist für die Entwicklung und die Wartung der Software unerlässlich. Zum Verständnis der Anforderungsbeschreibung, der Beschreibung der Oberfläche und den Ein- und Ausgabevarianten ist sie jedoch nicht von großer Bedeutung und wird deshalb an dieser Stelle nicht weiter beschrieben.

5.2.1 Informelle Beschreibung der Anforderungen

Die Spieleroberfläche für SESAM hat folgende Bestandteile, die in Bild 5.2 auf der nächsten Seite zu sehen sind:

Bildelemente des Spieleroberflächenrahmens

- **Menüleiste**

Die Menüleiste umfasst die Punkte *File*, *Options*, *Model* sowie *Help*. In dem Untermenü *File* findet man die Einträge *Load* und *Exit*. *Load* bietet einen Auswahldialog zum Laden von Modellen an, *Exit* beendet das Programm. Unter *Options* kann man einstellen, ob man Tipps beim Start des Programms haben möchte. Daneben kann der Spieler einen Namen angeben, mit dem er im Spiel angesprochen werden will. Der Menüpunkt *Model* ist beim Start der Oberfläche zunächst leer. Je nach Plug-in werden hier für das Modell wichtige Optionen und Funktionen dargestellt, sobald das Plug-in geladen wurde. Der Menüpunkt *Help* enthält schließlich Untermenüpunkte zur Darstellung eines About-Dialogs sowie einer kleinen Online-Hilfe zur Benutzung der Spieleroberfläche.



Abbildung 5.2: Screenshot der Spieleroberfläche: Cafeteria

- **Iconmenüleiste**

Die Iconmenüleiste enthält Icons zur schnelleren Verwendung häufig verwendeter Funktionen. Darunter fällt das Laden eines Modells, das Einstellen spielweiter Optionen, das Anzeigen einer Online-Hilfe und der About-Box. Funktionen, die für das Modell von Bedeutung sind, beispielsweise Kommandos, sollen hier bewusst nicht dargestellt werden, um den Spieler nicht zu lenken.

- **Bildszene**

In diesem Bereich stellen die Modelle ihren Zustand graphisch dar. Verschiedene Bildelemente sind in dieser Szene unter Umständen auch interaktiv zu verwenden.

Bildelemente des Plug-ins für das Qualitätssicherungsmodell

- **Bildszene des QS-Modells**

Das QS-Modell-Plug-in kann drei verschiedene Raumtypen anzeigen. Diese umfassen die Cafeteria (siehe Bild 5.2), den Projektleiterrraum (siehe Bild 5.3 auf Seite 49) und die Mitarbei-

terräume (siehe Bild 4.3 auf Seite 34). Bildelemente in diesen Räumen, beispielsweise Mitarbeiter oder andere graphisch dargestellte Entitäten des Modells, können durch Anklicken mit der Maus in das aktuelle Kommando integriert werden. Dazu wird der Name der Entität jeweils an das im Eingabefeld vorhandene Kommando angehängt.

- **Plug-in-Menü**

Das Plug-in-Menü für das QS-Modell enthält Funktionen zum Laden und Speichern eines Modellzustands, zum Ein- und Ausschalten von Tutor- und Debugnachrichten sowie hierarchisch gegliedert die Kommandos des Modells.

Wird das Menü zur Kommandoeingabe verwendet, werden die ausgewählten Kommandos im Eingabefeld dargestellt. Zusätzlich wird noch ein Simulationsschritt im Spiel ausgeführt. Der Spieler wird also für die Verwendung des Menüs „bestraft“. Um einen besseren Lernerfolg zu erzielen, muss er das Kommando aber dennoch beispielsweise um die Namen der Mitarbeiter ergänzen und mit *Enter* bestätigen. Auf diese Weise muss er sich mit dem Kommando auseinandersetzen und bekommt ein Ergebnis nicht nur durch Anklicken eines Menüpunkts.

- **Eingabefeld**

Das Eingabefeld entspricht dem Eingabefeld der bisherigen Spieleroberfläche. Hier können Kommandos an das SESAM-Modell eingetragen werden. Diese müssen immer mit *Enter* abgeschlossen werden. Die Spielereingaben werden dann vom Interpreter übersetzt und an die Basismaschine weitergeleitet. Ist ein korrekt formuliertes Kommando in einem bestimmten Raum nicht durchführbar, erscheint ein Hinweis der Spieleroberfläche, in welchem Raumtyp dieses Kommando Sinn ergeben könnte.

- **Navigationsfeld**

Das Navigationsfeld ist ebenfalls eine Neuerung der Spieleroberfläche. Hiermit kann man durch die verschiedenen Räume navigieren. Zur Zeit der Diplomarbeit gibt es für das QS-Modell die Räume Cafeteria, Projektleiterräum und jeweils einen Raum für die Mitarbeiter. Um in einen Raum zu gelangen, wählt man im Dropdownmenü einen Raumnamen aus. Bei Spielbeginn ist zunächst nur der Projektleiterräum und die Cafeteria sichtbar. Erst nach der Einstellung von Mitarbeitern kann man auch deren Räume besuchen. Die Räume sind direkt den Mitarbeitern zugeordnet. Wird ein Mitarbeiter entlassen, so macht es für den Projektleiter keinen Sinn mehr, in dessen Raum nachzuschauen. Der Raum wird daher aus der Auswahlliste entfernt.

- **Fortschrittsfeld und Fortschrittsbutton**

Diese beiden Elemente dienen dazu, die Simulationszeit voranschreiten zu lassen. Dabei wird beim Druck auf den Proceedbutton jeweils die Anzahl Simulationsschritte vorwärtsgegangen, die im Fortschrittsfeld eingetragen ist.

Während der Ausführung eines Spiels ändert sich vor allem die Bildszene abhängig vom Zustand des QS-Modells. Die Ausgabe der Spielernachrichten geschieht in den jeweiligen Räumen durch Anzeigen von kleinen Nachrichtenfenstern und graphischen Elementen, sofern dies möglich und sinnvoll ist (z.B. das Anzeigen einer Spezifikation).

5.2.2 Nichtfunktionale Anforderungen

Dieser Abschnitt stellt alle Anforderungen zusammen, die qualitative Aussagen zu den Anforderungen machen, die keine Funktion im Sinne der Bedienung haben. Die Aufgabenstellung hat wenige

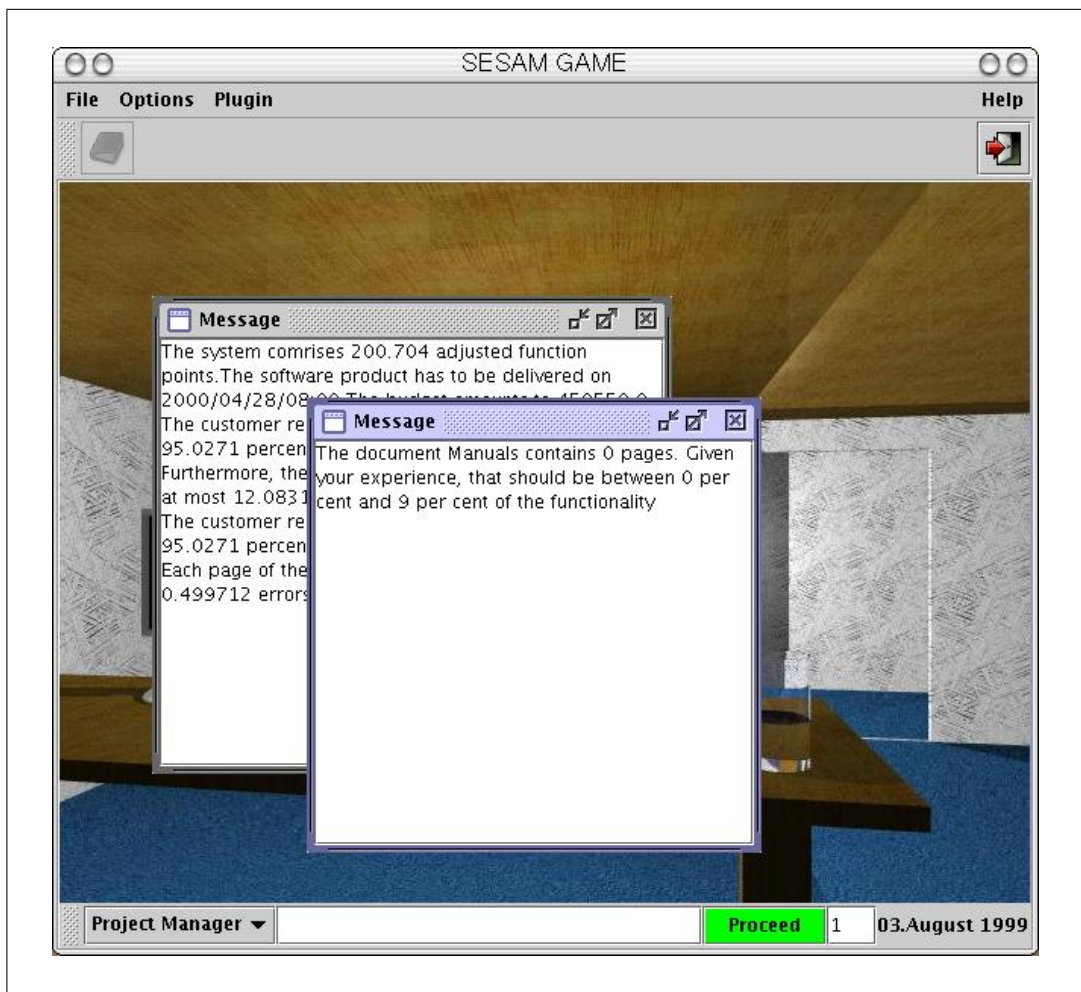


Abbildung 5.3: Screenshot der Spieleroberfläche: Nachrichtenfenster im Projektleiterräum

Vorgaben gemacht, so dass sich die meisten Angaben an den Merkmalen gängiger und ähnlich spezifizierter Systeme orientieren.

- **Antwortzeiten**

Die Antwortzeiten ergeben sich durch die Geschwindigkeit des Modells, der Basismaschine und der Oberflächen-Beschreibungssprache Swing. Eine genaue Auflistung der Antwortzeiten ist nicht vorgesehen, sie sollen aber zumindest so gering sein, dass sich keine störenden Wartezeiten ergeben. Welche Zeiten als störend empfunden werden, muss in Versuchsläufen durch Testspieler herausgefunden werden. Bei den durchgeführten Testspielen am Ende dieser Diplomarbeit wurden die Antwortzeiten nicht kritisiert.

- **Speicherverbrauch**

Der Speicherverbrauch soll so gering sein, dass keine Beeinträchtigung des Spiels wahrnehmbar ist.

Als Referenzsystem für die Überprüfung der oben genannten Anforderungen wird der Rechner Genf der Abteilung Software Engineering der Universität Stuttgart verwendet.

Mengengerüst

Maximal ein Spieler soll die Spieleroberfläche gleichzeitig bedienen können. Ein Mehrbenutzerbetrieb ist nicht erforderlich. Das System besitzt jeweils eine Datei zur Speicherung der Optionen für die Spieleroberfläche und für jedes Plug-in. Spezielle Optionen (beispielsweise Spielegraphik) können in weiteren Dateien gespeichert sein.

Eingabebeschränkungen

Für die Eingabe eines Benutzerkommandos sind alle Buchstaben und Zahlen einer deutschen Tastaturbelegung gültig. Groß- und Kleinschreibung wird bei der Interpretation der Kommandos nicht beachtet. Bei der Eingabe im Fortschrittsfeld sollen mindestens alle Zahlen von 1 bis 999 gültig sein.

5.3 Entwurf

Dieses Kapitel fasst die Ergebnisse der Entwurfsphase zusammen. Während der Entwurfsphase werden wichtige Entscheidungen getroffen, die die Umsetzung der Aufgabe in der Implementierungsphase betreffen. Hierbei spielen vor allem Sprachkonzepte der verwendeten Programmiersprache eine tragende Rolle. Die folgenden Abschnitte sollen erläutern, welche Bausteine die Spieleroberfläche verwendet, wie diese implementiert sind und wie sie untereinander zusammenhängen.

5.3.1 Bisheriges SESAM-System

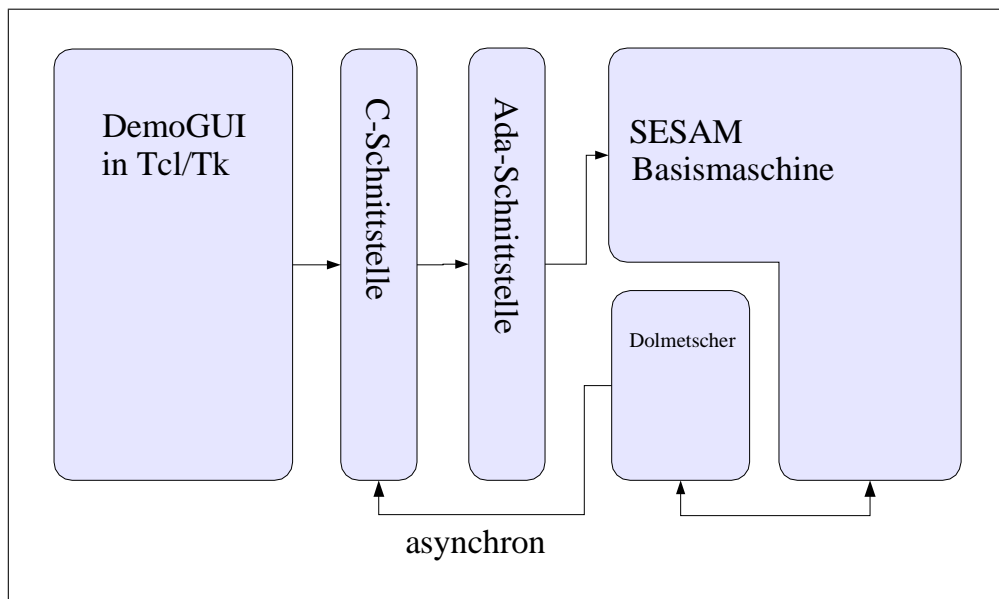


Abbildung 5.4: Architektur von SESAM heute

Die bisherige Spieleroberfläche kann in die folgenden Teilsysteme untergliedert werden, die auch in Abbildung 5.4 zu erkennen sind:

- **SESAM Basismaschine**

- **Spieleroberfläche** (DemoGUI)
- **Schnittstelle zwischen Oberfläche und Basismaschine**
- **Dolmetscher**

Um den Zusammenhang dieser Teilsysteme besser zu verstehen, soll nun ein kleines Beispiel eines Aufrufes durchgespielt werden. Hat der Spieler das Modell gestartet, kann er in der Spieleroberfläche ein Kommando eingeben. Die Spieleroberfläche bietet ihm hierfür ein Eingabefeld an. Sie stellt weiterhin die Ausgaben des Systems dar. Dieses Kommando wird nun an die Basismaschine gesendet. Da die Spieleroberfläche in Tcl/Tk programmiert wurde, die Basismaschine jedoch in Ada, muss das Kommando zunächst so übersetzt werden, dass die Basismaschine dieses versteht. Die Schnittstellenfunktionen, die die Basismaschine bereitstellt, müssen also der Spieleroberfläche bekannt gemacht werden. Dies geschieht mittels einer C-Schnittstelle, da beide Sprachen, Tcl/Tk und Ada, das Ansprechen von C-Funktionen direkt unterstützen. Die Spieleroberfläche ruft daher eine Funktion auf, die für sie wie eine Tcl/Tk Funktion aussieht, in Wahrheit aber eine C-Funktion ist, die wiederum eine Ada-Funktion aufruft. Die Basismaschine nimmt das Kommando auf und verarbeitet es. Dazu werden Regeln im Modell ausgeführt, die den Zustand des Modells ändern. Am Ende gibt die Basismaschine als Rückgabewert eine Reihe von Nachrichten aus, die die Spieleroberfläche darstellt. Die Aufrufe sind zu diesem Zeitpunkt nur in einer Richtung möglich. Die Spieleroberfläche kann Funktionen der Basismaschine aufrufen, jedoch nicht umgekehrt.

Allerdings werden die Kommandos nicht sofort von der Basismaschine interpretiert. Da die Spielereingaben in Form von Freitext eingegeben werden, muss man sie erst in Kommandos übersetzen, die die Basismaschine verstehen kann. Dazu wird die Spielereingabe an den Dolmetscher übergeben, der sie übersetzt. Ist das Kommando nicht eindeutig oder falsch, kann der Dolmetscher die Spieleroberfläche veranlassen, beim Spieler nachzufragen, was mit dem Kommando gemeint war. Dies geschieht jedoch nicht über den Rückgabewert des Originalaufrufs der Basismaschine, sondern durch einen asynchronen Aufruf der Spieleroberfläche vom Dolmetscher selbst. Asynchron bedeutet, dass der Dolmetscher zu einem beliebigen Zeitpunkt beim Spieler an der Spieleroberfläche nachfragen kann. Er kann also Methoden der Spieleroberfläche aufrufen, ebenso wie die Spieleroberfläche die Basismaschinenfunktionen verwenden kann. Dieser Umstand machte es notwendig, die Architektur des Systems grundlegender zu ändern als ursprünglich geplant. Java könnte solche Callbacks zwar ebenfalls realisieren, allerdings ging ich davon aus, dass der Aufruf einer Java-Funktion an dieser Stelle unnötig ist. Die Callbacks werden nur in dem Fall ausgeführt, wenn ein Kommando nicht erkannt wurde, niemals zu einem „beliebigen“ Zeitpunkt. Daher könnte man die Nachfragen auch in Form von Rückgabewerten eines Funktionsaufrufs von Seiten der Spieleroberfläche realisieren und hätte somit eine bessere Trennung zwischen der Spieleroberfläche und der Basismaschine. Die Anpassung des Dolmetschers wurde allerdings nicht verfolgt, da auch die Satzerkennung mittels Stichworten gefordert wurde. Daher wurde entschieden, einen eigenständigen Interpreter zu implementieren, der den Dolmetscher vollständig ersetzt.

5.3.2 SESAM-System mit neuer Spieleroberfläche

Das neue SESAM-System hat die in Bild 5.5 auf der nächsten Seite dargestellte Architektur. Der prinzipielle Aufbau ist ähnlich dem des Originalsystems. Die GPI ruft Methoden der Basismaschine auf. Dies geschieht über eine Schnittstelle, die in C implementiert ist, welche Funktionen einer in Ada programmierten Schnittstelle aufruft. Diese Schnittstelle ruft letztlich die Funktionen der Basismaschine auf. Anders als in der bisherigen Umsetzung werden die Funktionen der C-Schnittstelle jedoch nicht

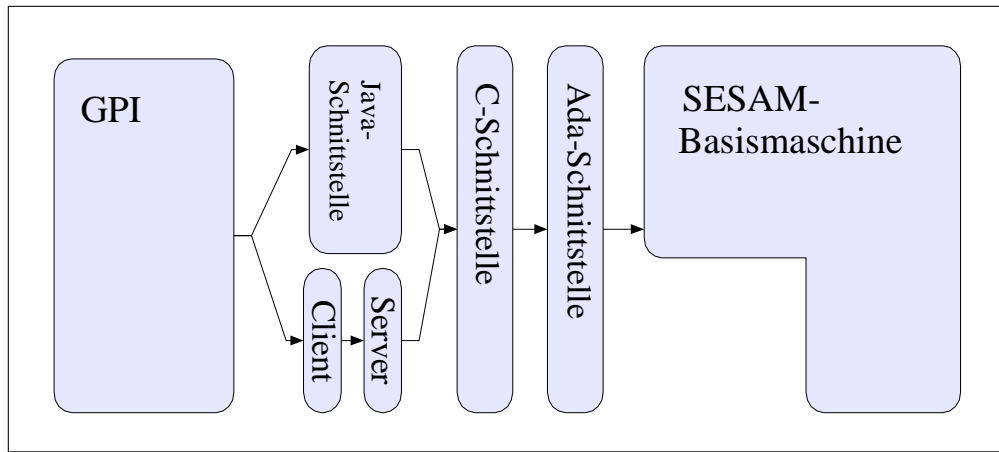


Abbildung 5.5: Neue Architektur von SESAM

direkt aus der GPI aufgerufen, sondern über eine vorgelagerte Java-Schnittstelle, die mittels des Java Native Interface (JNI) eine Verbindung zum C-Teil herstellt. Vorteil dieses Entwurfs ist eine beliebige Auswechselbarkeit der Spieleroberfläche und damit eine offenere Implementierung. Zudem kann man die Java-Schnittstelle beliebig ersetzen, solange die Schnittstellenfunktionen der Spezifikation entsprechen. So gibt es in der fertiggestellten Version dieser Diplomarbeit bereits zwei verschiedene Schnittstellen. Die erste Version bindet die Basismaschine direkt in das Programm ein, während die zweite Version eine Client-Server Beziehung öffnet, um das SESAM-Spiel verteilt ausführen zu können. Hier können Basismaschine und Spieleroberfläche auf verschiedenen Rechnern ausgeführt werden. Die Callback-Funktionalität wurde herausgenommen, indem der Dolmetscher durch einen eigenen Übersetzer ersetzt wurde. Alternativ kann man den Dolmetscher so abändern, dass keine Callback-Funktionen aufgerufen werden und diesen weiter verwenden.

5.3.3 Modellunabhängige Teile der Spieleroberfläche

Dieser Abschnitt soll den Entwurf der Spieleroberfläche verdeutlichen und die Einbettung in das Gesamtsystem erläutern. Wie bereits beschrieben, besteht das GPI aus zwei Teilen, dem Rahmen der Spieleroberfläche und den Modell-Plug-ins für die Hochsprachenmodelle. Der Rahmen stellt alle modellunabhängigen Teile der Oberfläche dar. Dazu gehören:

- die Darstellung des Fensters selbst
- eine Menüleiste
- eine Iconmenüleiste
- ein Bereich für die Bildszene

Die Funktionen dieser Teile beschränken sich auf das Starten und Beenden der Anwendung, die Darstellung allgemeiner Hilfen und About-Boxen sowie spielweite Optionen wie Spielernamen oder „Tipp des Tages“-Anzeige. Wie in Bild 5.6 auf der nächsten Seite zu erkennen ist, lädt der Rahmen der Spieleroberfläche Informationen aus Konfigurationsdateien, die zum heutigen Zeitpunkt die Sprache der Menüs festlegen. Für jede Sprache gibt es hierbei eine eigene Konfigurationsdatei, so dass ein

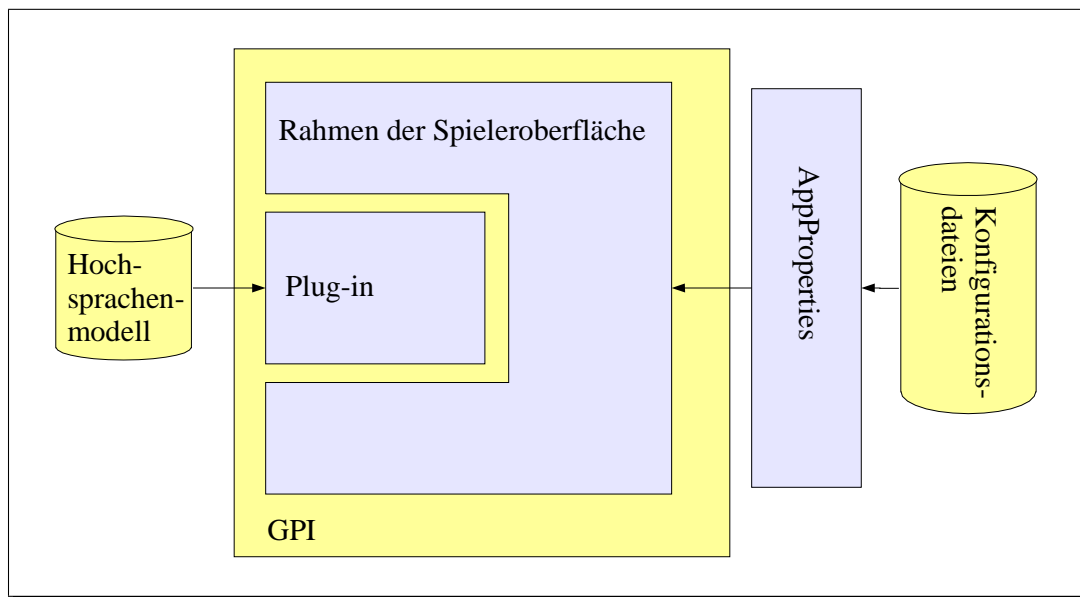


Abbildung 5.6: Plug-in Konzept der GPI

Austausch der Datei die Umstellung der Sprache einfach und schnell ermöglicht. Alle Konfigurationsdateien werden in einem Verzeichnis `.sesam` im Home-Verzeichnis des Spielers angelegt. Dadurch ist gewährleistet, dass jeder Spieler seine eigenen Konfigurationsdateien verwenden kann, selbst wenn das Programm in einem globalen Verzeichnis installiert wird. Dies funktioniert selbstverständlich nur auf Systemen, die mehrere Benutzer zulassen. Der Zugriff auf die Optionen ist in einer separaten Klasse (`AppProperties`) gekapselt, damit die Implementierung auch hier für Erweiterungen möglichst offen ist. Weitere Optionen, wie z.B. das Look&Feel der Applikation, können auf ähnliche Weise leicht in die Anwendung eingebaut werden. Die Konfigurationsdateien sind als Liste von name-value Paaren aufgebaut und können entweder mit einem Texteditor oder alternativ von der Anwendung selbst erzeugt werden. Die Spieleroberfläche stellt nach dem Start das Fenster der Anwendung dar, baut jedoch noch keine Verbindung zur Basismaschine auf. Erst beim Laden eines Modells geschieht dieser Schritt, der in den folgenden Kapiteln näher beschrieben wird.

5.3.4 Plug-ins

Die Aufgabe der Plug-ins ist das Darstellen des Modellzustands der Basismaschine. Der Modellzustand wird dem Spieler mittels Nachrichten der Basismaschine, genauer den Spielernachrichten, mitgeteilt. Daher muss ein Plug-in diese Nachrichten visualisieren können. Weiterhin muss der Spieler auch die Möglichkeit haben, Kommandos an die Basismaschine absenden zu können. Die Kommandos können, genau wie bei der bisherigen Spieleroberfläche, in eingeschränkter natürlicher Sprache eingegeben werden. Diese Kommandos müssen daher zunächst in ein Kommando umgewandelt werden, das vom Modell interpretiert werden kann. Hier gibt es zwei mögliche Ansätze zur Realisierung:

- Die Spielereingabe wird von der Spieleroberfläche interpretiert und vor der Übertragung an die Basismaschine übersetzt.
- Die Spielereingabe wird an die Basismaschine übergeben. Die Basismaschine kümmert sich

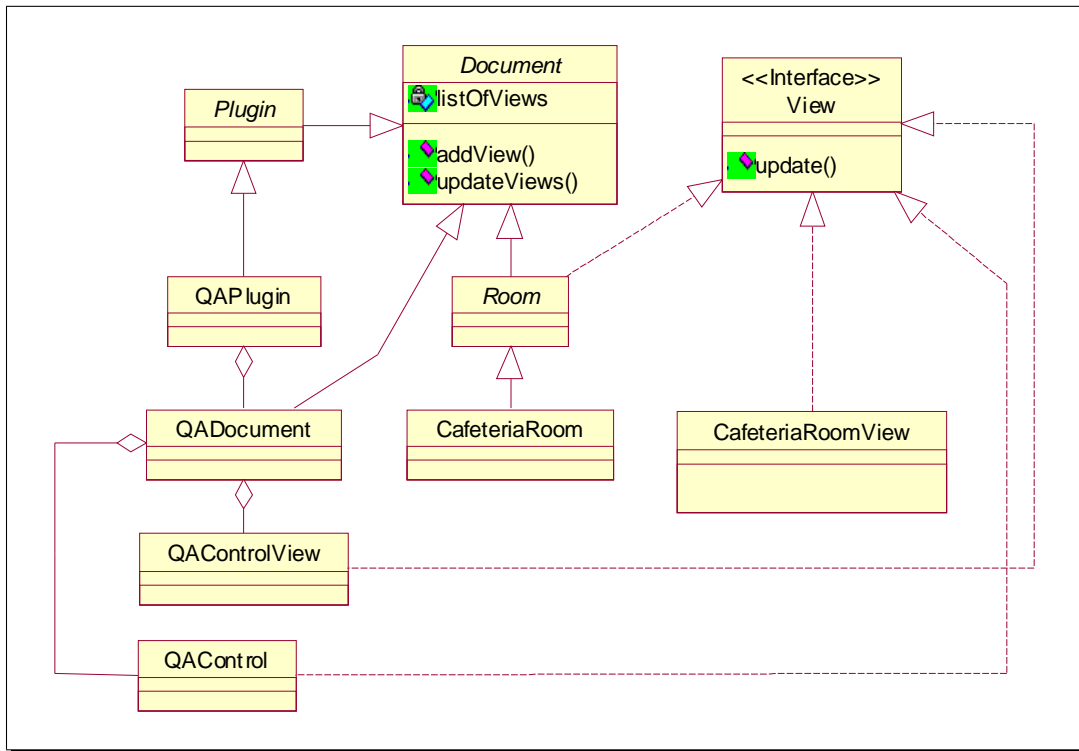


Abbildung 5.7: Document-View Muster der Plug-ins

selbst um die korrekte Übersetzung (z.B. mit einem Dolmetscher-Modul).

Für die Diplomarbeit wurde die erste Methode verwendet. Zwar besitzt die Basismaschine einen Dolmetscher, der diese Aufgabe übernehmen könnte, er verwendet jedoch die bereits beschriebenen Callback-Aufrufe, so dass er nicht ohne Abänderung verwendet werden kann. Die Neuimplementierung verfolgt mit der Stichworterkennung auch einen anderen Ansatz zur Interpretation der Kommandos, so dass der Dolmetscher nicht weiter verwendet werden konnte.

Die Plug-ins sind nun so realisiert, dass die Erweiterung oder Neuimplementierung der Plug-ins möglichst einfach durchzuführen ist. Dazu wurde eine allgemeine, abstrakte Plug-in Klasse eingeführt, deren Aufgabe es ist, eine Verbindung zur Basismaschine herzustellen und das entsprechende Modell zu laden, mit dem es kommunizieren kann. Die Modell-Plug-ins müssen von dieser Klasse ableiten, um die dort definierten Schnittstellen bereitzustellen. Eine Übersicht zu dieser Architektur gibt Bild 5.7. Zudem wird eine Methode als Schnittstelle zugewiesen, die die nötigen Informationen zur Einbettung des Plug-ins in die Oberfläche gibt. Die Plug-ins werden in Abschnitt 5.5 auf Seite 60 detailliert beschrieben, so dass Entwickler, die sich mit der Wartung der Spieleroberfläche befassen, einen guten Überblick über die Teilfunktionen der Plug-ins bekommen.

In Anhang A auf Seite 91 wird ein Gesamtüberblick über die einzelnen Klassen und ihre Abhängigkeiten gegeben. Diese Schaubilder enthalten sämtliche Klassen der Spieleroberfläche, deren Methoden und Schnittstellen.

5.4 Implementierung

5.4.1 Schnittstelle zur Basismaschine

Die Implementierung war zweigeteilt. Der erste Teil der Implementierung befasste sich ausschließlich mit der Umsetzung der Anbindung eines Oberflächen-Prototyps an die vorhandene Basismaschine. Die Schwierigkeit bestand darin, die Kommunikation von Java über C an Ada herzustellen.

1. Zunächst wurde eine Schnittstelle in Ada implementiert, die die Funktionen der Modellschnittstelle sowie der Zustandsprotokolle ansprechen konnte.
2. Danach wurde eine Schnittstelle in C implementiert, die die Ada-Funktionen aufrufen konnte. Um dies zu realisieren, wurden die Funktionen der Ada-Schnittstelle speziell deklariert und mit dem Compiler zu einem Shared-Object übersetzt, welches das C-Modul einlesen und verwenden konnte.
3. Der nächste Schritt bestand darin, eine Java-Klasse zu implementieren, die Zugriff auf C-Funktionen hat. Diese C-Funktionen waren anfangs einfach aufgebaut, um das Prinzip der so genannten Native-Aufrufe kennenzulernen, und wurden sukzessive verfeinert und mit verschiedenen Parametertypen versehen.
4. Als letzter Schritt der Anbindung wurde schließlich die Kommunikation der Java-Klasse mit den C-Funktionen der Schnittstelle zu Ada hergestellt. Dazu wurden die Ada-Funktionen und das C-Modul zusammen in ein Shared-Object kompiliert welches von der Java-Klasse eingelesen werden konnte. Damit kann der Java-Teil auf die Funktionen des Ada-Teils zugreifen.

Für jede Schnittstelle wurden jeweils mehrere Prototypen angefertigt, welche die generelle Vorgehensweise bei der Anbindung an die verschiedenen Sprachkonzepte anschaulich machen sollten. Die Hauptschwierigkeit bestand nicht in der Umsetzung der Schnittstellen selbst. Diese sahen in den einzelnen Modulen recht ähnlich aus und konnten weitgehend analog zueinander implementiert werden. Die Schwierigkeit bestand vielmehr darin, die einzelnen Teile korrekt zu übersetzen, zu linken und schließlich zu binden, so dass diese Teile als Shared-Objects eingebunden werden konnten. Bild 5.8 auf Seite 57 gibt einen Überblick über den Zusammenhang der Schnittstellendateien. Im Folgenden wird der Beispielcode für eine solche Schnittstelle gegeben. Die dazu gehörende Schnittstellenfunktion heißt in diesem Beispiel `loadBase2Model(String path, String name)`:

- **Schnittstelle in Java**

Der folgende Code zeigt, wie die Schnittstellenfunktion innerhalb einer Java-Datei bekannt gemacht wird (mit `public native ...`) und wie das Shared-Object mit der Methode `loadLibrary` eingelesen wird:

```
public class Base2Interface {
    // Eine der Schnittstellenfunktionen
    public native String
        loadBase2Model(String path, String name);

    // Hier wird das Shared-Object libbase2.so geladen.
    static {
        try {
            System.loadLibrary("base2");
        }
    }
}
```

```

        } catch (UnsatisfiedLinkError exc) {
            exc.printStackTrace();
        }
    }
}

```

Das Shared-Object `libbase2.so` muss nun gebaut werden. Dazu müssen die folgenden C- und Ada-Dateien erstellt und kompiliert werden.

- **Schnittstelle in der C-Header-Datei**

Die Header-Datei des C-Moduls wird automatisch von `jawah` generiert. `Jawah` ist ein Programm, das die C-Header-Dateien aus einer Java-Klasse generiert, sofern diese native Methoden einbindet. Die Namenskonvention bei der Benennung der Methoden folgt dem Schema `Java_package.Klasse.Methode`.

```

/* Schnittstelle des Java Native Interface (JNI) */
#include <jni.h>
#ifdef _Included_base2interface_Base2Interface

#define _Included_base2interface_Base2Interface

JNIEXPORT jstring JNICALL
    Java_base2interface_Base2Interface_loadBase2Model
    (JNIEnv *, jobject, jstring, jstring);

#endif

```

- **Schnittstelle in C**

Die C-Datei muss die Ada-Methoden nur an Java weiterreichen, daher werden sie als `extern` deklariert. Diese C-Datei implementiert nun die Definitionen der Schnittstellenfunktionen, wie sie in der C-Header-Datei deklariert wurden.

```

[... ]
/* Die von jawah generierte Header-Datei
   wird eingebunden */
#include "base2interface_Base2Interface.h"
[... ]

extern char* Load_Base2_Model(char*, char*);

```

- **Schnittstelle in der .ads-Datei in Ada**

Die Deklarationsdatei des Ada-Teils macht den Methodennamen für C mittels `pragma Export` bekannt.

```

function Load_Base2_Model (Filepath: in Chars_Ptr;
                           Dateiname: in Chars_Ptr)
    return Chars_Ptr;
pragma Export (C, Load_Base2_Model, "Load_Base2_Model");

```

- **Schnittstelle in der .adb-Datei in Ada**

Die Definitionsdatei des Ada-Teils implementiert letztlich die Methode.

```

function Load_Base2_Model (Filepath: in Chars_Ptr;
                             Dateiname: in Chars_Ptr)
    return Chars_Ptr is

    Success: Boolean;
    Messages: SMM.System_Message_List;
    RetMessage: Chars_Ptr;
begin
    -- Hier findet der eigentliche Aufruf
    -- der Basismaschinenfunktion statt
end Load_Base2_Model;
  
```

Das folgende Beispiel zeigt, wie man die Teilsysteme der Schnittstelle übersetzen muss, um ein Shared-Object zu erzeugen, welches von der Spieleroberfläche eingelesen werden kann. Bild 5.8 zeigt die dazu notwendigen Schritte.

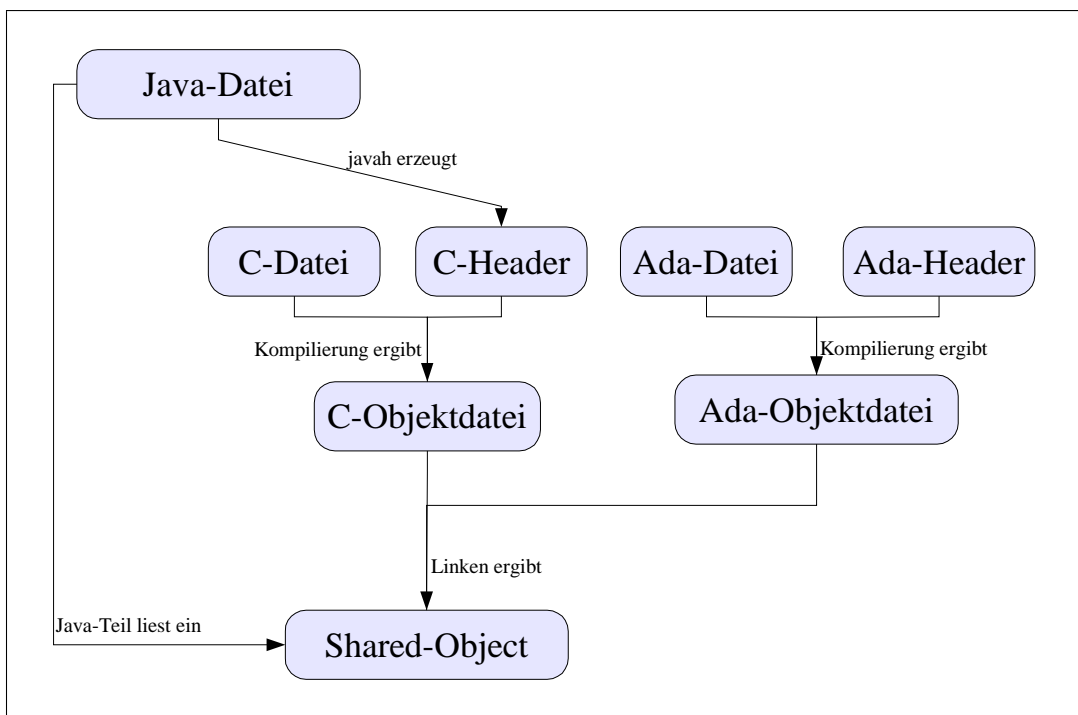


Abbildung 5.8: Erstellen und Einbinden des Shared-Objects

1. **Übersetzen der Java-Schnittstelle**

Die Java-Schnittstelle wird zunächst normal kompiliert. Dabei wird eine *.class-Datei* erzeugt.

2. **Erzeugen einer C-Header-Datei**

Anschließend muss eine Headerdatei erzeugt werden, mit der das C-Modul später übersetzt

werden kann. Der Aufbau der Headerdatei folgt bestimmten Regeln, damit das Java-Modul die Funktionen korrekt ansprechen kann. Vom Hauptpfad ruft man die Anwendung `javah` auf, welche die Headerdatei generiert. Dabei ist zu beachten, dass man den vollen Namensraum der Schnittstelle auf der Kommandozeile angibt, da sonst die Funktionen falsch benannt werden. Ein Beispiel für den Aufruf:

```
javah irgend.ein.package.Schnittstelle
```

Dies würde eine C-Header Datei für die Java-Datei `irgend/ein/package/Schnittstelle.class` erzeugen.

3. Erstellen der C und Ada-Schnittstelle

Ein Beispiel für die Implementierung einer solchen Schnittstelle ist im Code gegeben, der zu dieser Diplomarbeit ausgeliefert wurde. Die Benennung der Methoden des C-Teils muss den Definitionen der Headerdatei entsprechen.

4. Übersetzen der Teilsysteme

Nun muss zunächst die C-Schnittstelle als Objektdatei gebaut werden. Anschließend übersetzt man die Ada-Teile entsprechend, bindet und linkt sie danach zusammen. Dabei muss man beachten, dass alle Teile als C-Dateien übersetzt werden, damit sie später von Java eingebunden werden können. Die genauen Compileroptionen unterscheiden sich von Betriebssystem zu Betriebssystem erheblich. Der Diplomarbeit werden daher Beispiele für Linux und Solaris beigefügt.

5.4.2 Schnittstelle für verteiltes Arbeiten

Da die Implementierung der Schnittstelle wider erwarten schnell fertiggestellt werden konnte, was damit zusammenhing, dass die Übersetzung der Module durch den Ada-Compiler recht komfortabel zu bewerkstelligen war, beschloss ich, im ersten Iterationszyklus auch gleich die Schnittstelle für das verteilte Arbeiten zu implementieren. Dafür wurden wieder verschiedene Prototypen angefertigt, die die unterschiedlichen Möglichkeiten verteilten Arbeitens implementierten. Es gab jeweils einen Prototypen für Sockets, RMI und XML-RPC. Da die Umsetzung mit XML-RPC mit Abstand den geringsten Aufwand verursachte und der Aufgabe völlig gerecht wurde, entstand die Umsetzung des verteilten SESAM-Systems unter Zuhilfenahme dieses Ansatzes. Die Beispielimplementierung verwendet hierbei den Standardport für HTTP-Protokolle.

5.4.3 Entwicklung der Spieleroberfläche

Der zweite Iterationszyklus der Implementierung setzte auf der Java-Schnittstelle auf. Zuerst wurde jedoch eine graphische Oberfläche entwickelt, die ohne die Anbindung an diese Schnittstelle ausgeführt werden konnte. Die Schnittstelle wird erst von den Plug-ins aktiviert und benutzt, so dass es im zweiten Teil der Implementierung erneut zu einer Aufteilung der Arbeitspakete kam. Die Oberfläche ohne Plug-ins hat nur wenig Funktionalität und war deshalb recht zügig implementiert. Die wichtigsten Klassen sind in Bild 5.9 auf der nächsten Seite zu erkennen. Im Bild nicht zu erkennen ist die Klasse `AppProperties`, die die Attribute des Rahmens der Spieleroberfläche enthält.

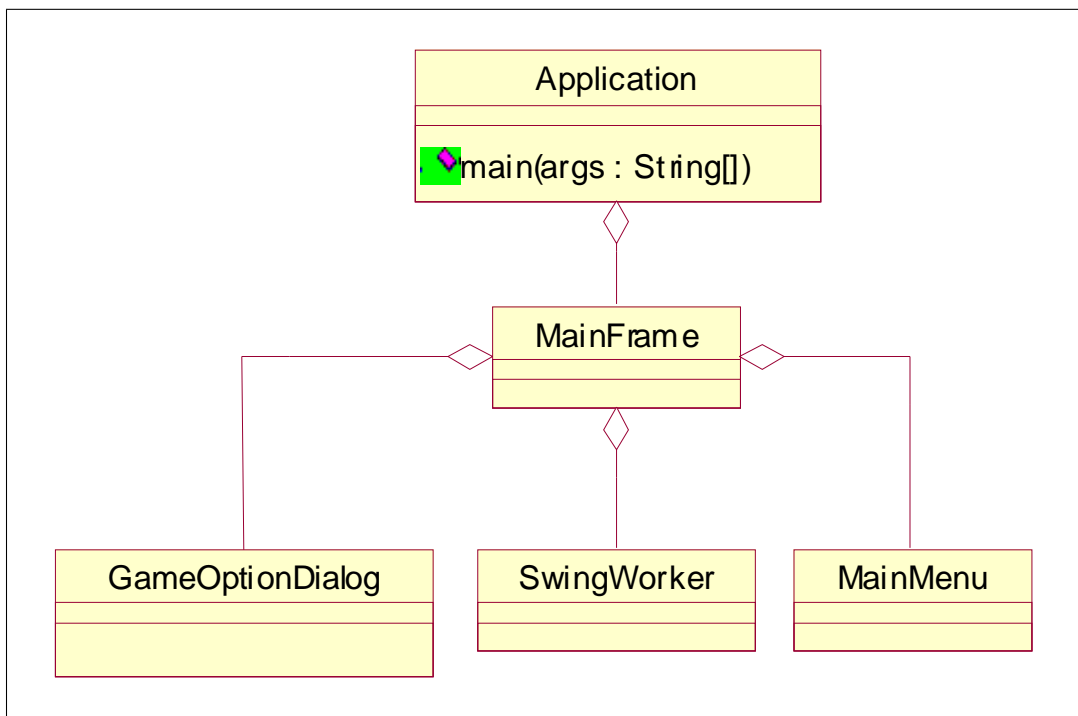


Abbildung 5.9: Überblick über den Entwurf der Plug-in unabhängigen Teile der Spieleroberfläche

5.4.4 Plug-ins

Da die Plug-ins sämtliche Nachrichten und Kommandos des QS-Modells verarbeiten müssen, wurde im ersten Schritt die Grundfunktionalität der Plug-ins implementiert, um anschließend im zweiten Schritt die Modellnachrichten und Kommandos vollständig zu integrieren. Zusätzlich wurde noch ein Mini-Dolmetscher implementiert, um das Spiel auch tatsächlich durchführen zu können. Da der Originaldolmetscher nicht wiederverwendet werden konnte, wurde dieser einfache Interpreter geschrieben. Im Gegensatz zum Originaldolmetscher werden in dieser Variante keine Teilsätze erkannt sondern lediglich Stichworte. Bei bestimmten Stichworten kann der Interpreter die entsprechenden Basismaschinenbefehle „erraten“. Trotzdem verursachte gerade diese Implementierung großen Aufwand, da selbst eine simple Stichworterkennung aufwändig ist. Für weitere Plug-ins kann dieser Interpreter verwendet und angepasst werden. Der Interpreter gehörte nicht zu den Hauptzielen dieser Diplomarbeit. Um jedoch ein SESAM-Spiel zu ermöglichen, wurde ein einfacher Interpreter umgesetzt, der allerdings noch wenig Möglichkeiten für eine Freitexterkennung zulässt.

5.4.5 Graphische Umsetzung

Ein Bestandteil der Entwicklung von Oberflächen, der gerne unterschätzt wird, ist die Entwicklung eines geeigneten Layouts und die graphische Gestaltung der Oberfläche. Bei der Umsetzung der Mitarbeiterfiguren wurde Wert darauf gelegt, den Aufwand für eine Erweiterung der Mitarbeiter eines Modells möglichst gering zu halten. Daher wurde nicht jedem Entwickler des QS-Modells ein eigenes Bild zugewiesen, sondern eine Art Bausatz. Dieser Bausatz umfasste Teile für Hosen, Pullover, Nasen, Augen, Köpfe usw. Diese Teile sind frei kombinierbar, ähnlich wie bei Lego-Figuren, so dass

jeder Mitarbeiter eine individuelle Zusammenstellung verschiedener Körperbauteile bekommt. Wird nun ein weiterer Entwickler in das Modell eingebaut, muss ihm lediglich ein eigenes Set an Teilen zugewiesen werden. Die Räume wurden mittels eines Raytracingprogramms, welches auch Radiosiyalgorithmen unterstützt, entworfen und gerendert. Die Szenen im Spiel ergeben sich später daraus, dass ein Raum Schicht für Schicht durch Hinzufügen von Einzelteilen der Szene aufgebaut wird.

5.5 Entwicklung weiterer Plug-ins

Dieser Abschnitt erläutert, wie Plug-ins für Hochsprachenmodelle entwickelt werden können und welche Teile des bestehenden QS-Modell-Plug-ins angepasst werden müssen, um weitere Modelle zu unterstützen.

Bislang unterstützt die Spieleroberfläche noch kein automatisches Erkennen von Plug-ins, weshalb auch Teile geändert werden müssen, die nicht zum eigentlichen Plug-in gehören. In Zukunft ist geplant, Plug-ins tatsächlich durch Installieren in ein Plug-in Verzeichnis automatisch hinzuzufügen und im SESAM-System verfügbar zu machen. Da dieser Umstand vorhanden ist, möchte ich zunächst darauf eingehen, wie man die vom Plug-in unabhängigen Teile der Spieleroberfläche ändern muss, um ein neues Plug-in zur Verfügung zu stellen.

5.5.1 Änderungen an Plug-in unabhängigen Teilen

Für das Einbinden eines neuen Plug-ins muss die Klasse `MainFrame` geändert werden. Sie befindet sich im Package `gui`.

- Die Methode `loadModelDialog` stellt einen Auswahldialog für die Plug-ins zur Verfügung. Dieser muss für weitere Plug-ins angepasst werden. Dazu muss ein neues Bild (Icon) für das Plug-in erstellt und angezeigt werden. Die `Switch`-Anweisung muss die Wahl des Spielers entsprechend behandeln. Ein Beispiel für die Erweiterung des Dialogs ist im Code als Kommentar enthalten.
- Die Methode `loadModel` wird bei der Wahl des Plug-ins für das QS-Modell aufgerufen. Die Methode instanziiert daraufhin ein Objekt der Klasse `PluginWorker`, die einen Thread für das Laden des eigentlichen Modells erstellt.
- Die interne Klasse `PluginWorker` lädt das Plug-in, welches durch die Instanzierung die Basismaschine anspricht und die Startsituation des Modells herbeiführt.

Will man ein neues Plug-in hinzufügen, muss dieser `PluginWorker` das entsprechende Plug-in instanzieren. Statt der Zeile

```
plugin.Plugin myPlugin =  
    new plugin.qaplugin.QAPlugin(MainFrame.this)
```

muss das jeweils ausgewählte Plug-in geladen werden. Dafür gibt es nun zwei Möglichkeiten. Die erste Möglichkeit ist, die Methode `loadModel` zu duplizieren und darin ein Objekt einer weiteren Klasse zu instanzieren, die das neue Plug-in ähnlich der Klasse `PluginWorker` lädt.

Wesentlich eleganter wäre die Abänderung der Klasse `PluginWorker`, so dass die zu instanzierende Plug-in Klasse als Parameter übergeben wird.

Weitere Änderungen müssen eventuell auch an den Property-Klassen erfolgen. Bei Fertigstellung dieser Diplomarbeit gab es nur eine solche Klasse für die plug-in-unabhängigen Teile der GPI. Die Klasse `AppProperties` befindet sich im Package `configuration`. Es handelt sich hierbei um eine Singletonklasse, die statisch aufgebaut ist. Das bedeutet, dass die Attribute, die diese Klasse zur Verfügung stellt und verwaltet, über den Klassennamen angesprochen werden kann. Die Attribute selbst werden in einer Konfigurationsdatei beim Spieler gespeichert. Die Attribute müssen unter Umständen angepasst werden, z.B. wenn das Spiel in einer anderen Sprache ausgeführt werden soll.

5.5.2 Erstellen einer Plug-in Klasse

Das eigentliche Plug-in bekommt als Parameter lediglich eine Referenz auf das Objekt der Klasse `MainFrame`, welches den Oberflächen-Rahmen darstellt. Diese Klasse bietet dem Plug-in dafür die folgenden Schnittstellen an:

- **public JMenu getMenu()**
Diese Methode liefert das Plug-in-Menü des Hauptmenüs als `Widget` zurück, so dass die Plug-ins hier eigene Einträge in das Menü setzen können.
- **public JPanel getPluginPanel()**
Diese Methode liefert eine Referenz der Swing-Widgets `JPanel` zurück, welches den Bereich der Oberfläche angibt, in die ein Plug-in seine eigenen Widgets und Bilder setzen kann.

Weitere Verbindungen zwischen dem Oberflächen-Rahmen und dem Plug-in gibt es nicht. Alle Plug-ins müssen von der abstrakten Klasse `Plugin`, zu finden im Package `plugin`, ableiten. Diese Klasse enthält die folgenden Methoden, die vom Plug-in realisiert werden müssen oder benutzt werden können:

- **protected void setStatus(boolean isSuccessful)**
Diese Methode sollte vom Plug-in dann aufgerufen werden, wenn die Verbindung zur Basismaschine erfolgreich hergestellt ist und das Plug-in vollständig geladen wurde, so dass ein Spiel durchgeführt werden kann. Erst wenn diese Methode mit dem Parameter `true` aufgerufen wurde, wird die Simulation gestartet.
- **protected boolean connect()**
Diese Methode stellt eine Verbindung zur Basismaschine her.
- **public boolean getStatus()**
Diese Methode liefert den Zustand des Plug-ins. Ist das Plug-in noch nicht vollständig geladen oder gab es Fehler während des Ladevorgangs oder des Verbindungsaufbaus zur Basismaschine, liefert diese Methode `false` zurück. Die Methode liefert nur bei einem vollkommen problemlosen Laden des Modells `true` zurück.
- **abstract public void startSimulation()**
Nach dem Laden des Plug-ins wird der Zustand des Plug-ins mittels der Methode `getStatus()` erfragt. War der Ladevorgang erfolgreich, wird die Methode `startSimulation()` aufgerufen, die das Modell initialisiert und die Startsituation herstellt. Die Methode ist als `abstract` deklariert. Das bedeutet, dass jedes Plug-in die Methode selbst implementieren muss.

- **public GUI2Base getBaseConnection()**

Diese Methode liefert eine Referenz auf das Objekt zurück, welches die Schnittstellenfunktionen zur Basismaschine enthält. Wird ein Kommando des Plug-ins an die Basismaschine gesendet, so geschieht das über dieses Objekt.

Möchte man den Dolmetscher auf Seite der Spieleroberfläche haben (im Gegensatz dazu kann man den Dolmetscher auch auf Seite der Basismaschine implementieren), so steht dem Entwickler noch das Interface `Interpreter` zur Verfügung. Dieses besitzt lediglich eine Methode:

- **public String translate(String userCommand)**

Die Methode ist als abstrakt deklariert und muss vom Plug-in selbst implementiert werden. Sie soll die Spielereingabe `userCommand` in ein Kommando übersetzen, das für die Basismaschine verständlich ist. Die Verwendung dieser Methode ist jedoch optional.

5.5.3 Beispiel: Ein Plug-in für das QS-Modell

Da die bisherigen Modelle untereinander recht ähnlich sind (z.B. das QS-Modell und das QSVA-Modell), soll hier nun detailliert vorgestellt werden, welche Methoden und Klassen des QS-Modell-Plug-ins welchen Zweck erfüllen, um den zukünftigen Entwicklern ein Gefühl dafür zu geben, welche Gedanken hinter den Implementierungsentscheidungen stecken und wie man selbst solche oder ähnliche Plug-ins entwickeln kann.

Generelle Aufteilung

Dem Plug-in für das QS-Modell wurde zunächst ein eigener Bereich zugewiesen, d.h. es wurde ein Package für dieses Plug-in erstellt. Dies sollte für jedes Plug-in geschehen, damit einerseits die Plug-ins klar abgegrenzt sind und die einzelnen Quelltext-Dateien nicht durcheinander geraten. Zum anderen lassen sich so die Plug-ins leichter in Jar-Dateien¹ verpacken.

Die Startklasse

Das QS-Modell-Plug-in² besitzt eine Startklasse, die nur dafür sorgt, dass eine Verbindung zur Basismaschine hergestellt, das QS-Modell geladen und die Startsituation initialisiert wird. Diese Klasse heißt `QAPLugin`. Sie leitet sich direkt von der Klasse `Plugin` ab.

Wie bereits beschrieben, wird die Methode `startSimulation` aufgerufen, sobald das Modell vollständig geladen ist. Diese Methode wird in der Klasse `QAPLugin` nun implementiert und instanziiert Objekte der folgenden Klassen:

- **QADocument**
- **QAControl**
- **QAControlView**
- **QAMenu**

Die Klassen werden im Folgenden genauer beschrieben.

¹Jar-Dateien entsprechen Zip-Dateien, d.h. sie packen mehrere Dateien zusammen und komprimieren sie. Im Gegensatz zu Zip-Dateien können die Jars von Java eingebunden und im gepackten Zustand ausgeführt werden.

²In der Implementierung wird aus Konsistenzgründen die englische Bezeichnung `QAPLugin` verwendet

Das Hauptdokument QADocument

Diese Klasse enthält alle Informationen und Daten des Modells und stellt die Schnittstelle für diese Daten zur Spieleroberfläche dar. Die eigentlichen Daten werden in der Basismaschine gehalten, das QADocument stellt lediglich Anfragen zu diesen Daten, wenn die Spieleroberfläche eine solche Information benötigt. Es werden aber auch zusätzliche Daten gehalten. Zum einen sind dies die Räume, die im Modell nicht definiert sind, zum anderen Daten, die häufig erfragt werden müssten, wobei jedoch immer ein Simulationsschritt im Modell ausgeführt werden müsste. Ein Beispiel dafür sind die eingestellten Personen. Die Cafeteria und auch die anderen Räume müssen Personen anzeigen und die Liste der eingestellten Personen aktuell halten. Ein Nachfragen, welche Person eingestellt ist, kostet jedoch Simulationszeit, weil die Antwort auf diese Frage erst im nächsten Simulationsschritt ausgegeben wird. Deshalb wird die Liste der verfügbaren Entwickler lediglich einmal angefordert und anschließend intern verwaltet.

Die Klasse QADocument ist, wie der Name andeutet, von der Klasse Document abgeleitet. Das bedeutet, in dieser Klasse werden ausschließlich Daten gehalten. Die Darstellung der Daten übernehmen weitere Klassen. Dazu zählen die Räume sowie die Eingabemaske am unteren Fensterrand der Spieleroberfläche. Somit besitzt die Klasse QADocument hauptsächlich Methoden zum Setzen und Erfragen von Daten des Modells und eine Methode zum Bearbeiten von Benutzerkommandos sowie eine Methode, die Nachrichten der Basismaschine verwaltet. Ändern sich die Daten, wird die Methode `updateViews` aufgerufen, die sämtlichen Views des QADocuments mitteilt, dass sie sich neu darstellen sollen, falls sich Daten geändert haben, die für sie relevant sind.

Falls eine Änderung des Modells erfolgt oder eine Weiterentwicklung des Plug-ins nötig ist, so müssen in dieser Klasse evtl. Anpassungen gemacht werden, sofern dies Daten betrifft, die vom QADocument gehalten werden.

Die Kontrollleiste

Die Klasse QAControl ist ein ausgelagerter Teil der Klasse QADocument. Hier werden alle Daten verändert, die mit der Kontrollleiste am unteren Rand der Spieleroberfläche visualisiert werden. Dazu zählen die Räume sowie Zeitangaben im Spiel. Die dazugehörigen Daten werden im QADocument gehalten, sie werden von der QAControl lediglich neu gesetzt, um die Hauptklasse nicht unübersichtlich werden zu lassen. Damit Veränderungen der Daten korrekt gesetzt werden, z.B. wenn eine Spielernachricht eine solche Veränderung erfordert, ist die Klasse QAControl eine Implementierung des Interfaces View, welche sich beim QADocument registriert.

Bei einer Veränderung des Modells sollten hier alle Informationen der Basismaschine bearbeitet werden, die die Kontrollleiste betreffen, z.B. wenn zusätzliche Informationen wie eine Liste aller Dokumente mit aufgenommen werden soll.

Die Darstellung der Kontrollleiste

Die Klasse QAControlView übernimmt die Darstellung der Kontrollleiste. Sie holt sich ihre Daten vom QADocument. Sie ist also ebenfalls eine Implementierung der Klasse View und registriert sich beim QADocument, um die neuesten Daten bei Veränderung abzuholen und darzustellen. Solange sich die Schnittstellen zum Dokument nicht ändern oder gar Informationen hinzukommen, sind bei einer Änderung des Modells nur geringe Anpassungen nötig.

Das QA-Menü

Die Klasse `QAMenu` stellt das Menü des Plug-ins dar. Auch diese Klasse stellt eine Implementierung der Klasse `View` für das `QADocument` dar. Sie enthält neben Einträgen für Debug- und Tutornachrichten auch eine hierarchisch gegliederte Kommandoauflistung des QS-Modells.

Das Menü muss bei einer Änderung des Modells an die neuen Kommandos angepasst werden.

Die Raumklassen

Die Räume Cafeteria und Projektleiterräum werden von der Klasse `QADocument` erzeugt, und zwar unmittelbar beim Instanzieren dieser Klasse, da die Räume im gesamten Spiel jederzeit verfügbar sind. Die Mitarbeiterräume hingegen werden erst beim Einstellen der Mitarbeiter erzeugt. Eine entsprechende Nachricht bei der Einstellung eines Mitarbeiters erhält die Klasse `QAControl`, die das `QADocument` veranlasst, einen Mitarbeiterraum zu erzeugen oder zu löschen, wenn der Mitarbeiter entlassen wird.

Die Views der Räume, die letztlich die Daten der Räume anzeigen, werden von der Klasse `QAControlView` erzeugt. Diese Klasse erzeugt für jeden neu hinzugefügten Raum eine View, sobald ihre Update-Methode durch das `QADocument` aufgerufen wird.

Sind im Plug-in später neue Raumarten erwünscht, muss nach dem bestehenden Muster für diese Räume jeweils eine `Document` und eine `View` Klasse erzeugt werden. Ferner müssen die Klassen `QADocument`, `QAControl` und `QAControlView` geändert werden.

Die Konfigurationsdateien

Die Konfigurationsdateien des QS-Modell-Plug-ins enthalten im Wesentlichen die Spielerausgaben, die Graphiken der Räume und Mitarbeiter sowie Spieleinstellungen. Soll das Modell auf eine andere Sprache umgestellt werden, andere Graphiken verwendet werden etc., so muss dies an den entsprechenden Stellen der Konfigurationsdateien geändert werden. Sie befinden sich im Paket `plugin.qaplugin.conf`. Die Konfigurationsdateien speichern die Werte als Wertepaare, die dem Schema

```
Name = Wert
```

folgen.

5.6 Test

Der Test der Implementierung erfolgte im Wesentlichen als Blackbox-Test. Es wurde zunächst gegen die Spezifikation und die dadurch gegebenen Testfälle getestet. Gefundene Fehler wurden anschließend behoben. In einem zweiten Test wurden keine weiteren Fehler gefunden.

Anschließend erfolgte ein zweifacher Regressionstest durch einen Testspieler. Dieser führte SESAM sowohl mit der neuen als auch mit der bisherigen Spieleroberfläche gleichzeitig aus und verglich bei jedem Schritt die Ergebnisse. Nach dem ersten Testlauf zeigten sich noch Fehler bei der Verarbeitung einzelner Kommandos, da diese in der neuen Oberfläche zum Teil nicht vollständig implementiert waren. Beim zweiten Testlauf gab es nur bei einer Kommando-Gruppe Probleme, weil der Interpreter bestimmte Stichworte nicht richtig erkannte.

5.7 Beispiel eines Spielverlaufs

Lang ist der Weg durch Lehren, kurz und wirksam durch Beispiele.
Seneca

Dieser Abschnitt stellt ein typisches Spiel mit der neuen Spieleroberfläche dar. Es soll zeigen, wie der Spieler mit der neuen Oberfläche umgehen kann und welche Möglichkeiten geboten werden.

5.7.1 Start des Programms

Zunächst wird das Spiel durch Aufruf eines Startskripts von der Konsole gestartet. Dazu gibt der Spieler den folgenden Text ein:

```
cd <SESAM-Verzeichnis>  
./sesam
```

Nun erscheint die Oberfläche des Spiels. Zu diesem Zeitpunkt besteht noch keine Verbindung zum eigentlichen SESAM-Simulator. Hier kann der Spieler lediglich einige Optionen einstellen, die für das spätere Spiel gelten sollen. In der aktuellen Version ist die Auswahl an Einstellmöglichkeiten auf die Wahl des Spielernamens sowie das Ein- und Ausschalten einer Tipp-des-Tages-Anzeige beschränkt. Der Spielername kann gewählt werden, um später im Spiel persönlich angesprochen zu werden. Beim ersten Start des Programms ist diese Option auf den Login-Namen des Anwenders gesetzt.

Im Startbildschirm hat der Spieler weiterhin die Möglichkeit, eine Online-Hilfe aufzurufen. Die tatsächliche Ausgabe eines Hilfetextes wurde in der Diplomarbeit jedoch nicht realisiert, da die Umsetzung viel Zeit gekostet hätte, die statt dessen in qualitätssichernde Maßnahmen investiert wurde.

Die wichtigste Funktion im Startbildschirm stellt das Laden eines Modells dar. Hier kann der Spieler das Plug-in für das SESAM-Modell wählen, mit dem er spielen möchte. Sobald er ein Modell gewählt hat, wird dieses geladen. Dieser Vorgang nimmt etwas Zeit in Anspruch und wird durch das Verändern des Cursors zu einer Sanduhr signalisiert.

5.7.2 Spieldurchführung

Der Spieler kann nun die Simulation durchführen. Ihm stehen dazu die bereits erwähnten Bildelemente zur Verfügung:

- **Bildszene**
- **Dropdown-Box für die Navigation**
- **Eingabefeld für Kommandos**
- **Fortschritts-Button**

Zunächst befindet sich der Spieler in der Cafeteria, in der alle im QS-Modell verfügbaren Mitarbeiter angezeigt werden. Welche Handlungen der Spieler ausführt muss dieser nun selbst entscheiden. Ich werde im Folgenden eine *typische* Vorgehensweise beschreiben. Die Kommandos, die ein Spieler im Eingabefeld eingibt, werden besonders hervorgehoben.

Der Spieler geht anfangs in den Projektleiterräum, indem er diesen in der Dropdown-Box auswählt. Dort wird ihm eine Nachricht präsentiert, in der die Aufgabenstellung des Projekts beschrieben wird.

Weiterhin erfährt er die Rahmenbedingungen für die Durchführung, etwa das verfügbare Budget, die Entwicklungsfrist und Qualitätsvorgaben an die Lösung. Um die Vorgaben nicht zu vergessen, markiert der Spieler den Text und fügt ihn per Copy&Paste in eine Textdatei ein, die er unabhängig vom Spiel führt und verwaltet. Hätte der Spieler diesen Schritt vergessen, wäre die Information nach einiger Zeit nicht mehr abrufbar.

Als nächstes macht sich der Spieler Gedanken zur Durchführung des Projekts. Er muss Mitarbeiter einstellen und ihnen Aufgaben zuweisen. Dazu begibt er sich in die Cafeteria und spricht einige der Mitarbeiter an und befragt sie zu ihren Referenzen und Kosten.

```
inspect Bernd
```

Das System zeigt ihm daraufhin die Daten von Bernd an. Diese Daten enthalten die Kosten für den Mitarbeiter, in welchen Gebieten der Mitarbeiter große Erfahrung hat, in welchen er mittlere und in welchen er überhaupt keine Erfahrung hat. Der Spieler entscheidet sich dann für einen der Mitarbeiter und stellt ihn ein.

```
hire Diana
```

Die Mitarbeiterin Diana verschwindet daraufhin aus der Cafeteria. In der Dropdown-Box des Navigationselements wird ein Raum für Diana hinzugefügt, der fortan ausgewählt werden kann.

Der Spieler entscheidet sich, keine weiteren Mitarbeiter einzustellen und begibt sich in das Büro von Diana. Dort erteilt er Diana den Auftrag, mit dem Kunden über seine Wünsche bezüglich des Produkts zu sprechen.

```
ask Diana to talk with the customer
```

Diana antwortet daraufhin, dass sie dieses sogleich beginnt.

Da das Gespräch Zeit in Anspruch nimmt, muss der Spieler die Simulationszeit voranschreiten lassen, um das Ergebnis des Gesprächs zu erfahren. Er könnte aber in diesem Simulationsschritt auch noch andere Aufgaben angehen, beispielsweise einen Mitarbeiter einstellen. Der Spieler entscheidet sich jedoch für die Weiterschaltung der Simulationszeit, indem er auf den Proceed-Button drückt. Diana berichtet ihm daraufhin, dass sie mit dem Kunden gesprochen hat und dass sie eine Ahnung davon hat, welche Anforderungen er an das Produkt stellt. Diese Information schreibt sie automatisch in das Analysedokument, auf das alle Mitarbeiter Zugriff haben.

Der Spieler kann nun eine Spezifikation erstellen lassen. Da diese Arbeit aufwändig ist, entschließt er sich, einen weiteren Mitarbeiter einzustellen. Dazu begibt er sich wieder in die Cafeteria und führt diesen Schritt durch. Er begibt sich anschließend in die Büros der Angestellten und erteilt ihnen den Auftrag, die Spezifikation zu erstellen.

```
ask Diana to create the specification
ask Bernd to create the specification
```

Hat der Spieler alle Aufgaben für diesen Simulationsschritt getätigt, drückt er erneut auf den Proceed-Button. Bernd und Diana melden daraufhin, dass sie mit der Arbeit an der Spezifikation begonnen haben. Zudem wird eine kleine Spezifikation auf dem Schreibtisch der Mitarbeiter gezeigt, so dass der Spieler sehen kann, an welchem Dokument die Angestellten gerade arbeiten. Der Spieler kann jetzt so oft Proceed drücken, bis die Mitarbeiter melden, dass sie mit ihrer Aufgabe fertig sind. Daraufhin verschwindet auch das Buch auf den Tischen. Während die Mitarbeiter noch arbeiten, kann der Spieler jederzeit die Tätigkeit eines Angestellten abrufen, in dem er beispielsweise

talk with Diana

eingibt. Daraufhin meldet Diana ihre aktuelle Tätigkeit.

Ist die Spezifikation fertiggestellt, kann der Spieler im Projektleiterräum den Status des Dokuments einsehen.

inspect specification

Dort wird ihm angezeigt, wie viele Seiten die Spezifikation hat und wie viel Prozent der Anforderungen damit in etwa abgedeckt werden. Nun kann der Spieler z.B. weitere Mitarbeiter einstellen, die das System entwerfen, ein Review der Spezifikation durchführen oder sonstige Aufgaben übernehmen.

Das weitere Vorgehen des Spiels sollte nun ersichtlich sein, so dass auf eine vollständige Beschreibung eines Spielablaufs an dieser Stelle verzichtet werden kann.

Eine Besonderheit des Spiels gilt es jedoch noch zu beachten. Weiß der Spieler einmal nicht weiter, so hat er die Möglichkeit, das Kommandomenü zu benutzen. Wählt er allerdings diesen Weg, so wird die Simulationszeit automatisch vorangeschritten. So soll der Spieler ermutigt werden, möglichst ohne Hilfe die richtige Vorgehensweise zu erlernen. Das Benutzen des Menüs soll das Nachschlagen in Handbüchern etc. simulieren, die in einem realen Projekt ebenso Zeit kosten.

Kapitel 6

Konzepte für verteiltes Arbeiten

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie Lamport

Dieses Kapitel beschreibt Konzepte für die verteilte Ausführung von Basismaschine und Spieleroberfläche. Damit soll es den Spielern in Zukunft möglich sein, SESAM lokal zu starten oder sich entfernt auf der Basismaschine einzuloggen.

6.1 Verteilte Systeme

Ein verteiltes System ist ein System, das auf einer Menge von Rechnern ausgeführt wird, die nicht über einen gemeinsamen Speicher verfügen, und das sich dem Benutzer wie ein einzelner Rechner darstellt [Tan94].

Bezogen auf SESAM könnte dies beispielsweise bedeuten, die Basismaschine auf einem anderen Rechner laufen zu lassen als die Spieleroberfläche. Die Spieler könnten so ihre Oberfläche lokal auf einem Rechner ausführen lassen und sich mit einem entfernten Rechner, auf dem die Basismaschine läuft, verbinden. Solche Architekturen nennt man auch Client-Server Modelle. Der Server stellt Funktionen zur Verfügung, die ein oder mehrere Clients verwenden können. Der Server hat kein *Wissen* über die Clients. Methodenaufrufe sind also nur in einer Richtung möglich, in diesem Fall kann die Spieleroberfläche also nur die Schnittstellenfunktionen der Basismaschine aufrufen, die Basismaschine jedoch keine Methoden der Spieleroberfläche.

6.2 Gründe für verteiltes SESAM

SESAM benötigt bei der Ausführung erheblich Speicher und Rechenzeit. Vor allem, wenn mehrere Benutzer gleichzeitig spielen wollen, wird das System, auf dem die Anwendung läuft, stark belastet. Diese Belastung kann durch ein verteiltes Ausführen der Basismaschine und der Spieleroberfläche verteilt werden, da die Ressourcen, die die Spieleroberfläche benötigt, von einem lokalen Rechner gestellt werden können.

Durch die Aufteilung zwischen Spieleroberfläche und Basismaschine sind die Schnittstellen und Aufgaben der einzelnen Teile gut zu trennen. Somit wird die Wartbarkeit des Produkts erleichtert, die Teile können einzeln für sich betrachtet werden und können sogar völlig getrennt voneinander neu

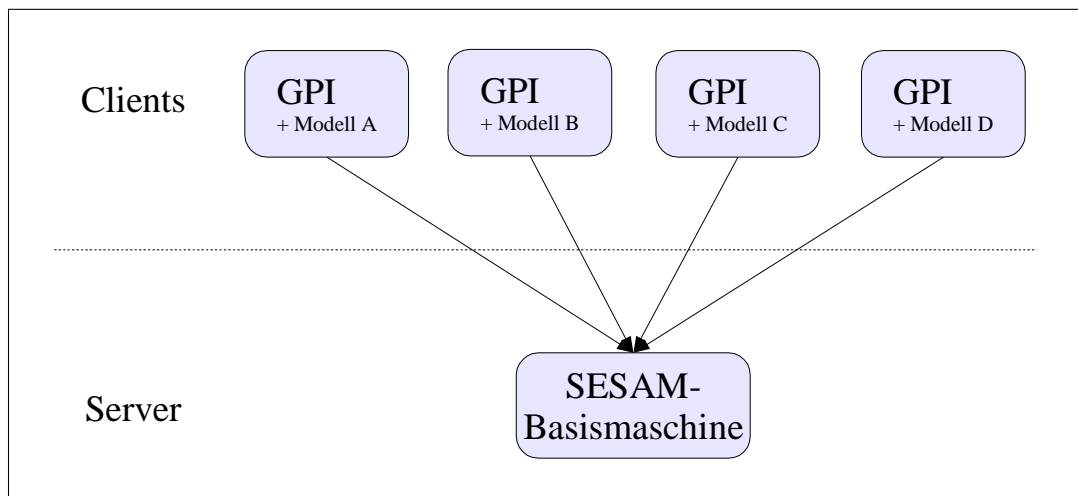


Abbildung 6.1: Verteilung des SESAM-Spiels im Idealfall

entwickelt werden. Die Abkopplung verringert auch die Gefahr von Seiteneffekten in den Programmteilen, die eventuell bei einer Vermischung der Basismaschine und GUI auftreten können. Ebenso ist es mit der Trennung möglich, unterschiedliche Sprachkonzepte für die beiden Teilsysteme einzusetzen, solange beide die entsprechenden Protokolle verstehen und sich an die Schnittstellenspezifikation halten.

Damit könnte es beispielsweise in Zukunft auch möglich sein, webbasierte Oberflächen für SESAM anzubieten, was die Verfügbarkeit des Programms erhöhen würde. Dazu kommt, dass die verteilte Anwendung zusätzlich das Potential hat, durch Parallelisierung der Prozesse eine Steigerung der Leistung des Systems zu schaffen.

Der Hauptgrund für verteilte Architekturen in kommerziellen Spielen ist die Möglichkeit, mehrere Spieler an einem Spiel teilnehmen zu lassen. Übertragen auf SESAM bedeutet das zum Beispiel, dass sich mehrere Spieler in einem Spielmodell einloggen und das Spiel gemeinsam gestalten können. So könnte es neben dem Projektleiter evtl. noch weitere Hierarchieebenen geben, die von menschlichen Spielern simuliert werden, so dass in Zukunft auch Modelle entstehen könnten, die Probleme in hierarchisch organisierten Projektgruppen simulieren können.

Die Aufteilung von GUI und Basismaschine ist der erste Schritt für eine verteilte Anwendung und wird in dieser Diplomarbeit prototypenhaft umgesetzt.

Bild 6.1 zeigt den Idealfall für eine verteilte Lösung für SESAM. Jeder Spieler hat seine für sich konfigurierte Spieleroberfläche und kann ein beliebiges Modell laden. Die Rechner der Spieler müssen lediglich die Oberfläche darstellen, die Berechnungen im Modell mittels der Basismaschine werden auf einem Server durchgeführt. Dazu muss die Basismaschine jedoch gleichzeitig mehrere Benutzer mit verschiedenen Modellen verwalten können.

6.3 Möglichkeiten zur Umsetzung

Dieser Abschnitt soll einen kleinen Ausschnitt aus den Möglichkeiten zur Umsetzung einer verteilten Anwendung geben. Zu diesem Zweck wurden die vier häufigsten APIs¹ und Kommunikationsschnitt-

¹API steht für Application Programming Interface.

stellen herausgegriffen, die vor allem im Java-Umfeld Verwendung finden. Die vier vorgestellten Konzepte stellen verschiedene Abstraktionsebenen für Netzwerkkommunikation dar. Sie bauen jedoch nicht zwangsläufig aufeinander auf. Beginnen möchte ich mit der geringsten Abstraktionsstufe, den Sockets.

6.3.1 Sockets

Sockets sind eine Methode für die Kommunikation zwischen Rechnern in einem Netzwerk. Aufbauend auf dieser Methode gibt es APIs, die eine Anwendungsschnittstelle für viele Programmiersprachen bereitstellen. Die am häufigsten verwendete Implementierung einer solchen API ist das Berkeley Unix C Interface.

Hierbei wird ein Socket (definiert durch eine IP-Adresse und einen Port) ähnlich einer Datei geöffnet, so dass die Programme lesend und schreibend darauf zugreifen können. Es werden also nicht die physikalischen Dateien geöffnet, sondern vielmehr Kommunikationskanäle. Analog zu Filehandlern gibt es für die Sockets Funktionen, die den Status überprüfen. Schickt nun ein Programm Daten über das Socket, so ändert das Socket seinen Status auf lesbar und die Anwendung kann die Daten empfangen und bearbeiten [Vog02].

Der entscheidende Nachteil von Sockets liegt darin begründet, dass man keine Datentypen direkt versenden kann. Über Sockets können lediglich Zeichenketten versendet werden. Andere Datentypen müssen über Zeichenketten codiert werden, so dass man ein eigenes Protokoll entwickeln muss. Gleiches gilt für Methodenaufrufe. Auch hier muss ein eigenes Protokoll die Aufrufe auf die entsprechenden Methoden der entfernt laufenden Anwendung vermitteln. Im Allgemeinen werden daher Sockets nur als grundlegende Technologie für weiter abstrahierte Konzepte verwendet.

6.3.2 RPC

Eine solche höhere Abstraktionsebene stellt RPC dar. RPC steht für *Remote Procedure Calls*. Für RPC gibt es mehrere, unterschiedliche Spezifikationen und Implementierungen, darunter die Open Network Computing (ONC) RPC, Distributed Computing Environment (DCE) RPC, und die RPCs der International Organization for Standardization (ISO).

Bei RPC sollen Prozeduren und Funktionen von entfernten Anwendungen von einer lokalen Anwendung so aufgerufen werden können, als seien es lokale Prozeduren. RPC stellt bereits Grunddatentypen zur Verfügung, die bei den Prozeduraufrufen verwendet werden können.

Für die Diplomarbeit wurde eine spezielle Implementierung von RPC gewählt: XML-RPC. Diese RPC Variante verwendet XML-Dateien, in denen grundlegende Datentypen wie Integer oder Boolean zwischen den verteilten Anwendungen übertragen werden. Gegenüber Sockets hat man somit den Vorteil, direkt Methoden der entfernten Anwendung aufzurufen und einfache Datentypen für die Parameter zu verwenden.

Für objektorientierte Programme hingegen ergibt sich der Nachteil, dass komplexe Datentypen, die in Objekte verpackt sind, über RPC nicht versendet werden können, es sei denn, man schreibt ein eigenes Protokoll hierfür. Auch hier gibt es nun abstraktere Lösungen für die Kommunikation zwischen objektorientierten Programmen.

6.3.3 RMI

RMI die Standardlösung für verteilte Anwendungen, die in Java implementiert sind. RMI steht für *Remote Method Invocation* und kann, ähnlich wie RPC, Methoden von entfernt laufenden Anwen-

dungen aufrufen. Mit RMI ist es nun möglich, auch Objekte als Parameter zu versenden, so dass keine Umwandlung der Datentypen erforderlich ist.

Eine vereinfachende Erklärung für den Ablauf einer RMI Kommunikation ist wie folgt: Eine Anwendung kann bei RMI Methoden für andere, entfernt laufende Anwendungen anbieten. Hierzu übergibt sie die Spezifikation der Methoden einer sogenannten Registry. Ein Programm, das eine dieser Methoden nun verwenden möchte, bekommt nach Anfrage bei der Registry Zugriff auf diese Methoden und kann sie daraufhin wie lokale Methoden ausführen.

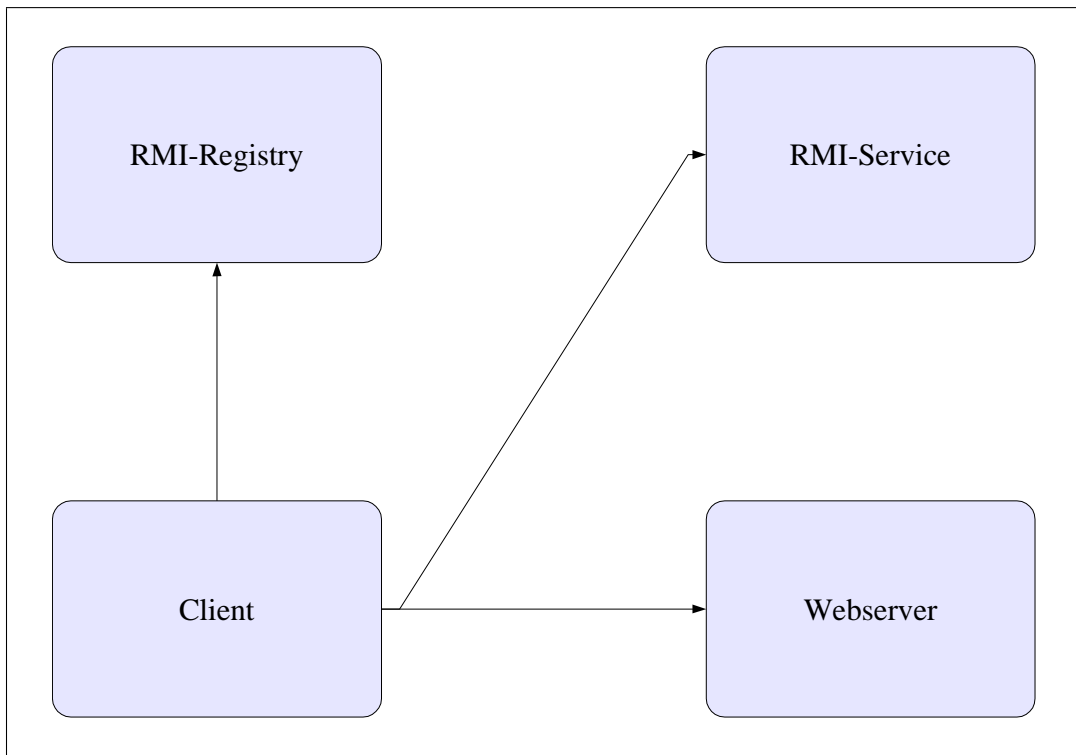


Abbildung 6.2: Remote Method Invocation (RMI)

Der Unterschied zu RPC liegt darin, dass Objekte versendet werden können. RMI hat jedoch den Nachteil, dass nur Java-Programme auf diesem Weg miteinander kommunizieren können. Anwendungen, die in anderen Programmiersprachen geschrieben wurden, können RMI nicht nutzen.

6.3.4 CORBA

Die höchste hier betrachtete Abstraktionsstufe für verteilte Anwendungen stellt CORBA dar.

CORBA, die **Common Object Request Broker Architecture**, ist eine Definition für verteilte Anwendungen, deren Ziel es ist, eine orts-, plattform- und implementierungsunabhängige Kommunikation zwischen Anwendungen zu gewährleisten. CORBA ist im Gegensatz zu RMI nicht an eine Sprache gebunden, so dass die Systeme im Netzwerk nicht alle einen Java-Interpreter benötigen. So kann ein Java-Programm mittels CORBA über ein Netzwerk mit einem Ada-Programm kommunizieren.

Damit eine Applikation einer entfernt laufenden Anwendung Methoden über CORBA anbieten kann, müssen diese Schnittstellenmethoden speziell beschrieben werden. Dies geschieht mittels der

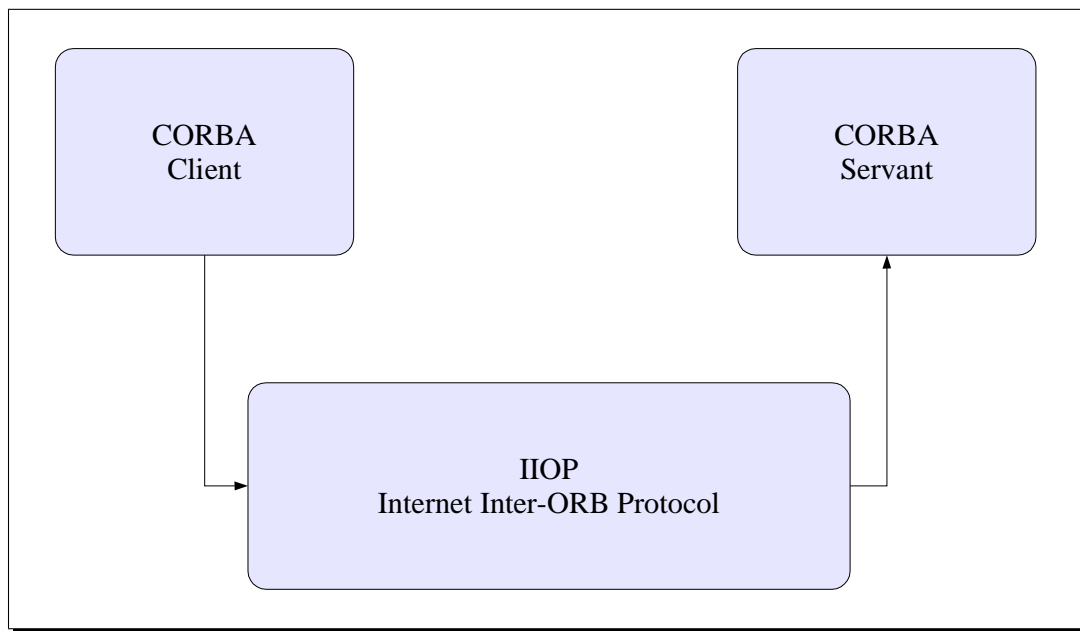


Abbildung 6.3: CORBA

sogenannten *Interface Definition Language*, kurz IDL. Diese IDL vermittelt die Datentypen, Pointer und Objekte zwischen den Programmiersprachen.

Das Vorgehen einer Kommunikation mittels CORBA ist in Bild 6.3 dargestellt. Methodenaufrufe zwischen den entfernten Anwendungen werden an Object Request Brokers (ORBs) geleitet, den technischen Umsetzungen der CORBA-Definition, die mittels des Internet Inter-ORB Protocols (IIOP) miteinander kommunizieren. Das ORB ist dafür zuständig, das Serverobjekt zu finden, die gewünschte Funktion aufzurufen und das Ergebnis an den Client zurückzureichen. Die ORBs stellen somit eine Schnittstelle dar, die nicht an eine Sprache gebunden ist und somit von vielen unterschiedliche Sprachen genutzt werden kann.

6.4 Realisierung in der Spieleroberfläche

Für die Diplomarbeit wurde eine besonders einfache Variante des RPCs verwendet, XML-RPC genannt. Die Verwendung der Programmiersprachen Java für die Spieleroberfläche und Ada für die Basismaschine hätte zwar die Verwendung von CORBA nahegelegt. Die Implementierung ist jedoch aufwändig und die Schnittstellenfunktionen der Basismaschine sind auf einfache Datentypen beschränkt. Dies ist darin begründet, dass die bisherige Anbindung zwischen der Basismaschine und der Spieleroberfläche eine Anbindung von Ada nach Tcl/Tk verwendet. Tcl/Tk ist eine untypisierte Skriptsprache, die lediglich Strings als Datentypen verarbeiten kann. Dadurch sind die Schnittstellenfunktionen auf die Datentypen Strings und Boolean beschränkt, da diese leicht zu portieren sind. Die vorteilhaften Konzepte von CORBA kämen daher nicht zu Tragen.

Bei XML-RPC werden die versendeten Daten in XML-Dateien verpackt und gesendet. Bei der Implementierung in Java muss lediglich eine Server- und eine Clientklasse erzeugt werden, die über HTTP miteinander kommunizieren. Alle Public-Methoden der Serverklasse gelten als Schnittstellenfunktionen und können von der entfernt laufenden Spieleroberfläche aufgerufen werden. Das Über-

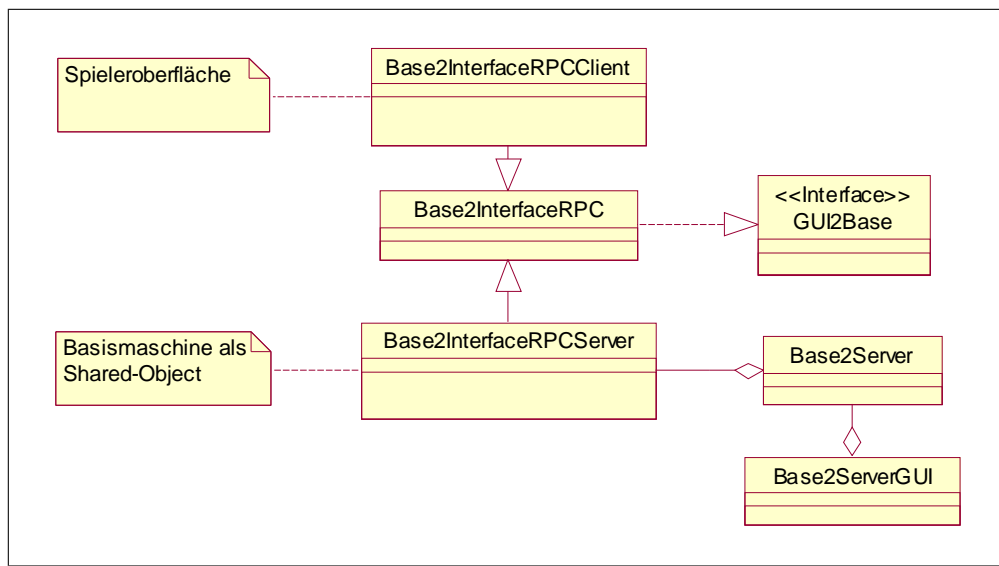


Abbildung 6.4: Entwurf der Schnittstellen bei verteiltem SESAM

setzen der Parameter in die XML-Datei übernimmt XML-RPC automatisch, so dass der Aufwand für eine verteilte Anwendung minimal ist. Server- und Clientschnittstelle wurden jeweils in Java implementiert, um zum einen den einfachen Austausch der Spieleroberfläche oder der Basismaschine zu gewährleisten, zum anderen, weil eine direkte Umsetzung von XML-RPC in Ada nicht unterstützt wird.

Eine Anbindung direkt an den C-Code, der die Ada-Funktionen anspricht, wäre zwar möglich gewesen, vom Implementierungsaufwand jedoch höher als die indirekte Anbindung von Java über C nach Ada. Es wurde so vorgegangen, dass die Java-Schnittstelle zwischen GUI und Basismaschine zweimal implementiert wurde. Einmal für die Serverseite und ein weiteres Mal für die Clientseite. Beide Schnittstellen stellen jeweils die Basismaschinenbefehle bereit, implementieren diese jedoch unterschiedlich. Somit ist gewährleistet, dass die GUI ebenso wie die Basismaschine lediglich die Funktionen der Schnittstelle aufrufen, so dass die Realisierung, wie die Schnittstellen miteinander kommunizieren, nicht relevant ist. Damit ist die Implementierung für alle Konzepte verteilten Arbeitens offen gehalten.

6.5 Risiken

Im Vergleich zu rein lokal ausgeführten Anwendungen gibt es jedoch nicht nur Vorteile.

1. Ausfall des Servers / Netzwerks

Ein Problem betrifft die Ausfallsicherheit im Netzwerk. Fällt ein Server aus, können die Spieler nicht weiterspielen, das Spiel ist ohne Sicherung eventuell sogar verloren. Im Gegensatz zu rein lokal ausgeführten Spielen kann dies gleich mehrere Spieler betreffen, so dass an dieser Stelle viel Aufwand in die Umsetzung des Servers gesteckt werden muss, um bei einem Ausfall eines Servers sofort ein lauffähiges System als Ersatz bereitzustellen. Dafür müsste der Zustand der Basismaschine redundant auf mehreren Rechnern mitgeführt werden, was zusätzlichen Verwaltungsaufwand bedeutet. Ein Ausfall hätte außer dem Verlust des Spielstands jedoch keine

weiteren Auswirkungen auf das SESAM-Spiel.

2. Sicherheitsaspekte

Die Sicherheit in einem verteilten System stellt immer ein Risiko dar, weil Benutzer von ihren Rechnern auf den Server zugreifen und bei mangelnder Sicherheit des Servers dort auch Schaden anrichten können. Dies bringt zusätzlichen Aufwand in der Implementierung und Wartung mit sich, da Sicherheitslücken vermieden werden müssen. Da jedoch die Eingabemöglichkeit des Spielers bei SESAM stark eingeschränkt ist, sollte der Sicherheitsaspekt für SESAM keine Rolle spielen.

3. Konsistente Zustände

Redundant gehaltene Daten müssen auf den verschiedenen Rechnern innerhalb des Systems in einem konsistenten Zustand gehalten werden. Bei einer strikten Trennung zwischen Darstellung und Datenhaltung sollte dieses Problem bei SESAM eher geringfügig sein.

4. Zusätzlicher Code

Die Wartbarkeit wird zwar durch die Trennung der Komponenten verbessert, jedoch muss berücksichtigt werden, dass ein verteiltes System mehr Code benötigt, der ebenfalls gewartet und angepasst werden muss. Für SESAM sollte dies nur eine untergeordnete Rolle spielen. Wenn eine Lösung für die Kommunikation zwischen Client und Server gefunden ist, sollten die Teile unabhängig voneinander gewartet werden können, ohne dass die Kommunikationsschnittstelle, die das verteilte Arbeiten ermöglicht, berührt werden muss. Lediglich bei einer Änderung der Schnittstelle zwischen GUI und Basismaschine müssen die Codeteile eventuell angepasst werden. Nach Einschätzung des Implementierungsaufwands der Prototypen, die für verschiedene Verteilungskonzepte gebaut wurden, sollte die Anpassung aber keinen allzu großen Aufwand verursachen.

Allerdings würde eine Umstellung der Basismaschine auf echte Mehrbenutzerfähigkeit weitreichende Veränderungen fordern, so dass der Implementierungsaufwand hierbei hoch sein dürfte. Hier muss vor allem bedacht werden, wie SESAM-Modelle geladen werden können. Wenn die Modelle auf der Spielerseite liegen sollen, so dass jeder Spieler die Modelle laden kann, die ihm lokal zur Verfügung stehen, müssen diese zunächst auf den Server übertragen werden. Da die Modelle zum Teil sehr groß sind, müssen diese vom Spieler auf die Basismaschine übertragen werden. Bei langsamen Netzwerkverbindungen kann dies zu langen Wartezeiten führen, so dass hier eventuell neue Konzepte erarbeitet werden müssen. Sollen die Modelle hingegen auf der Seite der Basismaschine, also auf dem zentralen Server, liegen, so muss die Basismaschine mindestens um eine Funktion erweitert werden, die an die Spieleroberflächen eine Liste aller verfügbaren Modelle sendet, so dass dem Spieler eine Auswahlliste angeboten werden kann.

6.6 Empfehlung

Nach Abwägen der Vor- und Nachteile einer verteilten Ausführung kann ich keine klare Aussage darüber treffen, ob die Implementierung einer solchen Verteilung tatsächlich sinnvoll und den Aufwand wert ist.

Einerseits bietet ein verteiltes Spiel, an dem mehrere Spieler gleichzeitig teilnehmen können, durchaus Vorteile bezüglich der Modellgestaltung. Die Spieler können sich gegenseitig beeinflussen und ihre Handlungen und deren Auswirkungen sind vermutlich näher an der Realität als die durch

einen Zufallsgenerator erzeugten Handlungen. Dennoch muss man sehen, dass in der langen Entwicklungszeit von SESAM überhaupt erst wenige Modelle entstanden sind und das noch viele Ideen für weitere Modelle, die nicht auf Mehrspielerfähigkeiten basieren, vorhanden sind. Der Entwicklungsbedarf für konventionelle Modelle ist also noch gross genug, so dass die Mehrbenutzerfähigkeit zwar eine wünschenswerte Erweiterung wäre, aber zum jetzigen Zeitpunkt nicht unbedingt notwendig erscheint. Die weiteren Vorteile zur Speicherminimierung und Entlastung sind eher zweitrangig. Man kann das Spiel komplett lokal laufen lassen, so dass ein Zwischenserver nicht mehr notwendig ist. Der Speicherverbrauch ist auch nicht so hoch, dass man spezielle Rechner dafür benötigen würde. Moderne Rechner haben mittlerweile soviel Kapazität an Rechenleistung und Speicher, dass man darüber nur noch am Rande nachdenken muss.

Insgesamt würde ich deshalb empfehlen, ein echtes verteiltes Spielen mit SESAM erst dann umzusetzen, wenn genug Zeit für die Implementierung vorhanden ist und Mehrspielermodelle tatsächlich auch erwünscht sind.

Kapitel 7

Zusammenfassung und Ausblick

Dieses Kapitel soll die Ergebnisse der Diplomarbeit zusammenfassen. Es beschreibt zudem die Erfahrungen, die ich während der Arbeit gemacht habe, sowie den Verlauf des gesamten Projekts. Am Ende wird noch ein Ausblick auf mögliche Erweiterungen der Diplomarbeit gegeben.

7.1 Zusammenfassung

Dieser Abschnitt zeigt, wie ich die Entwicklung der Spieleroberfläche und die weiteren Tätigkeiten der Diplomarbeit generell geplant habe. Dazu wird zunächst beschrieben, welche Phasen eine solche Entwicklung hat und wie diese gestaltet wurden.

7.1.1 Vorgehensweise

Die Entwicklung orientierte sich an gängigen Modellen, wie sie in der Softwaretechnik gelehrt werden. Generell gibt es während der Softwareentwicklung verschiedene Phasen, die durchlaufen werden müssen. Dazu gehören:

- Analysephase
- Spezifikationsphase
- Entwurfsphase
- Implementierungsphase
- Testphase
- Wartung

Die Wartung der Spieleroberfläche ist nicht Bestandteil der Diplomarbeit. Sie nimmt im Lebenslauf einer Software jedoch eine zentrale Rolle ein und musste damit zumindest soweit berücksichtigt werden, dass Entwickler der Wartungsphasen möglichst wenige Probleme mit dem Code haben.

Zu Beginn der Diplomarbeit wurde ein Projektplan erstellt, der inhaltliche Ziele mit Zeitpunkten in der Diplomarbeit verbunden hat. In der Analysephase wurden die Konzepte für eine graphische Benutzeroberfläche sowie Konzepte für ein verteiltes Arbeiten mit SESAM erarbeitet. Dazu wurde die bestehende GUI untersucht, um Konzepte dieser Version zu erfassen. Die weiteren Entwicklungsphasen der Softwareentwicklung wurden nicht linear durchlaufen. Hierfür wurde ein spezieller Entwicklungsprozess angewendet, der im folgenden Abschnitt beschrieben wird. Im Anschluss an die

Softwareentwicklung gab es noch eine Phase für die Ausarbeitung der Diplomarbeit, so dass die Entwicklung in drei große Blöcke unterteilt werden kann.

- Analysephase und Konzepterstellung des gesamten Projekts
- Iterative Softwareentwicklung
- Erstellung des Abschlussberichts

7.1.2 Wahl eines Entwicklungsprozesses

Die Wahl eines Entwicklungsprozesses stellt eine wichtige Aufgabe am Anfang der Softwareentwicklung dar. Dieser Abschnitt soll erläutern, was ein Softwareentwicklungsprozess ist und für welchen ich mich in dieser Diplomarbeit entschieden habe.

Als **Prozess** bezeichnet man (in diesem Zusammenhang):
Eine Abfolge von Schritten um einen Zweck zu erfüllen; Beispielsweise ein Softwareentwicklungsprozess.
Als **Softwareentwicklungsprozess** bezeichnet man:
Den Vorgang mit dem Benutzeranforderungen in ein Softwareprodukt übersetzt werden. Der Prozess umfasst das Umwandeln der Anforderungen in Softwareanforderungen, Umwandeln dieser in einen Entwurf der anschließend in Code übersetzt wird, das Testen dieses Codes und bei Bedarf das Installieren und Auschecken der Software für den Betrieb. Anmerkung: Diese Aktivitäten können überlappend oder nacheinander ausgeführt werden.

Abbildung 7.1: Definition von Softwareentwicklungsprozessen, IEEE Std. 610.12-1990

Ein Entwicklungsprozess beschreibt die Arbeitsschritte, mit denen man eine Software erstellen kann. Je nach Anwendung, Teamgröße, Aufgabe oder persönlichen Vorlieben gibt es mittlerweile viele Entwicklungsprozesse. Sie alle haben das Ziel, einem chaotischen Vorgehen in der Softwareentwicklung entgegenzuwirken. Ein Software-Prozessmodell ist demnach eine abstrakte Beschreibung ähnlicher Software-Prozesse, also eine Muster-Struktur für Organisation und Durchführung eines Software-Prozesses. Die offizielle Beschreibung von Prozessen nach IEEE ist in Definition 7.1 gegeben. Das oberste Ziel eines Software-Prozesses ist letztendlich die Minimierung von unnötigem Aufwand und damit die Kostenminimierung. Während der Diplomarbeit sind zwar keine direkten Kosten im Sinne von Geld vorhanden, eine Aufwandsminimierung ist aber schon deshalb notwendig, weil die definierten Ziele in einem sehr strikt gehaltenen Zeitrahmen von 6 Monaten realisiert werden müssen.

Das Spiralmodell

Die Wahl des Prozessmodells für die Diplomarbeit ergibt sich aus der Tatsache, dass die Implementierung zweigeteilt ist. Zum einen soll eine Oberfläche entstehen, die unterschiedliche Modelle darstellen kann, zum anderen soll ein entsprechendes Modell-Plug-in entwickelt werden, das diese Spieleroberfläche darstellen kann. Das Spiralmodell [Boe88] beschreibt die Idee, nicht die gesamte Software in

Spiralmodell

Ein Modell des Softwareentwicklungsprozesses, in dem die einzelnen Aktivitäten, typischerweise Anforderungsanalyse, Grob- und Feinentwurf, Codierung, Integration und Test, nacheinander wiederholend ausgeführt werden bis die Software fertiggestellt ist.

Abbildung 7.2: Definition des Spiralmodells, IEEE Std. 610.12-1990

einem Entwicklungsschritt zu erstellen. Vielmehr werden kleinere, besser überschaubare Teile entwickelt. Für jedes Teilsystem werden die normalen Entwicklungsschritte durchschritten, also von der Anforderungsanalyse bis zum Test. Am Ende wird das Teilsystem integriert, bevor anschließend das nächste Teilsystem entsteht. Für manche dieser Teilsysteme können zunächst auch Prototypen entstehen, um die Aufgabe besser abschätzen zu können. Die Aufteilung und der iterative Vorgang soll verhindern, dass die Komplexität der Aufgabe früher oder später die Entwicklung in eine Sackgasse führt und damit viel Geld kostet, oder, in diesem Fall, die Benotung der Diplomarbeit gefährdet. Daher wird beim Spiralmodell zusätzlich noch eine Risikoliste erstellt. Am Beginn der Entwicklung werden mögliche Risiken identifiziert und klassifiziert. Bei der Auswahl der Teilsysteme geht man nun so vor, dass das potentiell größte Risiko zuerst bewältigt wird. Tritt das Risiko ein, d.h. es wird zum Problem, hat man so noch die besten Chancen, gegenzusteuern. Führt das Problem gar zur Einstellung des Projekts, hat man so immerhin noch Geld gespart gegenüber einem Projekt, dass dieses Problem erst am Ende der Entwicklung identifiziert. Die Definition nach IEEE ist in Abbildung 7.2 gegeben.

Risikoliste

Durch die Risikoliste ergab sich die Reihenfolge, in der die Software-Teilsysteme entwickelt wurden. Die folgenden Punkte sind Risiken, die auftreten, falls einer der Punkte nicht realisierbar ist. Realisierbar bedeutet hier nicht nur, dass die Aufgabe generell lösbar ist, sondern dass sie in angemessener Zeit, d.h. mit einem sinnvollen Maß an Aufwand, bewältigt werden kann. Das erste Risiko stellt ein sehr hohes Risiko dar, während das letzte Risiko lediglich geringe Auswirkungen auf den Projektverlauf hat. Probleme treten auf, wenn folgende Punkte nicht umsetzbar sind:

1. Zugriff auf die SESAM-Basismaschine von Tcl/Tk losgelöst

Die bisherige GUI wurde in Tcl/Tk implementiert, die Basismaschine in Ada. Die Kommunikation der beiden Teilsysteme erfolgt über eine C-Schnittstelle. Das Risiko besteht nun darin, dass man die Basismaschine nicht so von der Oberfläche trennen kann, dass eine Java-GUI aufgesetzt werden kann. Dies ist beispielsweise bei Callback-Aufrufen denkbar, wenn der Ada-Code direkt auf Tcl-Code zugreifen würde.

2. Anbindung von Java an die SESAM-Basismaschine

Dieses Risiko tritt dann ein, wenn die Schnittstelle der Basismaschine bestimmte Formate, z.B. für die Parameterübergabe, vorgibt, diese aber von der Java-GUI nicht realisiert werden können. Auch die generelle Anbindung von Java an Ada fällt unter diesen Gesichtspunkt.

3. Parameterübergabe an die Basismaschine

Selbst wenn eine allgemeine Verbindung zwischen Java und Ada besteht, ist nicht sichergestellt,

dass die Parametertypen einfach konvertiert werden können.

4. **Umsetzung der graphischen Oberfläche mit Modellbezug**

Hier besteht die Gefahr, die Oberfläche nicht in der gewünschten Form realisieren zu können. Beispielsweise könnte die Implementierung der Bildelemente zu aufwändig werden.

5. **Implementierung des Raumkonzepts**

Das Raumkonzept stellt eine Erweiterung der Funktionalität des QS-Modells dar. Das Risiko besteht nun darin, dass die Umsetzung der Räume vom Modell nicht unterstützt wird oder sogar im Gegensatz dazu steht.

6. **Verteilte Ausführung**

Die Ausführung der GUI und der Basismaschine auf verteilten Rechnern ist zwar ein Risiko, dessen Eintreffwahrscheinlichkeit verhältnismäßig hoch ist. Die Auswirkungen wären jedoch eher gering, da die verteilte Ausführung nicht Kernthema dieser Diplomarbeit ist.

Maßnahmen bei Problemen

Jedem Risiko ist ein Maßnahmenpaket zugeordnet. So ist gewährleistet, dass bei Eintritt eines Risikos entsprechend schnell gehandelt werden kann, um Zeitverzug bei der Diplomarbeit entgegenzutreten. Für die oben genannten Risiken wurden daher folgende Maßnahmen getroffen:

- **Risiken 1-3**

Der Betreuer passt die Aufgabenstellung der Diplomarbeit an. Da das Problem in einer frühen Phase auftreten würde, könnte über den Neuanfang einer Diplomarbeit entschieden werden. Bestehen die Probleme lediglich in einem zu hohen Aufwand der Implementierung, so wird eine Beispielimplementierung für ausgewählte Teilsysteme entwickelt.

- **Umsetzung der graphischen Oberfläche mit Modellbezug**

Hier wird der Betreuer eine Anpassung der Diplomarbeit erwirken, in der eine Umsetzung des Modells möglich ist. Eventuell wird zunächst eine reine Textumsetzung, wie sie bereits in Tcl/Tk realisiert ist, entstehen. Somit würde die Implementierung nur eine Portierung vorsehen, die eine graphische Umsetzung des Modells offen hält.

- **Implementierung des Raumkonzepts**

Entsteht hierbei ein Problem, wird das Raumkonzept aus der Diplomarbeit herausgenommen oder auf eine mögliche Implementierung abgeändert.

- **Verteilte Ausführung**

Sollte die verteilte Ausführung Probleme bereiten, wird sie aus der Diplomarbeit herausgenommen.

7.1.3 Entwicklungsphasen

Die einzelnen Phasen während der Entwicklung der Diplomarbeit sind in Bild 7.3 auf der nächsten Seite dargestellt. Begonnen wurde mit der Schätzung des Projektaufwands. Dazu wurde das Projekt grob in Phasen unterteilt und darauf basierend eine Schätzung für den Aufwand einzelner Teilziele gegeben. Vorgegeben durch den Rahmen der Diplomarbeit war ein maximaler Zeitumfang von 6 Monaten. Legt man eine 40-Stunden-Woche zugrunde, kommt man auf einen Gesamtumfang von etwa 1040 Stunden. Da ich während der Diplomarbeit einer geringen Nebenbeschäftigung im Umfang von

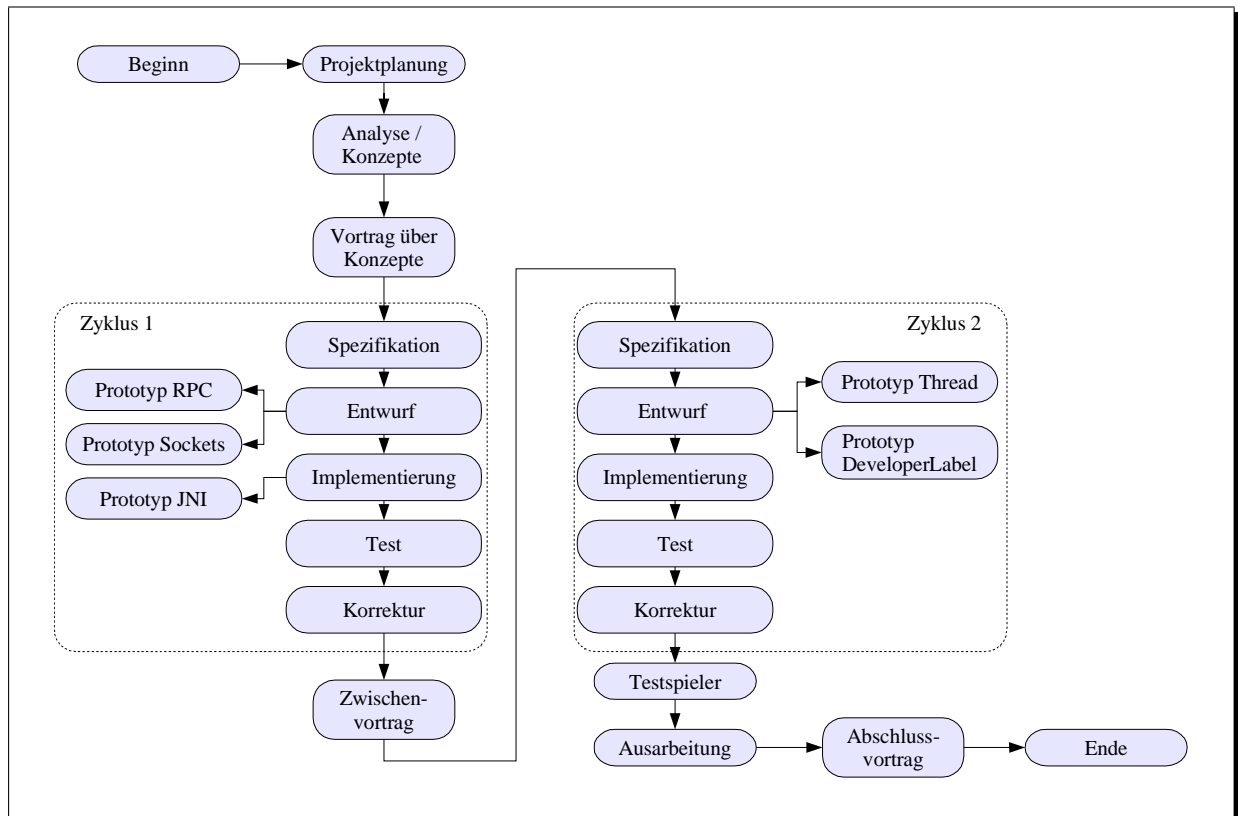


Abbildung 7.3: Die Phasen der Diplomarbeit

4 Stunden pro Woche nachging, blieben effektiv 936 Stunden übrig. Diese wurden auf die einzelnen Phasen aufgeteilt, wie in Bild 7.7 auf Seite 83 dargestellt. Um bei Problemen nicht in Zeitnot zu geraten, wurde jeweils eine Woche vor Weihnachten und eine am Ende der Diplomarbeit als Pufferzeit für eventuelle Probleme eingeplant. Gleichzeitig wurde noch eine Aufwandsschätzung mit dem Kostenschätzverfahren COCOMO durchgeführt. Die Ergebnisse waren aber nicht auf objektorientierte Software anwendbar und ergaben zu pessimistische Schätzungen.

Im zweiten Schritt der Entwicklung wurden Konzepte für eine neue Oberfläche erstellt. Die Ergebnisse wurden in einem Dokument zusammengefasst und in einer kurzen Präsentation den Mitarbeitern der Abteilung Software Engineering vorgestellt. In einer anschließenden Diskussion wurde beschlossen, welche Punkte des Konzeptplans umgesetzt werden. Aufgrund der Diskussion wurde der Konzeptplan nachträglich noch auf die Wünsche der Mitarbeiter abgeändert.

Nun folgte die erste Phase der eigentlichen Softwareentwicklung. Hier entstanden die ersten Teile der Spezifikation, des Entwurfs und Codes. Ferner entstanden mehrere Prototypen. In der Abbildung 7.3 sind die drei wichtigsten abgebildet. Zwei dieser Prototypen befassten sich mit der verteilten Ausführung von Java-Anwendungen, der dritte Prototyp diente als Beispiel für die Anbindung von Java- und Ada-Code. Weitere Prototypen, z.B. ein Prototyp zum Zeichnen von Bildern, wie sie in der späteren Entwicklung beispielsweise für die Mitarbeitergraphiken verwendet wurden, sind in der Abbildung nicht dargestellt, weil ihre Entwicklung meist nur kurz dauerte und oft lediglich zum Probieren von verschiedenen Konzepten diente.

Nachdem der erste Zyklus des Spiralmodells durchlaufen war, konnte man bereits einfache Spiele

durchführen. Der zweite Zyklus wurde bereits vor dem dafür vorgesehenen Termin begonnen, so dass zur Hälfte der Zeit für die Diplomarbeit bereits ein kleiner Vorgeschmack auf die Spieleroberfläche zu sehen war. Diese Ergebnisse wurden in einem Zwischenvortrag den Mitarbeitern der Abteilung Software Engineering vorgeführt. Die weitere Implementierung wurde nach einer etwa zweiwöchigen Pause (über Weihnachten) fortgeführt, so dass die zweite Entwicklungsphase bereits Anfang Februar beendet war.

Anschließend wurde mit der Ausarbeitung der Diplomarbeit begonnen. Parallel dazu hat ein Testspieler mit der neuen und alten Spieleroberfläche gespielt und Feedback zur Spielbarkeit und Korrektheit der neuen Spieleroberfläche gegenüber der bisherigen Referenzoberfläche geliefert. Nach Fertigstellung einer ersten Version der Ausarbeitung gab es einen zweiwöchigen Einschub, in dem die gesammelten Testresultate ausgewertet wurden. Danach wurde der Code nochmals komplett überarbeitet und vollständig kommentiert, sofern die Kommentare sich bis zu diesem Zeitpunkt nicht an die Vorgaben von Javadoc hielten oder unklar formuliert waren. Anschließend erfolgte ein Codereview mit dem Betreuer der Diplomarbeit. Bevor die Ausarbeitung weiter bearbeitet wurde, wurde sie in der Zeit der Codeüberarbeitung von mehreren Personen korrektur gelesen.

Die letzte Phase des Projekts war leicht chaotisch, was weniger an Zeitmangel als vielmehr an der Tatsache lag, dass in dieser Zeit der Umzug der Fakultät an den Campus stattfand und zeitweise weder Räume noch Rechner zur Verfügung standen. Alles in allem lief die Entwicklung aber reibungslos.

7.1.4 Aufwand und Ergebnisse der Arbeit

Arbeitspaket	September	Oktober	November	Dezember	Januar	Februar	März	April		
Projektplan										
Konzepte										
Spezifikation 1										
Entwurf 1										
Implementierung 1										
Test 1										
Korrektur										
Zwischenvortrag										
Spezifikation 2										
Entwurf 2										
Implementierung 2										
Test 2										
Korrektur										
Abschlussbericht										

Abbildung 7.4: Zeitplan geplant

Während der gesamten Diplomarbeit wurde genau über die geleistete Arbeit und die dafür benötigte Arbeitszeit Buch geführt. Es wurde jeweils auf die Viertelstunde genau abgerechnet. Zur Arbeitszeit wurde lediglich die tatsächliche Zeit, die an der Diplomarbeit gearbeitet wurde, gerechnet. Pausen wurden nicht mitgerechnet.

In Tabelle 7.6 auf der nächsten Seite sind Aufwand und Ergebnisse für die jeweiligen Projektphasen aufgeführt. Bei Dokumenten wird jeweils die Anzahl der DIN-A 4 Seiten angegeben, bei Code die entstandenen Simple Lines of Code, also Codezeilen inklusive Kommentaren und Leerzeilen. Die Dokumente der Software entstanden durch die iterative Vorgehensweise nicht auf einmal. Die Seitenzahl sowie der Aufwand bezieht sich daher lediglich auf das Endprodukt des Dokuments.

Tabelle 7.7 auf der nächsten Seite gibt einen Überblick über den geplanten Aufwand im Vergleich zum tatsächlichen Aufwand für die Phasen. Man kann sehen, dass sich die gesamte Planung um einige Tage verschoben hat. Allerdings überraschenderweise nicht in die „falsche“ Richtung. Die gesamte

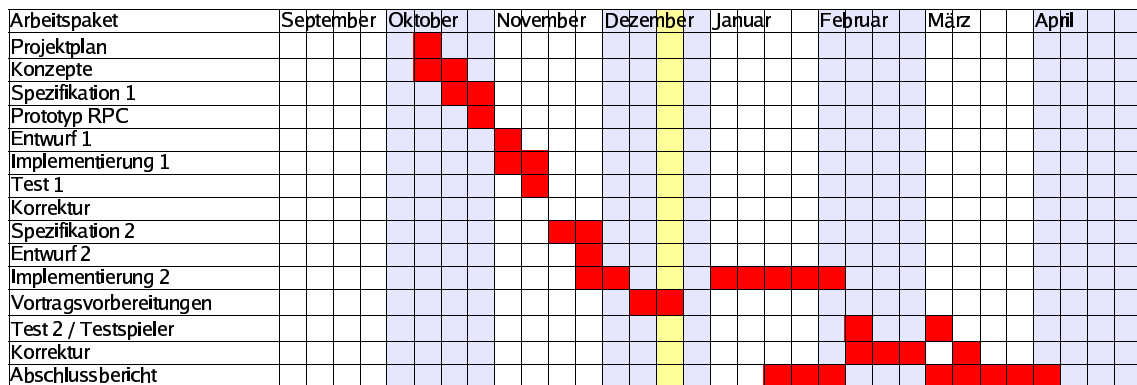


Abbildung 7.5: Zeitplan real

Phase	Aufwand IST	Ergebnisse
Vorarbeit	45	Arbeitsplatz eingerichtet, Programme installiert
Projektplanung	45	Projektplan, 22 Seiten
Konzepte	70	Konzeptplan: 31 Seiten, Kurzvortrag: 7 Folien
Spezifikation	80	Spezifikation, 25 Seiten
Entwurf	60	Entwurf: 15 Seiten, Zwischenvortrag: 17 Folien
Implementierung	250	Schnittstelle/Server: 950 SLOC, GUI/Plugin: 8900 SLOC
Test	20	
Ausarbeitung	270	Ausarbeitung, 104 Seiten
Gesamt	840	197 Seiten, 9850 SLOC, 24 Folien

Abbildung 7.6: Aufwand und Ergebnisse der Phasen

Phase	Aufwand IST	Aufwand SOLL	Anteil IST	Anteil SOLL	Abweichung total	Abweichung anteilig
Vorarbeit	45	60	5,4%	6,8%	-15	-1,4%
Projektplanung	45	50	5,4%	5,6%	-5	-0,3%
Konzepte	70	80	8,3%	9,0%	-10	-0,7%
Spezifikation	80	120	9,5%	13,5%	-40	-4,0%
Entwurf	60	80	7,1%	9,0%	-20	-1,9%
Implementierung	250	200	29,8%	22,6%	50	7,2%
Test	20	56	2,4%	6,3%	-36	-3,9%
Ausarbeitung	270	240	32,1%	27,1%	30	5,1%
Gesamt	840	886			-46	-5,48%

Abbildung 7.7: Soll- und Ist-Aufwandsverteilung

Entwicklung wurde früher beendet als ursprünglich geplant.

Ich bin bei der Planung der Diplomarbeit davon ausgegangen, dass viel Aufwand in der Implementierung stecken wird, da ich vor der Diplomarbeit noch nie mit der Programmiersprache Java

gearbeitet habe. Somit konnte ich die Möglichkeiten der Sprache nur wenig abschätzen und plante eher mehr Zeit für diese Phase ein. Tatsächlich stellte sich heraus, dass Java einfach zu erlernen ist und zudem viele Bibliotheken zur Verfügung stellt, die Lösungen für schwierige und häufig auftretende Probleme bieten. Die Bibliotheken sind gut dokumentiert, was das Auffinden der benötigten Klassen und Methoden vereinfachte.

Die Phasen Implementierung und Ausarbeitung liegen dennoch deutlich über dem geplanten Aufwand. Ursprünglich war geplant, einen Teil der im Vorfeld erarbeiteten Konzepte zu implementieren, so dass ersichtlich wird, in welche Richtung eine graphische Oberfläche für SESAM gehen kann. Nachdem die vorangehenden Phasen jedoch rasch durchlaufen werden konnten, wurde dieser Prototyp weiter entwickelt, so dass eine spielbare Version der Spieleroberfläche entstand. Allerdings wurde nicht die gesamte überschüssige Zeit auf die Implementierung gelegt. Der größte Teil der Zeit wurde auf die Ausarbeitung der Diplomarbeit vergeben, um hier nicht in Zeitnot zu geraten. Daher wurde die Ausarbeitung verhältnismäßig früh begonnen. Die Zeit, die nach der Fertigstellung des Dokuments blieb, wurde auf Tests und Bugfixes der Spieleroberfläche sowie eine umfangreiche Codekommentierung gelegt, so dass am Ende eine gut vorführbare Version des Programms entstand, bei dem auch der Code für die Wartungsarbeiten gut lesbar ist.

7.1.5 Untersuchung des Codes

Der Code für die Anbindung der Java-GUI an die Basismaschine und der Code der Plug-ins unterscheiden sich deutlich und sollen deshalb getrennt betrachtet werden. Der Anbindungscode lässt sich wiederum in drei Teile gliedern:

- **Ada-Code**
- **C-Code**
- **Java-Code**

	Ada95	C	Java-GUI	QS-Plugin	Gesamt
Simple Lines of Code	521	398	2264	6668	9851
Codezeilen mit „;“ abgeschlossen	195	118	790	3539	4642
Unkommentierte Zeilen	478	189	1016	4813	6496
Kommentarzeilen	43	209	1248	1855	3355

Abbildung 7.8: Analyse des Codes

Alle Codezeilen wurden „von Hand“ implementiert, das heißt ohne Hilfe von Generatorprogrammen, wie sie bei der GUI-Programmierung häufig eingesetzt werden. Der Grund hierfür ist die bessere Kontrolle über die Struktur des Codes, da ich nicht davon ausgehen kann, dass spätere Entwickler während der Wartung alle mir zugänglichen Werkzeuge einsetzen können. Außerdem ist die Anbindung von Java-Code über ein C-Modul an Ada eher ungewöhnlich, so dass man davon ausgehen kann, dass nicht jedes Programm eine solche Anbindung überhaupt unterstützen kann. Die Betrachtung von Source Lines of Code (SLOC) soll im Folgenden einen Einblick in die Produktivität der Entwicklung geben. Man sollte die Ergebnisse jedoch nicht überschätzen. Die SLOC nur dann etwas sinnvolles über die Produktivität aus, wenn sie mit anderen Projekten verglichen werden. Dafür

müssen die Projekte zumindest ähnlich sein, was Aufgabe, Programmiersprache, Teamgröße usw. angeht. Da dieser Diplomarbeit ähnliche Projekte nicht zum Vergleich bereitstanden, werden die Zahlen *aus dem Bauch heraus* interpretiert. Dazu kommt die Tatsache, dass SLOC von verschiedenen Programmiersprachen verglichen werden, wobei zwischen Ada, Java und C sogar ein grundsätzlich unterschiedliches Sprachkonzept besteht.

Die Analyse des Codes wurde mit den Programmen Slocount und Clc durchgeführt, welche die Gesamtzeilen des Codes, die kommentierten Zeilen, Leerzeilen und Zeilen, die mit „;“ enden ausgeben. Letztere sollen einen Überblick über die echten Codezeilen geben, da in der Regel eine Zuweisung oder Methodenaufruf mit einem Semikolon abgeschlossen wird. So können Codezeilen, die sich nur der Übersichtlichkeit halber auf mehrere Zeilen erstrecken, herausgerechnet werden.

Ein Maß für die Produktivität kann nun angegeben werden, indem man die entstandenen Codezeilen auf die Entwicklungszeit umrechnet. So bekommt man folgende Rechnung für die Produktivität während der Implementierungsphase:

$$9800 \text{ SLOC} / 250 \text{ h} = 39,2 \text{ SLOC} / \text{h}$$

Oft wird die Produktivität auf die gesamte Projektdauer gemessen. Damit ergibt sich folgender Wert.

$$9800 \text{ SLOC} / 840 \text{ h} = 11,67 \text{ SLOC} / \text{h}$$

Bei den Werten ist jedoch Vorsicht angebracht. Zum einen handelt es sich hierbei lediglich um Mittelwerte. Tatsächlich gab es große Schwankungen bei der Implementierung. Zum anderen sagen diese Werte wenig über die tatsächliche Arbeitsbelastung und Schwierigkeit der Teillösungen aus. Da ich alleine an der Implementierung gearbeitet habe, kann man die Produktivität auch nicht als Mittelwert für größere Teams heranziehen, die eventuell später an der Wartung beteiligt sind. Dennoch denke ich, dass die Werte in einem „normalen“ Bereich liegen. Die Erkenntnis aus diesen Werten ist also die, dass die Implementierung weder besonders einfach noch außergewöhnlich anspruchsvoll war. Es entstanden keine besonderen Schwierigkeiten, die so schwer lösbar waren, dass sie die Produktivität stark eingeschränkt hätten.

7.1.6 Bewertung der Ergebnisse

Die Implementierung der GPI wurde soweit fertiggestellt, dass ein Spiel des QS-Modells damit möglich ist. Das graphische Feedback ist jedoch nur vereinzelt realisiert, da die Entwicklung geeigneter Graphiken viel Zeit in Anspruch genommen hätte. Der Code besitzt auch noch einige kleine Unschönheiten, die sich hauptsächlich auf mangelnde Konfigurierbarkeit von Optionen beziehen. Der größte Mangel der neuen Spieleroberfläche ist in der Satzerkennungsfähigkeit des Interpreters zu sehen. Trotz der Stichworterkennung ist ein Spielen von SESAM ohne Liste der akzeptierten Stichworte kaum möglich. Die Implementierung lässt zwar eine Erweiterung des Sprachschatzes auf einfache Weise zu, generell sollte aber über eine neue Lösung für dieses Problem nachgedacht werden. Die Zeit für die Diplomarbeit ließ eine bessere Umsetzung des Interpreters nicht zu, da dieser nicht Bestandteil der Aufgabenstellung war und somit nur deshalb implementiert wurde, da ein Spielen von SESAM sonst nicht möglich gewesen wäre. Hier sehe ich den größten Handlungsbedarf für eine Anpassung der Spieleroberfläche. Erst eine sinnvolle und leicht erweiterbare Lösung für einen Interpreter der Spielereingaben kann das SESAM-Spiel gut bedienbar machen.

Nach Fertigstellung des Codes zeigte sich auch, dass das Document-View Prinzip für die Plug-ins durchgängig angewendet wurde, für die Darstellung der restlichen Code-Teile der Spieleroberfläche jedoch nicht, obwohl dies auch hier zum Teil sinnvoll gewesen wäre. Da diese Teile der Oberfläche jedoch recht klein und überschaubar sind, sollte der Nachteil nicht so gravierend sein, dass er die

Wartung erschwert. Ein Überarbeiten dieses Codes und die Umstellung auf Document-View sollte ebenfalls leicht möglich sein, sofern ein einheitliches Konzept der Implementierung gewünscht ist.

Es wurde daher besonders darauf geachtet, dass der Code gut verständlich ist. Das bedeutet vor allem, dass er gut kommentiert ist, damit spätere Entwickler bei der Wartung wenig Probleme haben. Alle Kommentare halten sich an die Konventionen des Dokumentationswerkzeuges Javadoc und des Code-Styleguides der Firma Sun, so dass eine Übersicht über die Klassen automatisch aus dem Code heraus erzeugt und als Webseite verfügbar gemacht werden kann.

7.1.7 Offene Punkte

Obwohl das Spielen von SESAM mit der neuen Oberfläche möglich ist, sind einige Punkte aus Zeitgründen nur prototypenhaft entstanden. Das betrifft im Wesentlichen die Ausgaben der Nachrichten und den Interpreter. Die Nachrichten werden derzeit in reiner Textform ausgegeben. Nur bei Beginn und Ende der Spezifikation wird zusätzlich das Dokument graphisch im Bild angezeigt. Diese Form der Darstellung sollte für alle Dokumente umgesetzt werden, was aber den Aufwand erheblich nach oben gesetzt hätte und somit nicht in angemessener Zeit bewerkstelligt werden konnte.

Der Interpreter sollte, wie erwähnt, neu implementiert werden. Eine Spracherkennung, die den Satzbau eines natürlichsprachlichen Satzes völlig ignoriert und statt dessen nur auf Stichworte reagiert, wird schnell komplex, wenn es darum geht, viele Eingaben des Benutzers zu akzeptieren und korrekt zu interpretieren. Das Problem besteht zum einen darin, dass auch ein Stichwort-basierter Interpreter zwischen Verben und Objekten in einem Satz unterscheiden muss, um herauszufinden, welche Aktionen mit welchen Objekten ausgeführt werden sollen. Beispielsweise sind Spielereingaben wie `Code` doppeldeutig und müssen entsprechend behandelt werden. Zusätzlich ergeben sich Probleme bei Über- und Unterspezifizierung von Sätzen. Die natürliche Sprache macht Annahmen über Randbedingungen, die ein Rechner nicht ohne weiteres nachvollziehen kann. Ebenso ist es sinnvoll, mehrere Kommandos zu bündeln. Ein Beispiel hierfür wäre:

```
hire Bernd and Axel
```

Bei solchen Kommandos ist der jetzige Interpreter der Spieleroberfläche chancenlos. Der prinzipielle Vorgang, nur Stichworte zu erkennen, ist sinnvoll, allerdings sollte die Umsetzung in Hinblick auf eine einfache und effiziente Lösung überarbeitet werden. Auch ein Nachfragen bei unvollständigen oder überspezifizierten Eingaben fehlt in dieser Implementierung völlig, da der Aufwand für die Erkennung dieser Spielereingaben nach ersten Ansätzen zur Implementierung zu hoch erschien.

7.2 Ausblick

Dieser Abschnitt gibt Anregungen für Verbesserungen von Eigenschaften, die mir während der Entwicklung und Arbeit mit SESAM aufgefallen sind.

7.2.1 Verbesserung der interaktiven Möglichkeiten

Die Diplomarbeit bot nicht die Zeit, alle Möglichkeiten für eine graphische Benutzeroberfläche auszureizen. Im Spiel fiel besonders auf, dass der Spieler immer noch wenig mit der Darstellung des Modellzustands interagieren kann. Die Oberfläche ähnelt stark typischen Textadventurespielen, bei denen der Hauptteil die Darstellung von Text ist und weniger die Darstellung von Zuständen mittels

graphischer Repräsentation. Hier sehe ich für die zukünftige Entwicklung noch Bedarf, die Textdarstellungen noch stärker (sofern dies Sinn ergibt) auf eine graphische Ausgabe zu beschränken. Beispielsweise könnte die Nachricht, dass ein Mitarbeiter angefangen hat zu spezifizieren, vollständig ersetzt werden, in dem dieser Sachverhalt durch die Darstellung der Spezifikation auf dem Tisch des Mitarbeiters umgesetzt wird.

Hilfreich wäre in diesem Zusammenhang auch die Einbindung von Animationen in der Szene, mit der dynamische Vorgänge wie das Bearbeiten von Dokumenten für den Spieler leichter erkennbar beschrieben werden könnten.

7.2.2 Entwicklung neuer Plug-ins

Neben dem QS-Modell gibt es weitere Modelle, mit denen SESAM gespielt werden kann. Dazu zählt beispielsweise eine feingranulare Variante des QS-Modells [Ham01] sowie ein Modell, das die Motivation der Mitarbeiter berücksichtigt. Für diese sowie für zukünftige Modelle sollten natürlich ebenfalls Plug-ins entwickelt werden, welche den Zustand des Modells jeweils graphisch repräsentieren.

7.2.3 Halbautomatische Generierung von Plug-ins

Die Erstellung von Plug-ins nimmt viel Zeit in Anspruch, die zusätzlich zur aufwändigen Erstellung von Modellen anfällt. Da viele Modelle jedoch untereinander recht ähnlich sind, wäre eine halbautomatische Stubgenerierung von Plug-ins denkbar. Dieser Generator könnte beispielsweise die Klassen für Räume und Kommandoerkennung automatisch aus der Modellbeschreibung ableiten, so dass der Entwickler der Plug-ins lediglich die Informationen neu implementieren muss, die nicht im Modell beschrieben sind, z.B. die Zuordnung von Kommandos zu Räumen.

7.2.4 Verteilte Ausführung und echte Mehrbenutzerfähigkeit

Die verteilte Ausführung hat durch die klare Schnittstelle zwischen GUI und Basismaschine in Zukunft eine große Chance, tatsächlich realisiert zu werden. Dazu müsste die Basismaschine allerdings noch angepasst werden. Zum einen wäre es wünschenswert, wenn die Basismaschine das Hochsprachenmodell nicht nur innerhalb des eigenen Systems laden könnte, sondern auch Daten von anderen Rechnern nutzen könnte. So wäre es möglich, dass die Spieler ihre Modelle lokal auf ihren Rechnern haben und diese an die Basismaschine schicken können. Weiterhin sollte es möglich sein, innerhalb der Basismaschine mehrere Zustände gleichzeitig zu führen, um mehreren Spielern die Möglichkeit zu geben, mit der Basismaschine zu kommunizieren, ohne dass für jeden Spieler ein eigener Prozess gestartet werden müsste.

7.2.5 Feedbackunterstützung der Basismaschine

Wünschenswert ist auch eine bessere Feedbackunterstützung der Basismaschine. Vor allem beim Laden eines Modells, das viel Zeit in Anspruch nimmt, wäre eine einfache Rückmeldung über den Ladezustand wünschenswert, so dass der Spieler nicht das Gefühl hat, die Applikation sei „abgestürzt“.

Abschließend bleibt zu bemerken, dass SESAM auch dazu dient, möglichst vielen Studenten die Ausbildung zum Projektleiter zu erleichtern. Je mehr Spieler an SESAM üben, desto genauer kann man auch die Modelle abstimmen und desto mehr Untersuchungsergebnisse von Testspielern wird es geben. Daher ist es in meinen Augen auch notwendig, nicht nur den wissenschaftlichen Charakter der Anwendung zu pflegen, sondern auch einen Schritt in Richtung verkaufbares Produkt zu wagen, um SESAM für viele Spieler attraktiv zu machen.

Literaturverzeichnis

- [Aus74] AUSUBEL, DAVID PAUL: *Psychologie des Unterrichts*. Beltz, 1974.
- [Bal96] BALZERT, HELMUT: *Lehrbuch der Softwaretechnik*, Kapitel 2.21, 2.22, 2.23. Spektrum akademischer Verlag, 1996.
- [Bal01] BALZERT, HEIDE: *UML kompakt*. Spektrum akademischer Verlag, 2001.
- [BCL⁺02] BESSERIE, KATIE, IRINA CAEPARU, JONATHAN LAZAR, JOHN ROBINSON und BEN SHNEIDERMAN: *Social and Psychological Influences on Computer User Frustration*. Technischer Bericht, University of Maryland, 2002.
- [BJR01] BOOCH, GRADY, IVAR JACOBSEN und JAMES RUMBAUGH: *OMG Unified Modeling Language Specification*. Object Management Group, Inc., 2001.
- [Boe88] BOEHM, BARRY W.: *A Spiral Model of Software Development and Enhancement*. Computer, 21(5):61–72, 1988.
- [Bru73] BRUNER, JEROME S.: *Der Akt der Entdeckung*. In: *Der Prozess der Erziehung*. H. Neber, 1973.
- [Dei94] DEININGER, MARCUS: *SESAM und die Lehre*. In: *SESAM Software Engineering Simulation durch animierte Modelle*. Jochen Ludewig, 1994.
- [Dra94] DRAPPA, ANKE: *SESAM und die Realität*. In: *SESAM Software Engineering Simulation durch animierte Modelle*. Jochen Ludewig, 1994.
- [Dra99] DRAPPA, ANKE: *Quantitative Modellierung von Softwareprojekten*. Doktorarbeit, Universität Stuttgart, 1999.
- [Fie91] FIELD, PAUL: *Object-oriented Programming in C*. C Vu, 4, 1991.
- [FLS91] FRÜHAUF, KAROL, JOCHEN LUDEWIG und HELMUT SANDMAYR: *Software-Projektmanagement und -Qualitätssicherung*. Teubner-Verlag, 1991.
- [GHJV94] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [Ham01] HAMPP, TILMANN: *Eine feingranulare SESAM-Variante*. Diplomarbeit, Universität Stuttgart, 2001.
- [KND⁺02] KING, PETER, PATRICK NAUGHTON, MIKE DEMONEY, JONNI KANERVA, KATHY WALRATH und SCOTT HOMMEL: *Java Code Conventions*. Sun Microsystems.Inc., 2002.

- [Li94] LI, JINHUA: *SESAM als Simulator*. In: *SESAM Software Engineering Simulation durch animierte Modelle*. Jochen Ludewig, 1994.
- [Lud94] LUDEWIG, JOCHEN: *SESAM: Grundidee und Überblick*. In: *SESAM Software Engineering Simulation durch animierte Modelle*. Jochen Ludewig, 1994.
- [Mel98] MELCHISEDECH, RALF: *Entwurf der Basismaschine*. Universität Stuttgart, Abteilung Software Engineering, 1998.
- [Mil56] MILLER, GEORGE A.: *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*. The Psychological Review, 63:81–97, 1956.
- [MR98] MELCHISEDECH, RALF und RALF REISSING: *Anforderungsspezifikation für die Basismaschine*. Universität Stuttgart, Abteilung Software Engineering, 1998.
- [Opf02] OPFERKUCH, STEFAN: *Eine Untersuchung zum Einsatz komplexer Simulationsmodelle in der Projektmanagementausbildung*. Diplomarbeit, Universität Stuttgart, 2002.
- [Rei96] REISSING, RALF: *Konzeption und Realisierung einer Basismaschine für SESAM-2*. Diplomarbeit, Universität Stuttgart, 1996.
- [Sch94] SCHNEIDER, KURT: *SESAM: Die konzeptionelle Basis*. In: *SESAM Software Engineering Simulation durch animierte Modelle*. Jochen Ludewig, 1994.
- [Sch02a] SCHMIDBERGER, RAINER: *Diplomarbeitsausschreibung: Entwurf und Implementierung einer für Projektleiter realitätsnahen Simulationsoberfläche für SESAM*, 2002.
- [Sch02b] SCHWEIKHARDT, WALTRAUD: *Rechnerunterstütztes Lehren und Lernen*, 2002. Skript zur Vorlesung.
- [Shn97] SHNEIDERMAN, BEN: *Designing the User Interface - Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 1997.
- [SMM99] SCHNEIDER, BERND, MARC PHILIPP MESSNER und ULI MÜLLER: *Modellbeschreibungssprache für SESAM-2*. Technischer Bericht, Universität Stuttgart, 1999.
- [Spi99] SPIEGEL, ANDRE: *Konstruktion eines Dolmetschers*. Diplomarbeit, Universität Stuttgart, 1999.
- [Sta73] STACHOWIAK, HERBERT: *Allgemeine Modelltheorie*. Springer Verlag, 1973.
- [Tan94] TANENBAUM, ANDREW S.: *Moderne Betriebssysteme*. Addison-Wesley, 1994.
- [Vog02] VOGEL, ULRICH: *Netzwerkprogrammierung mit Sockets und C*, 2002.

Anhang A

Gesamtübersicht des Entwurfs

Dieser Anhang soll die Zusammenhänge und den Aufbau der Klassen der Spieleroberfläche darstellen. Dazu werden auf den folgenden Seiten für die einzelnen Subsysteme UML-Diagramme abgebildet.

A.1 Verzeichnisstruktur der Spieleroberfläche

Nachfolgend ist die Verzeichnisstruktur der Spieleroberfläche dargestellt. Die Pakete sind dabei *kursiv* gesetzt.

- ❖ *base2interface*
 - ❖ Base2Interface.java
 - ❖ Base2InterfaceFake.java
 - ❖ Base2InterfaceRPC.java
 - ❖ GUI2Base.java
- ❖ *configuration*
 - ❖ AppProperties.java
- ❖ *documentview*
 - ❖ Document.java
 - ❖ View.java
- ❖ *gui*
 - ❖ Application.java
 - ❖ GameOptionDialog.java
 - ❖ MainFrame.java
 - ❖ MainMenu.java
 - ❖ SwingWorker.java
- ❖ *plugin*
 - ❖ Plugin.java

- ❖ Interpreter.java
- ❖ *qaplugin*
 - ❖ QADocument.java
 - ❖ QAControl.java
 - ❖ MultiValueHashtable.java
 - ❖ QAMenu.java
 - ❖ QAPlugin.java
- ❖ *rooms*
 - ❖ DeveloperRoom.java
 - ❖ DeveloperRoomView.java
 - ❖ ProjectManagerRoom.java
 - ❖ ProjectManagerRoomView.java
 - ❖ Cafeteria.java
 - ❖ CafeteriaView.java
 - ❖ Room.java
- ❖ *interpreter*
 - ❖ CommandInterpreter.java
 - ❖ CommandValidator.java
 - ❖ MessagePlayerSorter.java
 - ❖ MessageSplitter.java
- ❖ *configuration*
 - ❖ DeveloperLabel.java
 - ❖ PositionedIcon.java
 - ❖ QAErrorMessages.java
 - ❖ QAMessages.java
 - ❖ QAPictures.java
 - ❖ QAProperties.java

A.2 Übersicht über die Klassen und Pakete

Nachfolgend ist eine Übersicht über alle Pakete und Klassen der neuen Spieleroberfläche gegeben. Aus Gründen der Übersichtlichkeit sind Teilsysteme dargestellt, da die gesamte Architektur zu unübersichtlich wäre.

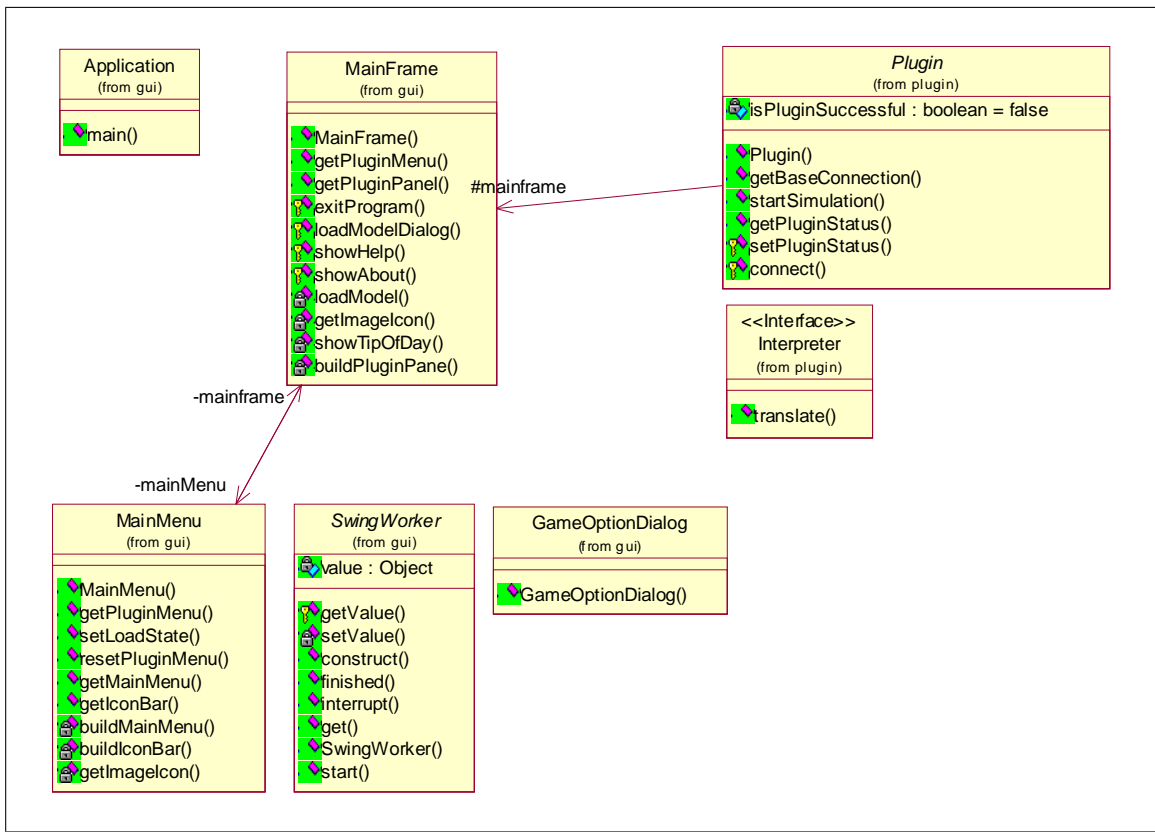


Abbildung A.1: Überblick über den Rahmen der Spieleroberfläche (ohne Plug-ins)

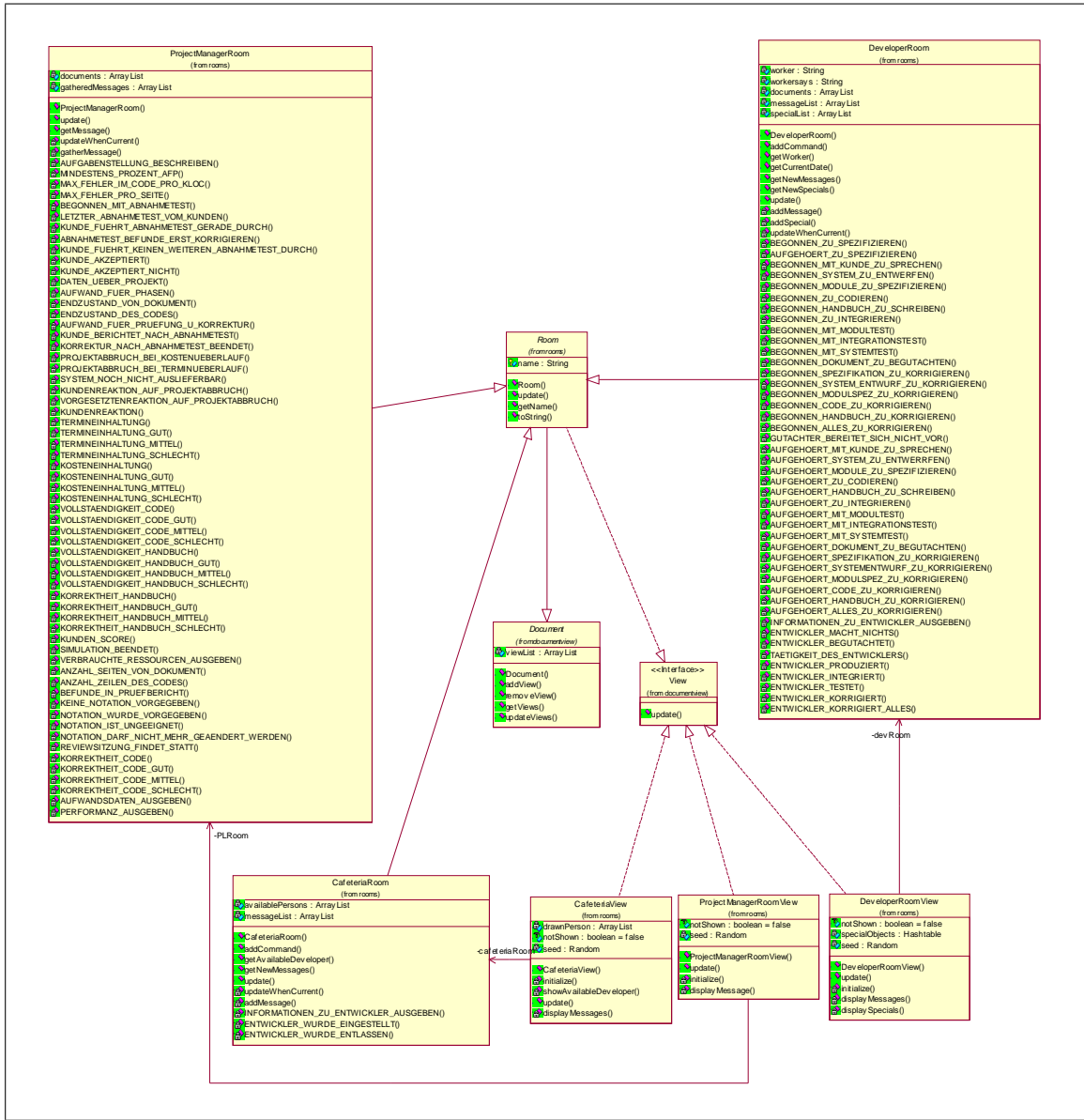


Abbildung A.3: Überblick über die Raumklassen

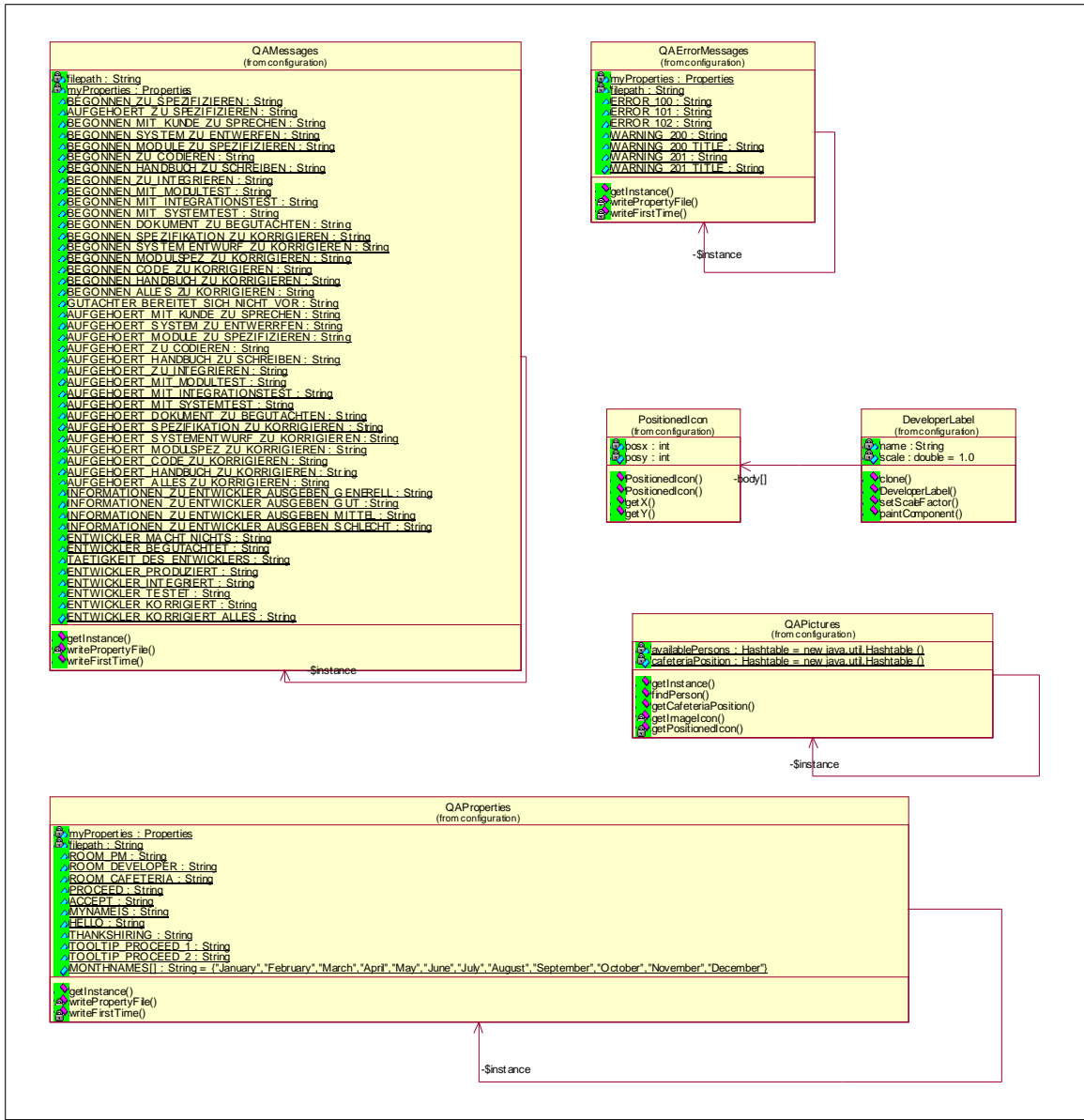


Abbildung A.4: Überblick über die Konfigurationsdateien des QS-Modell-Plug-ins

Anhang B

Begriffslexikon

Dieses Begriffslexikon stellt eine Sammlung von Begriffen dar, die während der Diplomarbeit verwendet wurden. Ein großer Teil der Definitionen stammt aus dem Begriffslexikon des SESAM-Projekts. Manche dieser Begriffe wurden in dieser Diplomarbeit anders verwendet und sind daher zum Teil mit mehreren Definitionen versehen.

Ada Eine Programmiersprache. In dieser Ausarbeitung ist mit Ada immer Ada95, die objektorientierte Version der Sprache, gemeint.

Aktivität Eine Aktivität steht für eine übergeordnete Tätigkeit im SESAM-Modell, zum Beispiel die Erstellung der Spezifikation.

Attribut Eigenschaften einer Klasse. Auch Instanzvariablen genannt.

Ausführungswerkzeug Ein Werkzeug, mit dem der Spieler ein SESAM-Modell ausführen kann. In SESAM-1 der Simulator, in SESAM-2 die Kombination aus Basismaschine und Dolmetscher.

BASE-2 Sprache der Basismaschine. BASE-2 steht für **Basissprache** für **SESAM-2**.

Basismaschine Werkzeug zur Ausführung von in der Basissprache geschriebenen Modellbeschreibungen.

Basissprache Modellbeschreibungssprache der Basismaschine. Sie ist Zielsprache bei der Übersetzung einer Hochsprache. Die Basissprache besitzt im Vergleich zur Hochsprache wenige Konzepte, da sie primär ausführungsorientiert ist.

Benutzer Das SESAM-System verfügt über drei verschiedene Benutzerklassen. Das sind die Modellierer, die Spieler und die Tutoren. Benutzer ist der gemeinsame Oberbegriff.

Benutzerkommando Ein vom Spieler eingegebener Befehl, der für die Verwendung in der Basismaschine noch in die Basissprache übersetzt werden muss.

Benutzeroberfläche Eine Eingabemaske für die Interaktion eines Anwenders mit der Applikation.

Benutzerschnittstelle Siehe Benutzeroberfläche.

Betreuer Im Zusammenhang mit SESAM-2 auch Tutor genannt. In dieser Diplomarbeit ist mit Betreuer jedoch ausschließlich der Betreuer der Diplomarbeit, Rainer Schmidberger, gemeint.

Button Widgets, die Schaltflächen darstellen, beispielsweise Pushbuttons, Radiobuttons oder Checkbuttons.

C Eine konventionelle Programmiersprache.

C++ Weiterentwicklung der Programmiersprache C. C++ unterstützt Objektorientierung.

Dokument Oberbegriff für schriftlich und elektronisch vorliegende Unterlagen eines Softwareprojekts. Dies umfasst neben weiteren Unterlagen beispielsweise die Spezifikation, den Entwurf und den Code des Programms.

Dolmetscher Ausführungswerkzeug, das zwischen dem Spieler und der Basismaschine vermittelt. Nimmt Eingaben des Spielers entgegen und wandelt sie in Kommandos an die Basismaschine um. Umgekehrt schickt die Basismaschine Nachrichten an den Dolmetscher, der sie in Ausgaben an den Spieler übersetzt. Zur Vermittlung bedient sich der zur Zeit verfügbare Dolmetscher eines Wörterbuchs. Der Dolmetscher der neuen Spieleroberfläche hingegen verwendet dieses Wörterbuch nicht.

Effekt Oberbegriff für Regeln, Aktivitäten und Benutzerkommandos. Effekte erzeugen die Dynamik eines SESAM-Modells. Sie werden im Effektemodell zusammengefasst.

Effektemodell Bestandteil eines SESAM-Modells. Enthält alle Effekte, also Regeln, Aktivitäten und Benutzerkommandos.

Entität Instanz eines Entitätstyps. Besitzt einen externen Namen.

Entitätstyp Typ für die Objekte der abstrakten Welt, die durch das Modell definiert ist und in der die Dynamik des Modells abläuft. Bestandteil des Schemamodells.

GPI **Graphical Player Interface**. Die Spieleroberfläche, die während dieser Diplomarbeit entstanden ist.

GUI **Graphical User Interface**. Eine graphische Benutzeroberfläche, bestehend aus Icons, Menüs, Pushbuttons und weiteren graphischen Elementen im Gegensatz zu Textkonsolen.

Hochsprache Modellbeschreibungssprache eines SESAM-Modells. Besitzt im Vergleich zur Basissprache viele, reichhaltige Konzepte. Kann durch einen Hochsprachenübersetzer in Basissprache übersetzt werden. Die Modellbeschreibung in Hochsprache wird von den Modellierungswerkzeugen erzeugt. Zum Zeitpunkt dieser Diplomarbeit ist die einzige Hochsprache SEMOS-2. Daher ist mit dem allgemeinen Begriff Hochsprache diese spezielle Ausprägung gemeint.

Hochsprachenübersetzer Werkzeug, das eine Modellbeschreibung in Hochsprache in eine Modellbeschreibung in Basissprache übersetzt.

Java Eine objektorientierte Programmiersprache.

Kommando Ergebnis der Übersetzung einer Spielereingabe durch den Dolmetscher, das an die Basismaschine weitergeleitet wird und dort die Ausführung eines Benutzerkommandos auslösen kann. Ein Kommando hat einen Namen und eine (möglicherweise leere) Menge von Parametern. In dieser Diplomarbeit wird zum Teil auch die Spielereingabe selbst bereits als Kommando bezeichnet, sofern sich der Sinn aus dem Kontext erschließen lässt.

Komponente Oberbegriff für Entitäten und Relationen.

Icon Ein Widget, das ein kleines Bild darstellt und meistens auf Pushbuttons angebracht wird.

Modell Abbildung eines Ausschnitts der Realität zur Veranschaulichung komplexerer Vorgänge und Zustände. Häufig im Sinne von SESAM-Modell gebraucht, wobei das Original dann ein Software-Projekt ist.

Modellbeschreibungssprache Sprache zur Beschreibung eines SESAM-Modells. In SESAM-2 werden zwei Ebenen unterschieden: Basissprache und Hochsprache.

Modellbauer Siehe Modellierer.

Modellierer Entwerfer und Realisierer eines SESAM-Modells. Der Modellierer verwendet die Modellierungswerkzeuge zum Zwecke der Modellerstellung oder -modifikation.

Modellierungswerkzeuge Werkzeuge, die zur interaktiven Erstellung der Modellbeschreibung in Hochsprache dienen.

Nachricht 1. Mitteilung des SESAM-Modells an den Spieler. In SESAM-2 liegt die Nachricht in einer symbolischen Form (Nachrichtenbezeichner und Parameter) vor, die durch den Dolmetscher in eine dem Spieler verständlichen Form gebracht und ausgegeben wird. Das Versenden einer Nachricht ist eine Aktion.

2. Kommunikation zwischen zwei Objekten in der objektorientierten Programmierung. Auch Botschaft genannt.

Oberfläche Im Sinne von Benutzeroberfläche oder Benutzerschnittstelle verwendet. Die (oft graphische) Eingabemaske für einen Anwender.

Original Ausschnitt der realen Welt oder eines Modells, der zum Gegenstand einer Modellierung gemacht wird. In SESAM-Modellen in der Regel ein Software-Projekt.

Parameter 1. Bestandteil eines Kommandos.
2. Bestandteil einer Methode in der Programmierung.

Protokoll Von der Basismaschine geführte Aufzeichnung über die eingegebenen Kommandos des Spielers und die Spielstände während der Modellsimulation. Das Protokoll dient als Eingabe für die Auswertungswerkzeuge und damit zur Analyse eines Spiels.

Pushbutton Ein Widget, das einen Druckknopf darstellt.

QA-Modell Siehe QS-Modell.

QS-Modell Qualitätssicherungsmodell. Ein SESAM-Modell eines Entwicklungsprozesses, dessen Fokus auf qualitätssichernden Maßnahmen liegt.

Regel Regeln sind Effekte, die der Änderung des Modellzustands dienen.

Regelmodell Veraltet, siehe Effektemodell.

Relation Instanz eines Relationstyps.

Relationstyp Typ für die Beziehungen zwischen den Objekten der abstrakten Welt des Modells, also für Beziehungen zwischen Entitäten. Bestandteil des Schemamodells.

- Schemamodell** Bestandteil eines SESAM-Modells. Beschreibt ähnlich einem Entity-Relationship-Modell die abstrakte Welt, in der das Modell abläuft. Besteht aus Attributtypen, Entitätstypen und Relationstypen.
- Scrollbar** Ein Widget, mit dem man einen Bildausschnitt so verschieben kann, dass man alle Teile des Bilds sehen kann.
- SEMOS-2** Eine Hochsprache. Abkürzung für **SESAM-Modellbeschreibungs-Sprache** für SESAM-2.
- SESAM** **Software Engineering Simulation by Animated Models**. Eine interaktive Computersimulation zur Ausbildung von Projektleitern. In dieser Diplomarbeit ist mit SESAM in aller Regel SESAM-2 gemeint.
- SESAM-1** Erste Implementierung des SESAM-Systems. Dient als Pilotsystem für weitere Implementierungen.
- SESAM-2** SESAM-System, das als Nachfolger des SESAM-1-Systems konzipiert wurde.
- SESAM-Modell** Ein Simulationsmodell, das durch das SESAM-System animiert werden kann. Ein SESAM-Modell besteht aus einem Schemamodell und einem dazu passenden Effektmodell und Situationsmodell.
- SESAM-System** Konkrete Implementierung der Konzepte von SESAM. Umfasst Aspekte der Modellerstellung und Modellanimation sowie der Spielauswertung. Besteht aus einer Menge von Werkzeugen.
- Simulationsschritt** Der Vorgang der Herstellung einer neuen Situation aus der aktuellen Situation durch Anwendung der Effekte des Effektmodells. Bei einem Simulationsschritt wird die Simulationszeit um die Simulationsschrittweite fortgeschaltet.
- Simulationsschrittweite** Die Zeitspanne, die zur Simulationszeit hinzugezählt wird, wenn ein Simulationsschritt durchgeführt wird. Die Simulationsschrittweite definiert damit die Zeiteinheit der Simulation.
- Simulationszeit** Zeitmarkierung des Simulationsmodells.
- Simulator** Einheit aus Basismaschine und Dolmetscher. Für die neue Spieleroberfläche ist in dieser Diplomarbeit mit Simulator auch die Einheit von Basismaschine und Spieleroberfläche gemeint.
- Situation** Kurz für Situationsmodell.
- Situationsmodell** Bestandteil eines SESAM-Modells. Repräsentiert den eigentlichen Modellzustand (Szenario), auf den das Effektmodell angewandt wird. Es enthält Instanzen der im Schemamodell eingeführten Entitätstypen und Relationstypen. Damit kann es als Instanz des Schemamodells aufgefasst werden. Zusätzlich umfasst das Situationsmodell eine Simulationszeit.
- SLOC** Source Lines of Code. Ein Maß für den Umfang von Quellcode. Dabei werden sämtliche Zeilen des Programms ermittelt, auch Leer- und Kommentarzeilen.
- Spiel** Vorgang der Benutzung der Ausführungswerkzeuge durch einen Spieler zum Zwecke der (interaktiven) Ausführung eines SESAM-Modells.

Spieler Benutzer des SESAM-Systems, der den Projektleiter des simulierten Projekts mimit. Er kommuniziert über das Ausführungswerkzeug mit einem SESAM-Modell. Dabei kann er Kommandos an das SESAM-Modell schicken und von ihm Nachrichten empfangen.

Spieleroberfläche Die Schnittstelle des Spielers zur Basismaschine. Für das bisher bestehende System war dies die DemoGUI, die in dieser Diplomarbeit durch die GPI ersetzt werden soll.

Spielstand Zustand einer Ausführung eines SESAM-Modells. Besteht aus einem Situationsmodell und der Menge der momentanen Instanzen kontinuierlicher Effekte.

Startsituation Ein Situationsmodell, das den Anfangszustand eines SESAM-Modells beschreibt.

Tutor Betreuer (Spielleiter) bei Spielen mit dem SESAM-System. Aufgabe des Tutors ist die Einführung des Spielers in das konkrete SESAM-Modell, die Vorbereitung geeigneter Startsituationen, die Betreuung des Spielers während des Spiels und die Auswertung und Besprechung der Spielergebnisse mit dem Spieler.

UML Unified Modeling Language. Eine formale Notation, die im Wesentlichen dazu geeignet ist, die Spezifikation und den Entwurf eines Softwareprodukts zu beschreiben.

Werkzeug Werkzeuge in SESAM-2 sind die Modellierungswerkzeuge, die Basismaschine, der Hochsprachenübersetzer, der Dolmetscher und die Auswertungswerkzeuge.

Widget Ein Bildelement einer graphischen Benutzeroberfläche, z.B. ein Eingabefeld, ein Button oder ein Icon.

Wörterbuch Das Wörterbuch wird vom Dolmetscher verwendet, um Spielereingaben in Kommandos für die Basismaschine zu übersetzen und Nachrichten in normale, für Menschen verständliche Ausgaben zu wandeln.

Anhang C

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken nach bestem Wissen und Gewissen als solche kenntlich gemacht habe.

Stuttgart, den

.....

