# Interaktive Realoptionen-Simulation

–

Ein Framework für interaktive Finanz-Simulationen
und
Mobile-Computing-Experimente

Diplomarbeit im Fach Wirtschaftsinformatik
vorgelegt von

David Frank
Zürich, Schweiz
Matrikelnummer 90-912-99

Bubentalstrasse 7
CH-8304 Wallisellen
david.frank@tiscali.ch

Angefertigt am
Institut für Informatik
der Universität Zürich
Prof. Abraham Bernstein, Ph.D.

in Zusammenarbeit mit Prof. Marc Chesney
vom Swiss Banking Institute, Zürich

Betreuer: Abraham Bernstein und Felix Morger
Abgabe der Arbeit: 24. August 2004

## Inhaltsverzeichnis

**Abbildungsverzeichnis**



*Sämtliche Abbildungen wurden selber erstellt.*

**Tabellenverzeichnis**

**Dank**

Ein herzliches Dankeschön gilt Prof. Abraham Bernstein und Prof. Marc Chesney für die interessante Aufgabenstellung.
Felix Morger hat mit seiner ausgezeichneten Spezifikation und der Hilfe beim Testen wesentlich zum Gelingen dieses Werkes beigetragen.
Ein spezieller Dank gebührt meiner Familie, ohne deren moralische Unterstützung und geduldige Nachsicht diese Arbeit nicht zustande gekommen wäre.

**Zusammenfassung**

Für das Einüben von Realoptionen-Berechnungen (einer Methode zur Entscheidungs-Bewertung) im Unterricht wurde für das Swiss Banking Institute der Universität Zürich ein Simulations-Spiel programmiert. Als Spiel-Szenario dienen Ölförder-Firmen, die in Ölfelder und Technologien investieren und über Fördermengen entscheiden können. Dem Spiel wurde eine verteilte, meldungsbasierte Client-Server-Architektur zugrunde gelegt, die sich auch für Experimente mit mobilen Geräten eignet. Der Server lässt sich von Fern administrieren, ist modular aufgebaut, unterstützt die Persistenz von Spiel-Informationen und stellt Hilfsfunktionen zur Verfügung, mit denen viele Typen von Spielen einfach implementiert werden können. Als Client-Benutzerschnittstelle kann wahlweise eine vielseitige Java-Applikation oder ein XHTML-kompatibler Browser benutzt werden.

**Abstract**

A simulation game has been built allowing students at the Swiss Banking Institute (University of Zurich) to practice investment valuation based on the Real Options method. The play scenario lets students take control of an oil company. They can invest in oil fields and drilling technologies and decide when and how much oil to produce.
The game is built on top of a modular, message-driven client/server architecture, making it very easy to add completely different types of games and it supports remote administration and data persistence. It handles Java/Swing and XHTML clients, making it also suitable as a platform for experimenting with mobile devices.

# 1. EINLEITUNG

## 1.1. AUFGABENSTELLUNG

Am Swiss Banking Institute der Universität Zürich wird für den Unterricht ein Wirtschafts-Simulationsspiel benötigt, mit dem Unterrichtsinhalte spielerisch aber doch realistisch trainiert werden können. Nachfolgend wird – allerdings nur soweit es für diese Arbeit relevant ist – der Kontext beschrieben, in dem das Spiel eingesetzt werden soll. Weiter werden diejenigen Vorgaben aufgeführt, die Teil der Aufgabenstellung sind.

### 1.1.1. Kontext: Realoptionen

Die Bewertung von Entscheidungen ist für die Finanzwissenschaften eine interessante Fragestellung. Insbesondere Entscheide mit grossen finanziellen Auswirkungen (grosse Investitionen und strategische Transaktionen) wecken das Bedürfnis nach einer systematischen Erfassung von Chancen und Risiken und nach einer Quantifizierung des Wertes und Nutzens von Investitionen und Entscheiden.
Die Realoptionen-Methode, welche am Swiss Banking Institute von Prof. Marc Chesney gelehrt wird, sieht den Wert von Investitionen in ihrer Eigenschaft, zu einem späteren Zeitpunkt (gewinnbringend) verwertbar zu sein. Sie bewertet Investitionen und Entscheide deshalb ähnlich wie das Finanzinstrument der 'Option', die ja ebenfalls ein Recht darstellt, das zu einem künftigen Zeitpunkt ausgeübt werden kann (z.B. das Recht, Wertpapiere zu einem bestimmten Kurs zu kaufen oder verkaufen). Wie eine Option kann auch eine Investition in der Zukunft genutzt werden oder ungenutzt verfallen. Und beide, Option wie Investition, verlangen eine Vorleistung (Kaufpreis) und beinhalten ein Risiko (z.B. im Fall der Option den unbekannten zukünftigen Kurs des Wertpapiers).
Ein knapper Überblick über die Realoptionen-Theorie und interessante Betrachtungen zu ihrer Anwendbarkeit auf IT-Projekte, sowie weiterführende Literatur findet sich in der Seminararbeit von Eibensteiner [1].

### 1.1.2. Spiel-Szenario: Ölförderung

Zur Einübung der mathematischen Bewertungsformeln der Realoptionen-Methode soll ein Spiel bereitgestellt werden, das die gesuchten Elemente (Entscheide fällen, deren Wert sich erst in der Zukunft manifestiert) in einem einigermassen plausiblen Szenario anbietet.

Felix Morger, Assistent am Swiss Banking Institute, hat mit Prof. Marc Chesney ein Spiel entwickelt, das diese Anforderung erfüllt und vielseitig genug ist, um später erweitert werden zu können.

Es versetzt den Studenten in die Rolle einer Ölfirma, welche bei einer Auktion Bohrrechte erstehen kann und darauf in Bohrtechnologie investiert, um Öl fördern zu können, das dann verkauft werden kann. Sowohl das Auktionsgebot als auch die Wahl der Bohrtechnologie, der Zeitpunkt der Investition in Bohrtechnologie und die Wahl der Fördermenge sind dabei Entscheide, deren Wert erst beim Verkauf des Öls manifest werden, denn erst zu diesem späten Zeitpunkt wird ein Ölpreis festgelegt. Das Szenario kann beispielsweise durch die Wahl unter mehreren Bohrtechnologien mit verschiedenen Kosten und Förderkapazitäten oder durch die Möglichkeit, Öl vor dem Verkauf zu lagern, beliebig ausgebaut werden.

Im Anhang A ist die Spezifikation von F. Morger zum Spielablauf abgedruckt, die sich als sehr durchdacht und nützlich erwiesen hat.

### 1.1.3. Zusätzliche Vorgaben

Das Spiel soll so aufgebaut sein, dass künftige Erweiterungen möglich sind. Gewisse Spielparameter sollten ohne Neuprogrammierung veränderbar sein und das Spiel soll von Nicht-Informatikern administriert werden können.

Das Realoptionen-Spiel (oder Varianten davon auf derselben technischen Basis) soll auch bei Experimenten mit mobilen Kleingeräten eingesetzt werden können.

## 1.2. VORGEHEN

In *Kapitel 2* werden zuerst die genannten Anforderungen präzisiert und systematischer festgehalten. Daraus werden dann erste Ansprüche an die Architektur abgeleitet. *Kapitel 3* untersucht die Vor- und Nachteile einiger Technologien, die zu einer Lösung beitragen könnten. Im *Kapitel 4* wird dann die gewählte Architektur vorgestellt und die zugrunde liegenden Konzepte werden erläutert. Das *Kapitel 5* wendet sich Implementationsaspekten der Spielplattform zu und beschreibt die

verwendeten Hilfsmittel. *Kapitel 6* zeigt am Beispiel des Administrationstools, wie sich mit der Spielplattform mit einfachen Mitteln verteilte Anwendungen realisieren lassen und im *Kapitel 7* wird schliesslich mit den gleichen Mitteln das Realoptionen-Spiel gebaut. Das *Kapitel 8* zieht zusammenfassend Bilanz über die geleistete Arbeit und zeigt auf, wohin sich das Spiel entwickeln könnte.

# 2. SPEZIFIKATION

In diesem Kapitel werden die in der Einleitung vorgestellten Anforderungen ergänzt und kommentiert. Durch Vorgaben präjudizierte Design-Entscheide werden angesprochen und es wird aufgezeigt, in welchen Bereichen Spielraum für Architekturentscheide besteht.

## 2.1. ANFORDERUNGEN REALOPTIONEN-SPIEL

Für das Realoptionen-Spiel hat Felix Morger eine detaillierte Spezifikation ausgearbeitet (vgl. Anhang A), die sich sachlich auf den Spielablauf konzentriert und keine direkten Vorgaben zur Implementation macht.
Die primären Anforderungen an das Spiel sind:

- Mehrere Spieler spielen zusammen ein Spiel (multi-player).
- Jeder Spieler soll von einem eigenen Rechner aus spielen können (distributed).
- Spielbar auf gängigen Rechnern; möglichst niedrige Anforderungen an die Server-Hardware und -Software.
- Das Spiel ist rundenbasiert, d.h. die Spielschritte wiederholen sich zyklisch.
- Das Spiel muss flexibel mit unterschiedlich vielen Spielern spielbar sein.
- Diverse Parameter (z.B. zur Ölpreis-Berechnung) sollten vor dem Spielstart einfach angepasst werden können.
- Es sollte von Nicht-Informatikern bedienbar sein.
- Zu einem gewissen Grad sollte auch die Spielstruktur ohne Umprogrammierung anpassbar und modular erweiterbar sein.
- Spielresultate sollten abgespeichert/exportiert werden können.

Aus diesen Punkten ergeben sich erste Schlussfolgerungen:

- Es braucht einen zentralen Server, auf dem das Spiel läuft und Clients, mit denen die Spieler Informationen zum Spiel betrachten und Spiel-Entscheide an den Server übermitteln können.
- Das Spiel muss von einem Administrator (Kursleiter) parametrisiert gestartet werden können. Das sollte ohne physischen Zugang zum Server möglich sein.
- Um eine gewisse Flexibilität zu erreichen, muss es eine konzeptionelle Unterteilung in Spiel-Infrastruktur und Realoptionen-Spiel geben. Auf derselben Infrastruktur sollten (wenn möglich sogar gleichzeitig) verschiedene Varianten des Realoptionen-Spiels laufen können. Neue Spiele oder neue Varianten sollten möglichst einfach zu erstellen sein.
- Es braucht einen Authentifizierungs-Mechanismus, da die Spieler nicht anonym sein sollen. Allerdings darf dadurch kein administrativer Aufwand (z.B. zur Eröffnung von Benutzer-Konti) entstehen.
- Das Design des Realoptionen-Spiels erfordert eigentlich keine hochinteraktive, aufwändige Benutzerschnittstelle. Trotzdem gehen die Interaktionsmöglichkeiten des Spiel-Clients über diejenige einer typischen Web-Seite (HTML) hinaus, indem nicht nur nach dem Hol-Prinzip (pull) Informationen vom Server zum Client fliessen sollten, sondern die Kommunikation jederzeit auch auf Initiative des Servers (push) möglich sein muss. Im Prinzip könnte das aber auch durch Polling simuliert werden (d.h. der Client erkundigt sich periodisch beim Server nach den neusten Informationen).

Aus dem Universitäts-Umfeld, in dem die Spiele durchgeführt werden, ergeben sich zwei weitere Einschränkungen:

- Erstens ist die Netzwerk-Infrastruktur am Einsatzort unbekannt. Es kann zwar davon ausgegangen werde, dass ein TCP/IP-Netzwerk zur Verfügung steht und dass Bandbreite kein Problem darstellt. Ungewiss ist jedoch,
    - ob Firewalls zwischen Client und Server liegen und wie die ggf. konfiguriert sind.

- ob Client und Server sich im selben Subnetz befinden.
- ob dem Spiel-Server ein Domain-Name (via DNS) zugewiesen wird.

Daraus folgt, dass die verwendete Kommunikationstechnologie es zulassen sollte, im Extremfall über einen einzigen Netzwerk-Port konfiguriert zu werden, denn auch in einem sehr restriktiven Umfeld ist z.B. Port 80 [HTTP] meist durchlässig. Weiter entfällt wohl die Möglichkeit 'Broadcasting' einzusetzen (sei es, um den Server zu finden oder zur Verteilung von Informationen). Und schliesslich muss der Client den Server sowohl über einen Namen (Domain-Name) als auch über eine IP-Nummer anwählen können.

- Zweitens ist die Art der Client-Rechner (Betriebssystem/Ausstattung) unbekannt.

Darum kommt für den Client (und indirekt auch für den Server) nur eine Technologie in Frage, die einigermassen plattformneutral ist. Durch die Aufgabenstellung ist dafür bereits Java vorgegeben. Entschieden wurde darüber hinaus mit Professor Bernstein, dass keine Rücksicht (in der Form von Rückwärtskompatibilität) auf alte Java-Versionen genommen werden soll. Damit wird praktisch aber nur das veraltete Mac OS 9 als Client-Betriebssystem ausgeschlossen, weil dafür nur Java 1.1 erhältlich ist. Für alle gängigen Rechner mit Windows, Mac OS X oder Linux ist hingegen das verwendete Java 1.4 kostenfrei erhältlich.

### 2.2. ANFORDERUNGEN MOBILE COMPUTING

Neben dem Realoptionen-Spiel, das im 'Klassenzimmer' zum Einsatz kommt, sollen verwandte Spiele auch auf derselben Plattform über mobile Geräte gespielt werden können. Dies hat Auswirkungen auf die gesamte Architektur von Client und Server:

- Der Einsatz unterschiedlicher Präsentationstechnologien (möglicherweise sogar gleichzeitig im selben Spiel!) wirft die Frage auf, auf welcher Abstraktionsebene der Server Informationen und Interaktionsschnittstellen anbieten muss, die dann von unterschiedlichen Clients verschieden präsentiert werden können.

- Spieler müssen jederzeit aus dem Spiel aussteigen können und werden vielleicht erst Tage später wieder ins Spiel einwählen und weiterspielen. Auch abwesende Spieler müssen aber (spätestens wenn sie wieder einwählen) über den Spielverlauf orientiert werden können.
- Die Kopplung zwischen Client und Server muss sehr lose sein können. Als Extrembeispiel könnte etwa das Versenden einer SMS mit dem Text 'BID 1650000' genügen, um bei einer Ölfeld-Auktion mitzubieten. Das legt nahe, dem Spiel zumindest für die Aktionen der Spieler eine meldungs-artige Kommunikationsform zugrunde zu legen.
- Auf mobilen Kleingeräten stehen allenfalls Java-Umgebungen mit stark eingeschränktem Funktionsumfang zur Verfügung. Häufiger anzutreffen – und mit weniger Entwicklungsaufwand verbunden – ist hingegen die Unterstützung für einfache XHTML-Schnittstellen.

### 2.3. ZUSÄTZLICHE ZIELE

Im Rahmen der Vorgespräche ist der Wunsch geäussert worden, im Hinblick auf eine mögliche Weiterentwicklung (z.B. durch andere Informatiker am Institut für Informatik) seien möglichst nur 'gängige' Technologien einzusetzen und keine Bibliotheken oder Frameworks zu benutzen, die einen hohen Einarbeitungsaufwand erfordern würden.

Als 'erlaubte' Technologien wurden dabei JSP (Java Server Pages) und allenfalls 'Struts' zur Generierung von HTML-Seiten erwähnt, JDBC als Datenbankschnittstelle und (in der Aufgabenstellung) RMI, WebServices und JavaSpaces als mögliche Kommunikations-Technologien.
Kapitel 3 wird sich im Detail mit der Auswahl von Kommunikations- und Darstellungs-Technologien beschäftigen.

### 2.4. ZUSAMMENFASSUNG DER ANFORDERUNGEN

Es folgt eine stichwortartige Zusammenfassung der Anforderungen an die Spielumgebung:

| **Anforderung** | **Impliziert…** |
|---|---|
| Mehrere Spieler | Client/Server, Authentifizierung |
| Mehrere Client-Typen, verteilt | Abstrahiertes User-Interface |
| Mehrere Spiel-Typen | Abgrenzung Spiel-Infrastruktur / Realoptionen-Spiel |
| Administrierbar von Nicht-Informatiker | Administrator mit anderen Rechten als die übrigen Spieler |
| Konfigurierbar ohne Neuprogrammierung | System zur Gliederung und Konfiguration eines rundenbasierten Spiels |
| Unbekanntes Netzwerk-Umfeld | Flexible Port-Wahl, wenig Infrastruktur-Anforderungen |
| Unbekannte Client-Computer | Client in Java programmiert |
| Mobile-Clients, lose Kopplung | Meldungs-basierte Kommunikation |
| Weiterentwicklung durch Dritte | Verbot exotischer Technologien |

**Tabelle 1: Anforderungen an die Spielumgebung**

## 3. EVALUATION

### 3.1. VERTEILUNG UND BENUTZERSCHNITTSTELLE

Obwohl Java als 'die' Sprache bzw. Plattform schlechthin für verteilte, mobile Anwendungen gilt, hat es sich doch als unerwartet schwierig erwiesen, in der Vielfalt vorhandener Verteilungs- und Schnittstellen-Technologien eine zu finden, die unsere Bedürfnisse abdeckt.

Wünschbar wäre ja einerseits, die Spiel-Logik (auf einem Server) von der Benutzerschnittstelle (auf entfernten Clients) zu trennen. Andererseits wird eine Technologie gesucht, durch welche die Präsentationsschicht abstrahiert werden kann und die zumindest Java/Swing und HTML unterstützt.

Viele der verfügbaren Technologien lösen jedoch nur Teilaspekte unseres Problems. So bietet beispielsweise **SwingML** [2] die interessante Möglichkeit, ein Java-Swing-Interface mittels XML zu beschreiben, womit sich im Prinzip aus einer gemeinsamen XML-Repräsentation sowohl eine Swing- als auch eine HTML-Schnittstelle generieren liesse. In eine ähnliche Richtung gehen auch **SwiXml** [3] und **Thinlet** [4]. Letzteres ist speziell auf mobile Java-fähige Geräte zugeschnitten.
Noch vielversprechender ist **WebOnSwing** [5], das in der umgekehrten Richtung aus Swing-Schnittstellen HTML-Seiten generiert.
Sie alle scheinen jedoch wenig ausgereifte und mit relativ hohem Lernaufwand verbundene 'Exoten' zu sein.

Unter den Java-Basistechnologien kommt man sicher nicht daran vorbei, RMI und J2EE genauer zu betrachten.

### 3.1.1. RMI

RMI (Remote Method Invocation) könnte auf zwei Arten von Nutzen sein. Einerseits könnte der Spiel-Client eine relativ schmale RMI-Schnittstelle zum Server für die Kommunikation benutzen[1]. RMI bringt so angewandt aber kaum Vorteile, erfordert einen RMI-Verzeichnisdienst (was kein Hindernis aber mitunter lästig ist) und spezielle Kompilierverfahren zur Stub-Generierung. Gegenüber Sockets ist ein etwas höherer Laufzeitaufwand zu beobachten.
Die zweite Anwendungsart bestünde darin, die gesamte Java-Benutzerschnittstelle serverseitig aufzubauen und dem Client per RMI zur Verfügung zustellen. Das ist insofern attraktiv, als es erlauben würde, den Spielern einen sehr einfachen Rumpf-Client zu verteilen, in den dann eine auf das jeweilige Spiel zugeschnittene 'reichhaltige' Benutzeroberfläche geladen werden könnte. Nachteilig ist dabei die relativ hohe Kopplung, die zwischen Client und Server entsteht. Ein grosser Teil der Präsentationsschicht würde in den Server verlagert. Das erhöht sowohl den Rechenaufwand für den Server als auch den Kommunikationsbedarf zum Client und ergibt dadurch vermutlich eine langsamere Benutzeroberfläche für den Spieler. Schwerer wiegt, dass die Komplexität auf der Serverseite stark zunimmt und ebenso

[1] Dies wurde im Realoptionen-Spiel als sekundäre, experimentelle Kommunikations-Möglichkeit neben Sockets auch tatsächlich implementiert.

durch die enge Kopplung das Risiko, dass ein Fehler auf dem Client den Server beeinträchtigt. Eine stabile Lösung liesse sich wohl nur erreichen, wenn serverseitig zwischen dem Spielprozess und der über RMI zu exportierenden Schnittstelle wiederum klar entkoppelt würde. In dem Fall ist aber nicht einzusehen, warum dieses 'Kopplungsprotokoll' nicht gleich als Kommunikationsprotokoll zum Client eingesetzt werden kann. Zudem liesse sich RMI so nur beim Java-Client anwenden, kaum aber beim Web-Client.

### 3.1.2. J2EE und Client-Frameworks

Als 'State-of-the-Art' für Serverprogramme und Business-Logik gilt bei Java gemeinhin J2EE (Java 2 Enterprise Edition) und EJB (Enterprise Java Beans) – ein Sammelsurium von Technologien, die Modularisierung in Komponenten, Verteilung über "n Schichten", automatische Persistenz, Sitzungs-Management, Messaging, Lookup-Services und vieles andere mehr abdecken.
So viel Funktionalität hat natürlich ihren Preis. J2EE besteht nicht einfach nur aus Bibliotheken und Frameworks, die man benutzen kann, sondern erfordert die Befolgung eines eigenen Programmiermodells, das durch eine inflationäre Benutzung von Stubs, Interfaces und Descriptor-XML-Dateien ein Projekt schnell sehr komplex macht. Daneben braucht es eine relativ komplexe Deployment-Infrastruktur.
Es ist möglich, dass ein J2EE-Experte durch geschickte Kombination einiger J2EE-Technologien relativ elegant eine mächtige Spiel-Plattform entwickeln könnte. Allerdings scheint sich der Aufwand für J2EE eher für grössere Projekte zu lohnen, die redundant verteilt über mehrere Rechner laufen müssen. Jedenfalls habe ich keinerlei plausible Gründe dafür gefunden, warum J2EE mir für meine Aufgabe von Nutzen sein könnte.

Immerhin schient es aber einige interessante Benutzerschnittstellen-Frameworks zu geben, die auf der Basis von J2EE-Serverprozessen funktionieren.
Das sind einerseits **ULC** [6] und das **Thin Client Framework** von IBM [7], die aber beide als kommerzielle Produkte ausscheiden.
Weiter gibt es mit **BS Framework** [8] ein kostenloses Produkt, das angeblich sowohl 'Rich Java/Swing Clients' als auch ein 'Struts Web-Interface' mitbringt. Beispiele für die Web-Schnittstelle gibt es allerdings keine und zum Zeitpunkt dieser Evaluation

war das Produkt zudem mitten in der Übergangsphase eines grösseren Versionswechsels.

### 3.1.3. Meldungsorientierte Middleware (MOM)

In Ermangelung eines passenden Produkts, das alle Problembereiche (Verteilung, Java- und Web-Client) sinnvoll abgedeckt hätte, wurde weiter Umschau gehalten nach reinen Verteilungstechnologien. Aus den Anforderungen lässt sich dabei ableiten, dass meldungsorientierte Technologien sich für unsere Zwecke besonders eignen würden.

Sowohl **Jini** (bzw. JavaSpaces) [9] als auch **JMS** (Java Message Service) [10] bieten als meldungsbasierte Technologien äusserst attraktive Eigenschaften (Entkopplung, garantierte Meldungs-Auslieferung, persistente Meldungen).

Eine weitere Variante wäre **JSDT** (Shared Data Toolkit for Java Technology), das sich noch besser für interaktive, zeitkritische Anwendungen eignet [11]. An der Universität Freiburg wurde es auch schon erfolgreich für Spiele eingesetzt [12].

Auf alle drei Technologien wurde letztlich aus demselben Grund verzichtet: Sie schaffen Abhängigkeiten und erfordern eine Infrastruktur, die konfiguriert und unterhalten werden muss, was für unseren Einsatzbereich ungünstig ist. Wie wir später sehen werden, würde es die gewählte Architektur aber erlauben, in der Kommunikationsschicht eine dieser drei Technologien einzusetzen.

### 3.1.4. Web-Technologien

Defacto-Standard bei Web-Technologien sind unter Java die so genannten Servlets. Das sind einfache Java-Klassen, die von `javax.servlet.http.HttpServlet` abgeleitet werden. Sie werden in einer speziellen Laufzeitumgebung (dem Servlet-Container) dynamisch geladen und werden ähnlich wie CGI-Programme unter Übergabe der HTTP-Anfrageparameter aufgerufen und generieren daraus dynamisch eine HTTP-Antwort, typischerweise in Form einer HTML-Seite.

Die grosse Schwäche der Servlets ist, dass durch Aneinanderreihen von einzelnen Zeichenketten mühsam ganze Seiten zusammengebaut werden müssen; d.h. Servlets bieten keinerlei spezifische Unterstützung für die HTML-Ausgabe. Umgekehrt bieten sie dadurch, dass sie praktisch auf HTTP-Protokoll-Ebene arbeiten, die sehr interessante Möglichkeit, dass sie jegliche Inhalte (seien es etwa Bilder oder PDF-Dokumente) dynamisch erzeugen und ausgeben können.

Java Server Pages (JSP) stellen das Prinzip der Servlets ("Java-Klassen die HTML-Text ausgeben") auf den Kopf und sind "HTML-Seiten mit eingebettetem Java-Code". Sie sind allerdings sehr eng mit Servlets verwandt, werden sie doch beim Laden in den Servlet-Container automatisch in Servlets umgewandelt.
Java Server Pages bieten komfortable Strukturierungsmöglichkeiten (sie können sich beispielsweise gegenseitig inkludieren) und durch so genannte Tag-Bibliotheken lassen sich wiederverwendbare Module erstellen. Unzählige Frameworks bauen auf JSP auf und bieten beispielsweise MVC-Architekturen an, die JSP zur Präsentation benutzen (z.B. Struts [13]).

Für unsere Zwecke eignen sich normale JSP wohl am besten, da mit ihrer Hilfe Seiten in beliebigen (X)HTML-Dialekten relativ einfach generiert werden können. HTML-, XHTML- oder WAP-Versionen sind dabei mit minimalen Änderungen realisierbar.
Nicht ganz befriedigt ist bei Web-Clients naturgemäss der Wunsch nach einer Push-Möglichkeit vom Server zum Client. Eine HTTP-Verbindung lässt sich zwar offen halten, aber die meisten HTML-Clients (Browser) und HTML-Generatoren (Application-Server) nutzen diese Möglichkeit zur simultanen Zwei-Weg-Kommunikation nicht. Eine gängige Lösung für dieses Problem ist Polling (z.B. durch ständiges Neu-Laden eines HTML-Frames mit der HTML-Direktive `<meta http-equiv="refresh" content="2">`), was jedoch bei mobilen Clients wegen der knappen Bandbreite und mangelnder Frame-Unterstützung problematisch ist.
Weitere Techniken dafür zeigen [14] und [15]. Natürlich kann das Push-Problem auch durch Benutzung anderer Medien gelöst werden (z.B. durch Versenden von SMS oder E-Mails, die auch Hyperlinks auf Serverinformationen enthalten können).

### 3.2. PERSISTENZ

Das anfängliche Bedürfnis nach einem Persistenz-Framework zur einfachen oder automatischen Abspeicherung von Spieldaten hat sich als eher überflüssig erweisen. Für unsere Zwecke (die selektive Abspeicherung ausgewählter Daten) erfordert ein ausgewachsenes Persistenz-Framework mehr Aufwand als es Nutzen bringt.
Ansonsten wären wohl das objekt-relationale Hibernate [16] oder Torque [17] interessante Hilfsmittel gewesen.
Für das Realoptionen-Spiel begnügen wir uns jedoch mit einfachem JDBC.

### 3.3. ALTERNATIVE SPIEL-FRAMEWORKS

Es wurde natürlich auch nach Vorbildern (bzw. Konkurrenten) auf dem Gebiet der Java-Spiele gesucht. In der Tat existieren einige Frameworks, die sich die Spiel-Unterstützung mit Java auf die Fahne schreiben. Mit Ausnahme des erstgenannten (GECCO) ist das Angebot an Spiel-Umgebungen aber erstaunlich dürftig:

- **GECCO** (Game Environment for Command and Control Operations) [18]
  Mit Abstand das interessanteste Vorbild ist GECCO, das als Meldungs-basiertes Strategie-Spiel einige Gemeinsamkeiten mit dem Realoptionen-Spiel aufweist. Als Spezialität benutzt es ein 'aktives' Spielfeld, das sich wie ein Zellulärer Automat verhält und so von Runde zu Runde eine Eigenaktivität entfaltet. Interessant ist, wie Gecco dynamisch Spielszenarien einliest und zur Laufzeit Spielklassen laden kann. Gecco benutzt für unterschiedliche Meldungstypen zwischen Client und Server je unterschiedliche Unterklassen eines Meldungs-Objekts (d.h. die Meldungen sind stärker typisiert als die Meldungen im Realoptions-Spiel).

- **Jags - The Java Game System** [19]
  Jags ist ein relativ altes (1999) System, das nicht über die Beta-Phase hinausgekommen ist. Es verwaltet mehrere Spiele und Spieler und stellt Kommunikationsdienste zur Verfügung. Das System ist kaum dokumentiert und wurde darum nicht weiter untersucht.

- **Ataraxia** [20]
  Antaraxia konzentriert sich primär auf Karten-Spiele und biete dafür ein generisches Java-Framework an mit Client/Server- und Peer-to-Peer - Kommunikation, Messaging. Äusserst interessant - aber leider nie über die Planungsphase hinausgekommen.

- **Gameframe** [21]
  Angepriesen als "Framework, das beim Bau von browserbasierten Mehrpersonenspielen mit Java und XSL helfen soll", ist Gameframe ebenfalls

in einer frühen Projektphase stecken geblieben.

- **Yale Game Server**
  Dieses System dient primär der Forschung auf dem Gebiet der künstlichen Intelligenz und lässt Agenten gegeneinander in rundenbasierten Brettspielen antreten. Die Stossrichtung ist zu unterschiedlich, als dass es uns etwas nützt.

Interessant ist übrigens, dass mit SAGSAGA [23] (Swiss Austrian German Simulation And Gaming Association - Gesellschaft für Planspiele in Deutschland, Österreich und Schweiz, e.V.) im deutschsprachigen Raum ein Verein existiert, der sich vorwiegend mit (computergestützen) Planspielen beschäftigt.

### 3.4. FAZIT

Die Evaluierungsphase war äusserst lehrreich. Auch wenn letztlich die Zahl der Technologien gering ist, die im Realoptionen-Spiel effektiv eingesetzt wurden.

Es wurde beschlossen, für den Server reines Java 1.4 zu verwenden, mit JDBC als Datenbankschnittstelle für die Persistenz und mit einer eigenen Netzwerkschicht welche Sockets und die Java-eigene Objekt-Serialisierung benutzt.
Für die Benutzeroberfläche soll JSP und Swing verwendet werden.

Die Architektur soll es aber erlauben, zusätzliche Technologien für die Benutzeroberfläche oder zur Kommunikation einzusetzen.

# 4. ARCHITEKTUR

In diesem Kapitel wird die Architektur der Spielumgebung vorgestellt, die so generell konzipiert ist, dass sie mit verschiedensten Arten von Spielen zurechtkommt. Bezeichnungen in Klammern (z.B. `Message`) schlagen dabei die Brücke zur Nomenklatur, die bei der Implementation verwendet wird.

## 4.1. ÜBERBLICK

Gewählt wurde eine meldungsbasierte Architektur mit Meldungen, die beliebige (serialisierbare) Attribute transportieren können. Die Architektur erlaubt die gleichzeitige Verwendung verschiedener Transportmechanismen (z.B. Sockets oder RMI). Es gibt einen zentralen Server, der im laufenden Betrieb beliebig viele Spiele unterschiedlicher Art laden, starten und beenden kann. Die Administration des Servers erfolgt vom Client aus. Spiele können Meldungen mit beliebigen Inhalten empfangen und nach einem selbstgewählten System darauf reagieren. Sie können aber auch auf eine vorgefertigte Infrastruktur zurückgreifen, um Informationen für die Clients strukturiert bereitzustellen und automatisch verteilen zu lassen.
Clientseitig stehen diese Informationen dann auf die gleiche strukturierte Art zur Verfügung. Die Darstellung der Informationen bleibt aber der Implementation des Clients überlassen.
Beim Server eingehende Meldungen können nach einem hierarchischen System einem zuständigen Spiel-Element zur Verarbeitung übergeben werden.

## 4.2. MELDUNGEN

Das Herzstück der Spielumgebung bildet die **Kommunikationsschicht**, die **Meldungen** (`Message`) zwischen beliebig vielen Clients und beliebig vielen Spielen auf dem Server transportieren kann. Die Kommunikation wird dabei in zwei Stufen vom Client her zum Server und weiter zu einem bestimmten **Spiel** (`Game`) aufgebaut. Der aufgebaute Kommunikationskanal kann dann auch vom Server benutzt werden, um Meldungen an den Client zurückzuschicken. Kommunikationskanäle können während einem ganzen Spiel offen bleiben. Clients werden im Spiel durch **Rollen** (`Role`), repräsentiert, die unabhängig davon existieren, ob der Spieler einen offenen Kommunikationskanal zum Spiel hat. Auch Spieler, die 'offline' sind, sind so

trotzdem im Spiel repräsentiert. Ein **Rollenverwalter** (`RoleManager`) kontrolliert die Zuordnung von Clients zu den Rollen und kann Zugangsbeschränkungen implementieren (z.B. passwortgeschützte Benutzerkonten).

Meldungen können mit beliebigen Inhalten zwischen Client und Spiel hin und her geschickt werden und die Clients und Spiele sind absolut frei, welche Bedeutung sie den Meldungsinhalten zumessen. Sie können selber ein Protokoll für den Inhalt einer Meldung wählen. Die im nächsten Kapitel vorgestellten 'Besitztümer' tun genau dies. Details zur Implementation der Kommunikationsschicht finden sich in Kapitel 5.3.

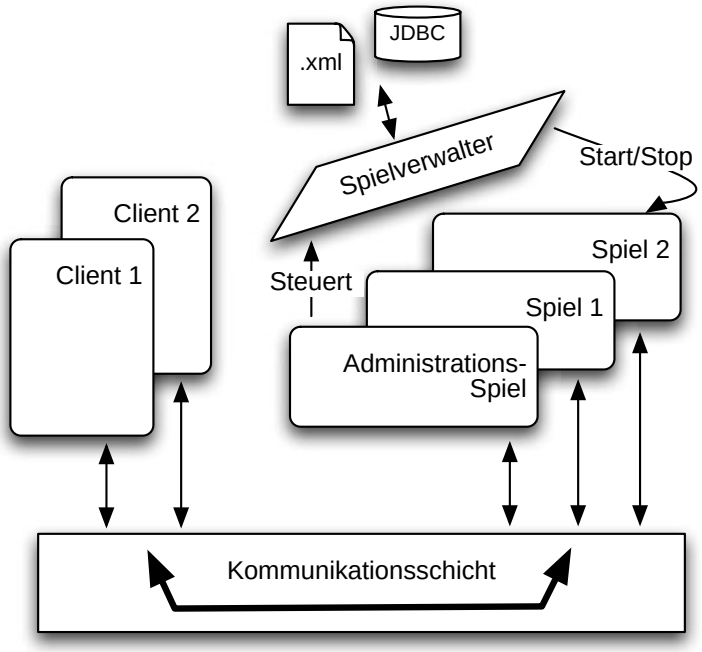



**Abbildung 1: Architektur-Überblick: Meldungen**

Die einzelnen Spiele werden von einem **Spielverwalter** (`GameManager`) kontrolliert. Er startet ein Spiel mit bestimmten Konfigurationsparametern, legt fest, über welche Arten von Kommunikationskanälen ein Spiel Meldungen empfangen darf und kann ein Spiel auch wieder beenden. Sämtliche Interaktionen des Administrators mit dem Spielverwalter laufen über das 'Administrations-Spiel', das in Kapitel 6 beschrieben ist, und benutzen damit genau dieselben Kommunikationsmethoden, die zwischen Clients und gewöhnlichen Spielen bestehen.

### 4.3. BESITZTÜMER

Als Ergänzung zu den 'unstrukturierten' Meldungen gibt es die Möglichkeit, auf einem höheren Abstraktionsniveau Informationen vom Spiel-Server zu den Clients zu verteilen.

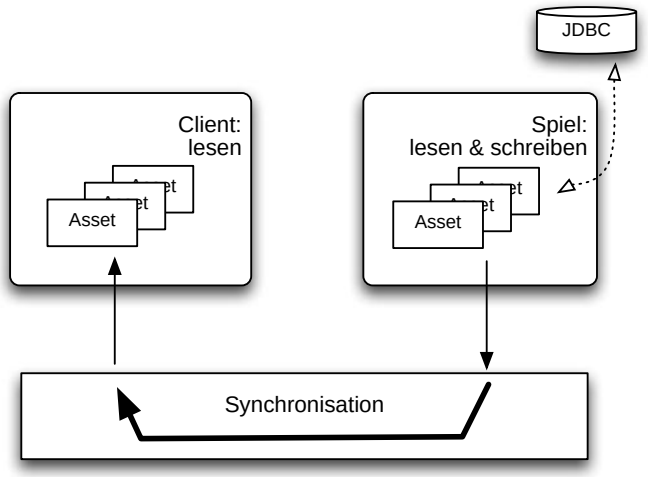



**Abbildung 2: Architektur-Überblick: Besitztümer**

Ein Spiel kann beliebig viele **Besitztümer** (`Asset`) anlegen, die beliebige Daten (serialisierbare Objekte) als Attribute enthalten können. Auf Besitztümer kann mit unterschiedlichen Mechanismen direkt zugegriffen werden. Besitztümer übernehmen deshalb für ein Spiel auch die Rolle von Globalen Variablen!
Wirklich nützlich werden sie aber erst dadurch, dass ihnen ein **Eigentümer** (`Role`) zugeordnet werden kann. Sobald dies geschieht, werden die Besitztümer automatisch zu demjenigen Client übermittelt, der mit der Eigentümer-Rolle verbunden ist. Es ist weiter möglich, Besitztümer gemeinsam allen Mitspielern und/oder Administratoren zuzuteilen. Wenn ein Spiel Änderungen an einem bestehenden Besitztum vornimmt (oder es gar löscht), werden diese Veränderungen dem Eigentümer automatisch weitergegeben. Ein Eigentümer findet jederzeit alle seine Besitztümer wieder vor, wenn er sich zu einem späteren Zeitpunkt erneut beim Spiel anmeldet. Besitztümer stehen dem Client nur lesend zur Verfügung – allfällige Änderungen, die der Client an seinen Besitztums-Kopien vornimmt, werden dem Spiel nicht gemeldet.
Besitztümer werden in Kapitel 5.5. ausführlicher beschrieben.

### 4.4. AKTIONEN

Während dem Client unkontrollierte Veränderungen seiner Besitztümer nicht erlaubt sind, können an jedes Besitztum beliebig viele **Aktionen** (`AssetAction`) angehängt werden. Sie repräsentieren vom Spiel sanktionierte Operationen auf ein Besitztum. Eine Aktion legt dabei das Format der Meldung fest, die beim Auslösen dieser Aktion an den Server übermittelt wird. Das Spiel kann so zu jedem Zeitpunkt durch Hinzufügen und Entfernen von Aktionen spezifizieren, welche Meldungen mit welchen Parametern es akzeptieren will. Zudem kann dem Spieler durch die Aktion auch erlaubt werden, einzelne Meldungsparameter formularartig selbst zu bestimmen.

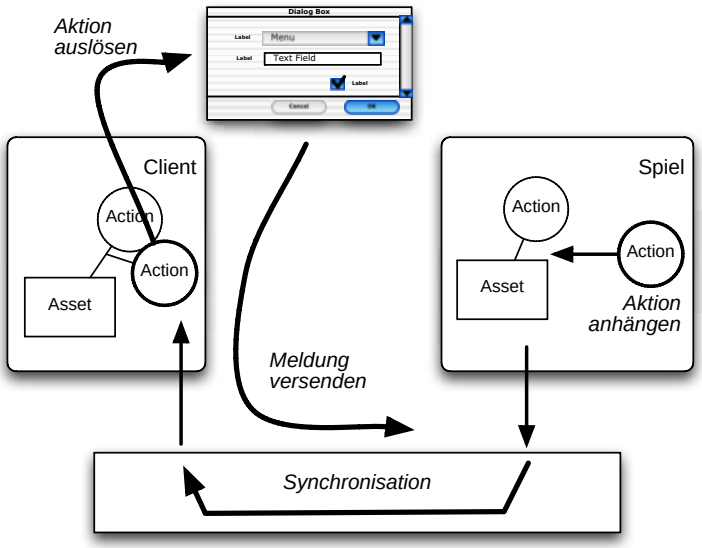



**Abbildung 3: Architektur-Überblick: Aktionen**

Jeder Client ist absolut frei, wie er die Besitztümer und Aktionen dem Benutzer präsentieren will. Es wird in der Implementation eine generische Benutzeroberfläche bereitgestellt, mit der alle Eigenschaften von Besitztümern betrachtet und Aktionen ausgelöst werden können. Ebenfalls bestehen Hilfsmittel, um einzelne Typen von Besitztümern herauszufiltern, um sie dann auf eine spezielle Art darzustellen.
Auf die Benutzerschnittstelle für Besitztümer und Aktionen wird im Kapitel 5.5.4 näher eingegangen.

### 4.5. SPIEL-SCHRITTE

Zur Strukturierung des Spiels (z.B. in Phasen oder nebenläufige Handlungen) wird serverseitig ein System angeboten, das durch hierarchische Schachtelung von **Spiel-Schritten** (`GameStep`) die Zuweisung eingehender Meldungen zu verschiedenen gleichzeitig aktiven Spiel-Elementen erlaubt. Damit lassen sich ähnliche Hierarchien aufbauen, wie sie etwa auch zur Event-Verteilung in Benutzeroberflächen verwendet werden (z.B. bei der hierarchischen Event-Verarbeitung im AWT im Gegensatz zum delegierenden Ansatz in JFC/Swing).

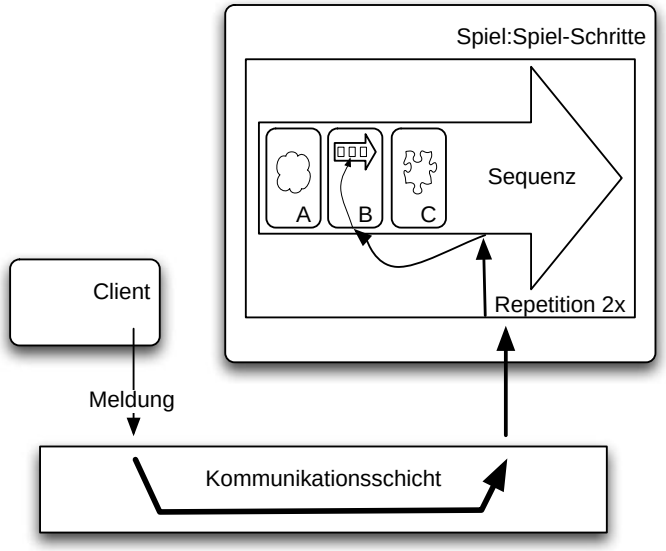



**Abbildung 4: Architektur-Überblick: Spiel-Schritte**

Abbildung 4 zeigt schematisch, wie eingehende Meldungen über mehrere Schritte einem bestimmten Spiel-Schritt zugeleitet werden können.
Einige standardisierte Spiel-Schritte (z.B. Iterations-Container für rundenbasierte Spiele) werden dabei von der Spielumgebung bereitgestellt. Ebenso wird eine Struktur der spiel-spezifischen Schritte vorgegeben, die dann mit wenig Aufwand durch Vererbung von vorgegebenen Klassen konstruiert werden können.
Spiel-Schritte werden in Kapitel 5.6 vertieft behandelt.

### 4.6. DISKUSSION

Zusammengefasst bilden Besitztümer, Aktionen und Spiel-Schritte ein System, das bis zu einem gewissen Grad dem Model-View-Controller -Konzept (MVC) entspricht. Besitztümer und Aktionen repräsentieren dabei als Modell den Spielzustand, der vom Client in einer selbst gewählten Art (View) präsentiert werden kann, worauf der Spieler mit dem Action-Formular (Controller) den serverseitig zuständigen Spiel-Schritt anweist, Änderungen am Modell vorzunehmen.

Die gewählte Architektur versucht, für Probleme, die vielen Spieltypen gemeinsam sind (nämlich, einen Spielzustand zu verteilen und Spielzüge zu verarbeiten), Lösungen anzubieten, ohne sich auf einen einzelnen Spieltyp einzuschränken.
Auf der Client-Seite wird eine benutzeroberflächen-neutrale Besitztümerverwaltung und die Möglichkeit der Kommunikation mit dem Server angeboten. Im Implementationsteil wird gezeigt, wie diese Dienste als Swing-Client oder JSP-Webseiten nutzbar gemacht werden können.

# 5. FRAMEWORK-IMPLEMENTATION

In diesem Kapitel wird im Detail die Infrastruktur beschreiben, die vom Realoptionen-Spiel benutzt wird. Der Quellcode liegt auf CD-ROM bei (vgl. Anhang C) und ist gründlich mit JavaDocs und Kommentaren dokumentiert.
Die bei der Umsetzung benutzen Hilfsmittel (Tools und Bibliotheken) werden in Kapitel 5.8. besprochen.

## 5.1. STRUKTURIERUNG

Das folgende Klassendiagramm zeigt, welche Beziehung die Infrastruktur-Klassen untereinander haben. Auf der linken Seite ist der Client mit seinen drei Netzwerkschichten und den replizierten Assets zu sehen, die mit dem `AssetBroker` and die Benutzeroberfläche vermittelt werden können.
Auf der rechten Seite, oberhalb der Netzwerkschicht, verfügt jedes einzelne Spiel – vom `GameManger` gestartet – über eigene Assets, eine Benutzerverwaltung (`RoleManger`) und über Spiel-Schritte, die den Spielverlauf ausmachen.

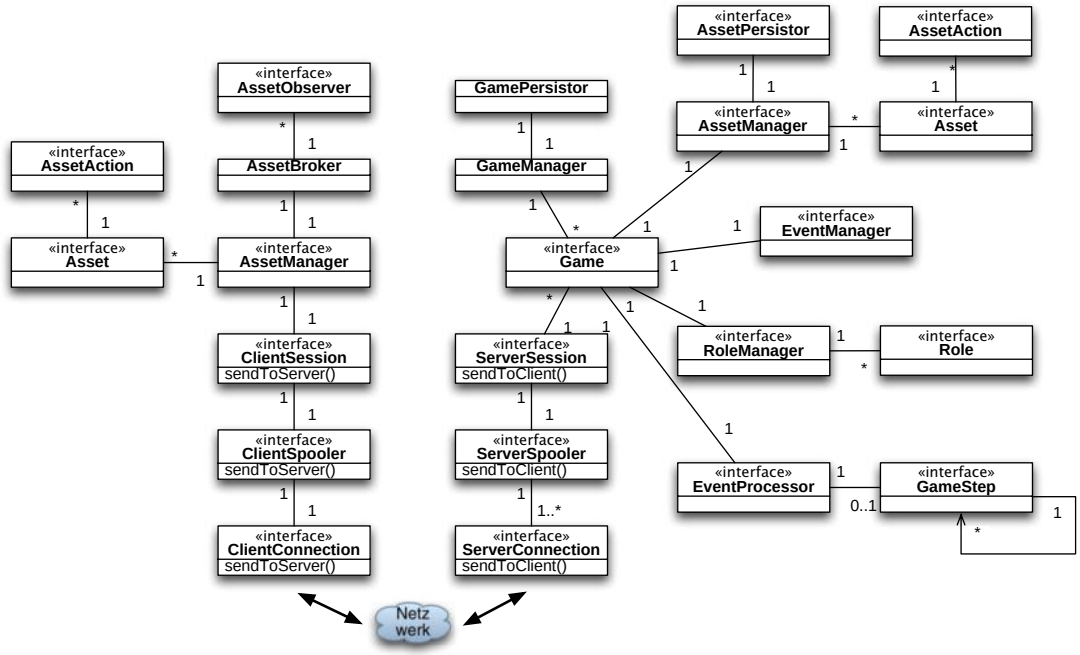



**Abbildung 5: Implementation: Beziehung der Klassen**

## 5.2. PROJEKT-STRUKTUR

Der Quellcode wurde wie in Java üblich hierarchisch in Pakete eingeteilt. Das gesamte Projekt befindet sich in einem Paket `dfrank.game` und ist in Client- und Server-Bereiche getrennt. Tabelle 2 zeigt die Paket-Struktur.

| **Paket-Hierarchie** | **Beschreibung** | Kapitel-Hinweis |
|---|---|---|
| `dfrank.game` | | |
| `client` | **Client**-Klassen | |
| `common` | | |
| `utils` | | |
| `connection` | Netzwerk: ClientConnection | 5.3.2. |
| `spooler` | Netzwerk: ClientSpooler | 5.3.3. |
| `session` | Netzwerk: ClientSession | 5.3.4. |
| `swing` | | 5.7.3. |
| `web` | Web-Client-Wrapper | 5.7.4. |
| `server` | **Server**-Klassen | |
| `admin_gamesteps` | Administrations-Spiel | 6. |
| `gamesteps` | Vorgefertigte Spiel-Schritte/Container | 5.6. |
| `impl` | Implementation von AssetManager, RoleManger, Game, etc. | 5.4.-5.5. |
| `utils` | | |
| `connection` | Netzwerk: ServerConnection | 5.3.2. |
| `spooler` | Netzwerk: ServerSpooler | 5.3.3. |
| `session` | Netzwerk: ServerSession | 5.3.4. |
| `common` | Von Client und Server **gemeinsam** benutzt | |
| `apachecommonscollection` | | |
| `rog` | **Realoptionen**-Spiel | |
| `server` | | |
| `rog_gs` | **Realoptionen**-Spiel-Schritte | 7. |
| `swingclient` | Java-Client | 5.7.3. |
| `models` | Datenmodelle zur Asset-Präsentation | 7.3. |
| `scrollabledesktop` | Fenster-Umgebung | 5.7.3. |
| `swingui` | | |
| `action` | Generierung der Action-Formulare | 7.3.2. |

**Tabelle 2: Code-Struktur**

## 5.3. KOMMUNIKATION

Die in Kapitel 4.2. vorgestellt Kommunikationsschicht, die Meldungen zwischen Client und Server transportiert, besteht server- und client-seitig aus je drei Schichten, bzw. drei Objekten, die eng zusammenarbeiten. Abbildung 6 illustriert dieses Zusammenspiel mit dem Klassen-Diagramm der serverseitigen Kommunikationsklassen und den verwendeten Schnittstellen.

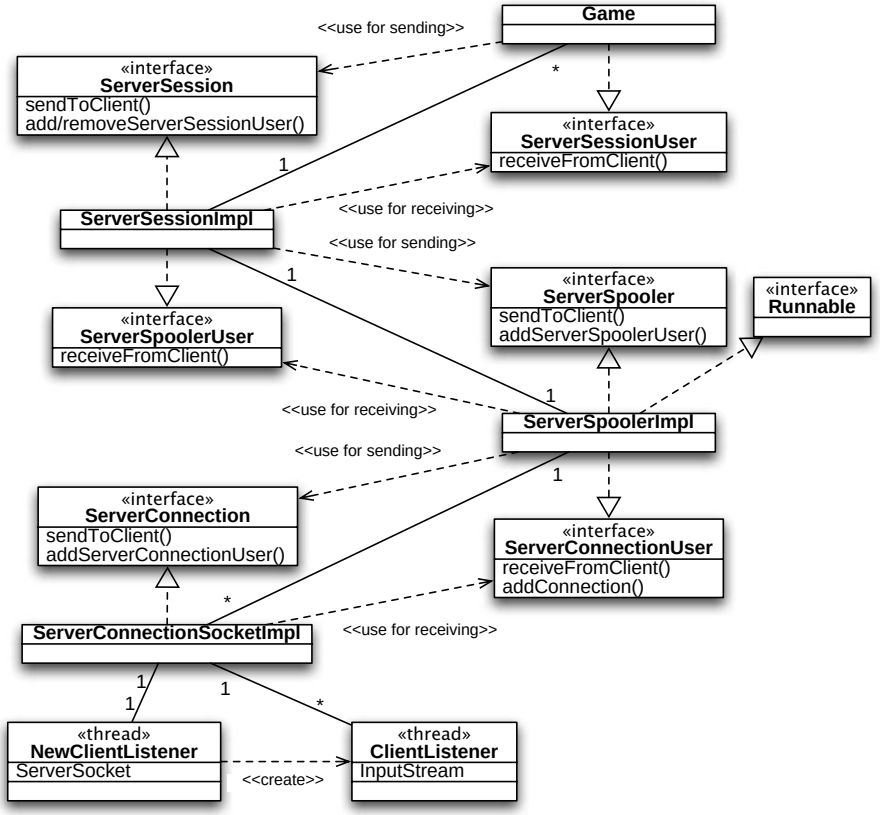



**Abbildung 6: Implementation: Server-Kommunikationsschicht**

Das *Connection*-Objekt in der untersten Schicht ist für Aufbau und Abbruch von Verbindungen zuständig und benutzt eine bestimmte Technologie (z.B. RMI oder Sockets), um serialisierbare Objekte über das Netzwerk zu transportieren. Auf der Client-Seite ist das Connection-Objekt für genau eine Verbindung zuständig – auf der Server-Seite hingegen für alle Verbindungen des selben Typs (z.B. für alle Socket-Verbindungen, die über Port 8080 hereinkommen).

Die mittlere Schicht, der *Spooler*, organisiert das Verschicken von Meldungen. Der *Server-Spooler* unterhält dazu eine Warteschlange ausgehender Meldungen und kann mehrere Connection-Objekte gleichzeitig benutzen (z.B. RMI plus Sockets über Port 8080 und 8081). Er verwaltet zudem die Verbindungsinformationen, die nötig sind, um die einzelnen Clients gezielt zu erreichen – also sowohl, welches Connection-Objekt für eine Verbindung zuständig ist, als auch allfällige weitere verbindungsspezifische Informationen. Jeder Client wird dabei über ein zugeteiltes Token identifiziert.

Der *Client-Spooler* bietet neben einer Warteschlange für ausgehende Meldungen auch die Möglichkeit, den Absender zu blockieren, bis eine Antwort auf die abgeschickte Meldung vom Server eingetroffen ist (d.h. RPC-artige synchrone Meldungen).
Die oberste Schicht, die *Session*, leitet auf der Server-Seite eingehende Meldungen an ein einzelnes Spiel weiter und kann dem Client Auskunft darüber geben, welche Spiele gerade laufen. Die Client-Session unterstützt entsprechend die Anfrage an den Server, welche Spiele verfügbar sind, und bietet eine Schnittstelle, um auf den Server einzuloggen.

### 5.3.1. Message

`dfrank.game.common.Message`

Messages sind Container, die beliebige serialisierbare Inhalte (Attribute) unter einem Namen (Key) speichern können. Dazu wird intern eine HashMap benutzt.
Attributnamen, die mit einer Unterstreichung ('_') beginnen, sind für das System reserviert. Messages, die bei einem Spiel eintreffen, werden dort auch als Events bezeichnet. Jede Message hat einen Meldungstyp, der primär dazu genutzt wird, herauszufinden, wer für ihre Behandlung zuständig ist. Jede Message bekommt zudem automatisch eine einmalige Identifikationsnummer zugeteilt.
Ein Client kann folgendermassen eine Meldung abschicken, sofern er eine funktionierende Session zum Server hat:

```
Message outMessage = Message.createMessage("myAddressMessageType");
outMessage.set("name", "david");
mySession.sendToServer(outMessage);
```

**Abbildung 7: Beispiel: Client verschickt Meldung**

Von einem Spiel aus kann entweder auf eine erhaltene Meldung geantwortet werden:

```
Message outMessage = inMessage.createResponse("LoginFailed");
outMessage.set("message", "Password is wrong!");
getGame().sendToClient(outMessage);
```

**Abbildung 8: Beispiel: Server antwortet auf eine Meldung**

Dabei wird ähnlich wie bei einem E-Mail ein 'In-Reply-To'-Attribut gesetzt, mit dem der Client die Antwort-Meldung einer Anfrage-Meldung zuordnen kann.

Das Spiel kann aber auch gezielt eine Meldung an einen Client schicken, der mit einer bestimmten Rolle verknüpft ist:

```
Message outMessage = Message.createMessage("myAddressMessageType");
outMessage.set("name", "david");
outMessage.setTo(inRole.getHandle());    // addressieren
getGame().sendToClient(outMessage);
```

**Abbildung 9: Beispiel: Server verschickt eine Meldung**

Messages werden mittels Java-Serialisierung verschickt, was uns jegliche Encoding-Probleme erspart. Allerdings bringen Messages als serialisierte Objekte, die zudem noch zusätzliche Adressierungsinformationen enthalten, einen gewissen Overhead mit sich, wodurch sie für Real-Time-Interaktionen (z.B. Sprachübertragung) eher ungeeignet sind.
Auf Message-Persistenz wird im Kapitel 5.4.2. näher eingegangen.

### 5.3.2. Connection

`dfrank.game.client.common.utils.connection.impl.ClientConnectionSocketImpl`
`dfrank.game.server.utils.connection.impl.ServerConnectionSocketImpl`

Ein Connection-Objekt baut von der Client-Seite eine Verbindung zu einem bestimmten Server auf und hält auch eine Referenz auf die Verbindungsinformationen (im Fall der Socket-Implementation auf `java.io.InputStream` und `OutputStream`) und übernimmt Versendung und Empfang von Nachrichten. Auf der Server-Seite ist das Connection-Objekt für die Annahme von eingehenden Verbindungen zuständig – die Socket-Implementation startet dazu beispielsweise einen Thread, der auf einem vorgegebenen Port Verbindungen akzeptiert. Die Information, wie ein Client 'zurückgerufen' werden kann (`OutputStream` und `Connection`) wird jedoch vom Spooler für die Connection verwaltet.
Die Prototypen einer Connection über RMI und einer 'lokalen' Connection, die ohne Netzwerk eine Kommunikation zwischen Client und Server in der gleichen virtuellen Maschine erlaubt, liegen bei.

### 5.3.3. Spooler

`dfrank.game.client.common.utils.spooler.ClientSpoolerImpl`
`dfrank.game.server.utils.spooler.ServerSpoolerImpl`

Der Spooler speichert ausgehende Meldungen in einer Warteschlange, die in einem eigenen Thread abgearbeitet wird und erlaubt so das asynchrone Versenden von Meldungen. Der Client-Spooler identifiziert sich beim Server, indem er jeder Meldung einen selbstgewählten, einmaligen Client-Namen beilegt. Der Server-Spooler speichert dann die Verbindungsinformationen zum Client unter dessen Client-Namen ab. Bei einem Verbindungsunterbruch (bzw. wenn das Versenden einer Meldung misslingt) weist der Client-Spooler die Client-Connection an, eine neue Verbindung aufzubauen. Meldungen im Spooler gehen deshalb bei einem Verbindungsunterbruch nicht verloren, sofern Client und Server weiterlaufen.
Der Client-Spooler kann den Sender auch blockieren, bis vom Server eine Antwort zur versendeten Meldung eingetroffen ist (synchrone Meldungen).

### 5.3.4. Session

`dfrank.game.client.common.utils.session.ClientSessionImpl`
`dfrank.game.server.utils.session.ServerSessionImpl`

Die Server-Session leitet eingehende Meldungen einem bestimmten Spiel weiter. Sie kann der Client-Session eine Liste aller laufenden Spiele übermitteln, worauf diese ausgehende Meldungen an ein bestimmtes Spiel adressieren kann.
Die Client-Session bietet zudem eine Login-Schnittstelle an, mit der der Client unter Angabe von Benutzername und Passwort in ein Spiel einloggen kann. Serverseitig wird diese Login-Meldung aber nicht von der Server-Session behandelt, sondern wie jede andere Meldung transparent ans Spiel weitergeleitet, wo sie in unserer Implementation an den `RoleManager` des jeweiligen Spiels zur Bearbeitung weitergeleitet wird.

Das Sequenzdiagramm in Abbildung 10 zeigt, wie eine ClientSession benutzt werden kann, um Meldungen via ClientSpooler und ClientConnection zu verschicken. Um (ab Schritt 8) eine Meldung empfangen zu können, muss der Benutzer der Client-Session das Interface `ClientSessionUser` implementieren und mit der Session durch `aSession.setClientSessionUser()` verbunden werden. Es wird nur ein User pro

Client-Session unterstützt. Hingegen kann ohne diese Einschränkungen der synchrone Aufruf `aSession.sendToServerAndWaitForAnswer(msg)` benutzt werden.

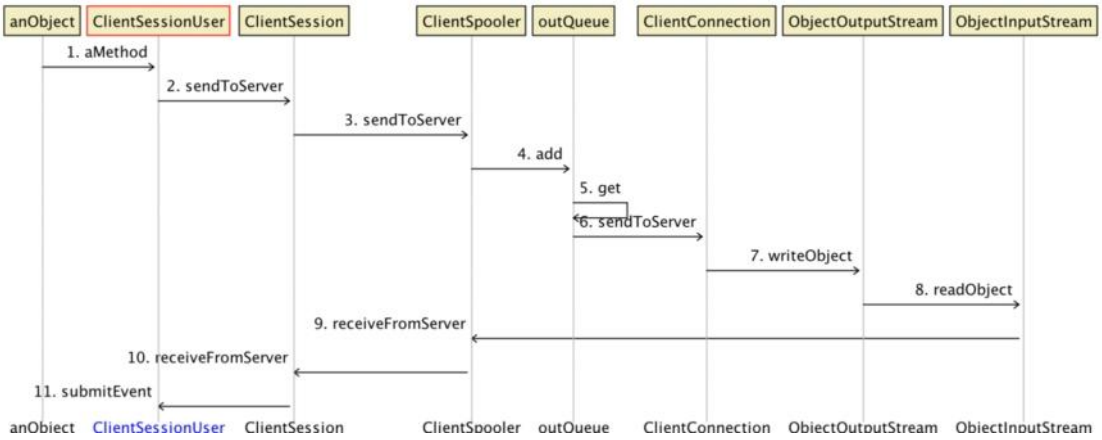



**Abbildung 10: Sequenzdiagramm: Client-Kommunikation**

Das Sequenzdiagramm in Abbildung 11 soll verdeutlichen, wie vom Spiel aus via ServerSession, ServerSpooler und ServerConnection Meldungen zum Client geschickt werden können und (ab Schritt 9) empfangene Meldungen vom `EventManager` abgearbeitet und vom `EventProcessor` an die Spiel-Schritte übergeben wird.

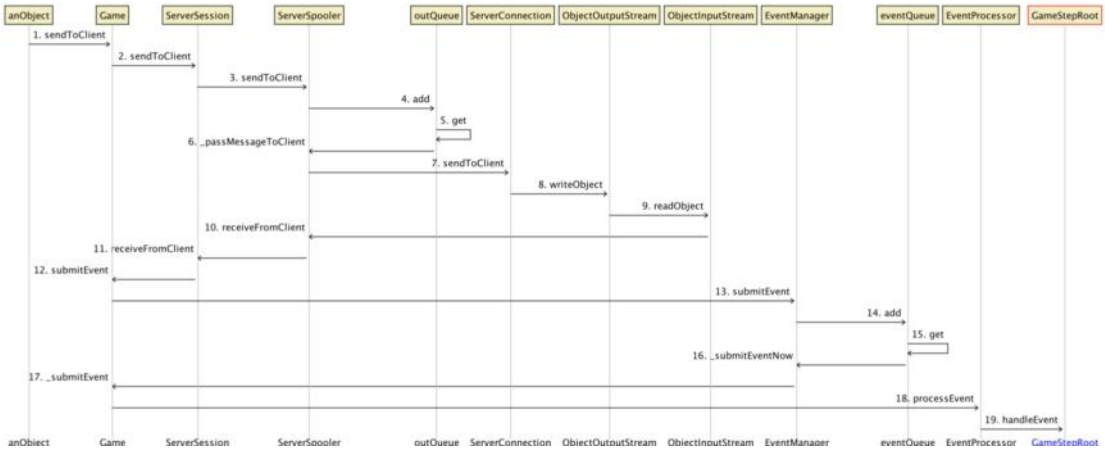



**Abbildung 11: Sequenzdiagramm: Server-Kommunikation**

## 5.4. GAME

`dfrank.game.server.impl.AbstractGame`
`dfrank.game.server.impl.GameImpl`

### 5.4.1. Struktur

Im Prinzip muss ein Spiel nur Meldungen empfangen und versenden können und braucht dazu nichts als eine zugewiesene Session.

Zusätzliche Dienste stehen aber bereit (vgl. auch Abb. 5):

- Jedes Spiel hat einen eigenen `RoleManager`, der die Authentifizierung der Benutzer übernimmt und die Anzahl der Mitspieler einschränken kann. Er verwaltet überdies die `Roles`, welche Benutzer repräsentieren – unabhängig davon, ob sie momentan angemeldet sind.
- Jedes Spiel hat einen `EventManager`, der eingehende Meldungen, in einer Warteschlange zwischenspeichert und sie auch in einer Datenbank abspeichern kann. Meldungen werden zur Ausführung ans `Game`-Objekt zurückgegeben, das sie entweder selber bearbeitet (z.B. im Fall von Login-Anfragen dem `RoleManager` übergibt) oder dem `EventProcessor` überlässt, der sie einem aktiven `GameStep` weiterreicht.
- Schliesslich verfügt jedes Spiel über einen `AssetManager`, der `Assets` und `Actions` verwaltet, an Clients verteilt und mit Hilfe des `AssetPersistor` auch abspeichern kann.

Von all diesen Diensten kann entweder die Standardimplementation verwendet werden oder für jedes Spiel eine eigene Implementation benutzt werden.

### 5.4.2. GameManager

`dfrank.game.server.impl.GameManager`
`dfrank.game.server.impl.GamePersistorSQLImpl`

Spiele werden durch den GameManager instanziiert, gestartet und beendet. Ihre genaue Struktur wird durch zwei XML-Dateien festgelegt. Die eine – auch Spiel-Template genannt – bestimmt, welche Implementationen von RoleManger, EventManager, EventProcessor und AssetManger verwendet werden und enthält die Verbindungsinformationen zur Datenbank für die Asset-Persistenz (Abb. 12 zeigt eine Standardkonfiguration). Die andere XML-Datei enthält die Spiel-Schritte und wird in Kapitel 6 besprochen. Beide XML-Konfigurationen benutzen den Long-

Term-Persistence-Mechanismus [24] von Java 1.4. Bei jedem Spielstart werden diese zwei XML-Dateien zusammen mit dem Namen des Spieles und dem aktuellen Datum in einer Datenbank festgehalten. Dem GameManager wird dazu der `GamePersistor` zur Seite gestellt, welcher auch vom EventManager des Spiels benutzt werden kann, um laufend die eingehenden Meldungen abzuspeichern.

```
<?xml version="1.0" encoding="UTF-8" ?>
<java version="1.4.0" class="java.beans.XMLDecoder">

    <object class="dfrank.game.server.impl.GameImpl" method="createGame">
        <object class="dfrank.game.server.impl.AssetManagerImpl" method="createAssetManagerFor">
            <object class="dfrank.game.server.impl.AssetPersistorSQLImpl" method="createAssetPersistor">
                <string>jdbc:mysql://localhost/game_assets</string>
                <string>org.gjt.mm.mysql.Driver</string>
                <string></string> <!-- username -->
                <string></string> <!-- password -->
            </object>
            <long>-1</long>
            <string>%gamename%</string>
        </object>
        <void property="Name">
            <string>%gamename%</string>
        </void>
        <void property="EventManager">
            <object class="dfrank.game.server.impl.EventManagerImpl" method="createEventManager"/>
        </void>
        <void property="EventProcessor">
            <object class="dfrank.game.server.impl.EventProcessorImpl" method="createEventProcessor"/>
        </void>
        <void property="RoleManager">
            <object class="dfrank.game.server.impl.RoleManagerImpl"/>
        </void>
    </object>

</java>
```

**Abbildung 12: GameManger: Allgemeine Game-Konfiguration**

Das fertig instanziierte Spiel muss dann nur noch an eine ServerSession gebunden und mit der Game-Methode `start()` gestartet werden. Der GameManger kann einem Spiel zur Laufzeit eine andere Session zuweisen und so z.B. den Port wechseln.
Beim Spielstart ist es möglich, dem Spiel eine Liste von Meldungen zu übergeben, die beispielsweise aus einer Datenbank stammen können. Damit kann ein unterbrochenes Spiel mit den Angaben, die der `GamePersistor` gespeichert hat, reaktiviert werden. Allerdings funktioniert das nur, wenn die serialisierten Daten wieder ausgelesen werden können[2].
Kapitel 6 beschreibt die Administrations-Schnittstelle zum GameManager.

[2] Java garantiert nicht, dass das nach grösseren Java-Versionsnummernwechseln funktioniert.

## 5.5. ASSETS UND ACTIONS

Wie bereits dargelegt, sind Assets und Actions ein Hilfsmittel zur Verteilung von Informationen und zur Zuweisung von Handlungsmöglichkeiten an den Client. Die Möglichkeiten die sie bieten sind ähnlich breit wie diejenigen von WebServices oder RPC. Die Benutzung dieser Hilfsmittel ist für jedes Spiel natürlich fakultativ. Assets werden allerdings auch von den Klassen `AbstractGame` und `RoleImpl` zur Speicherung ihrer privaten Daten verwendet.

### 5.5.1. Asset-Struktur

`dfrank.game.server.impl.AssetImpl`

Assets sind sehr ähnlich aufgebaut wie Messages, indem sie serialisierbare Datencontainer sind, in die beliebige Daten unter einem Bezeichner abgelegt werden können. Die Implementation benutzt wieder eine HashMap. Wie Messages haben auch Assets einen Typ und eine ID, die bei persistenten Assets dem Primärschlüssel in der Datenbank entspricht und bei nicht-persistenten eine fortlaufende negative Laufnummer ist. Assets sollten nie direkt instanziiert werden (ihr Konstruktor ist dementsprechend geschützt) sondern sie werden vom AssetManager angefordert. Der AssetManager indiziert alle Assets nach Typ und Eigentümer und erlaubt es, nach diesen Kriterien nach Assets zu suchen. Abbildung 13 zeigt Beispielscode für den gesamten Lebenszyklus eines Assets.

```
// erstellen
Asset as = getAssetManager().createAsset("myType");

// mit Inhalt füllen und auslesen
as.set("numberOfPoints", new Integer(100));
Integer pt = (Integer)as.get("numberOfPoints");

// finden
as = getAssetManager().getAssetWithID(as.getID())
List assetList = getAssetManager().getAssetsWithType("myType");
assetList = getAssetManager().getAssetsWithOwnerID("myType");

// löschen
getAssetManager().deleteAssets(assetList);
getAssetManager().deleteAsset(as);
```

**Abbildung 13: Beispiel: Assets erstellen, finden und löschen**

### 5.5.2. Asset-Synchronisation

`dfrank.game.server.impl.AssetManagerImpl`

Um ein Asset einem Spieler zuzuweisen wird der betreffende Spieler zum Besitzer (Owner) des Assets erklärt. Ein Asset kann nur einen Besitzer haben – es existieren aber vier spezielle Besitzer-Gruppen, nämlich "keiner", "alle", "admin" oder "user".

```
Asset msg = getAssetManager().createAsset("AT_NEWS");
msg.set("Text:", "this is a message to all administrators");
msg.synkToAdmin();   // alle Administratoren bekommen eine Kopie!
// msg.synkToUsers();
// msg.synkToAll();
// msg.synkToOne(roleID);
// msg.dontSynk();
```

**Abbildung 14: Beispiel: Asset-Synchronisation**

Jedes Asset kennt seinen AssetManager und meldet ihm, falls es einen Besitzer hat, jegliche Datenänderungen weiter. Der AssetManager sendet dann eine Meldung vom Typ '_MT_ASSET_SYNK' an die betroffenen Besitzer mit einer genauen Beschreibung, welches Asset neu geschaffen oder gelöscht bzw. welches Attribut hinzugefügt, geändert oder entfernt wurde. Wie Assets beim Client ankommen steht in Kapitel 5.7.

### 5.5.3. Asset-Persistenz

`dfrank.game.server.impl.AssetPersistorSQLImpl`

Werden Assets als persistent markiert, werden die Synchronisationsinformationen auch an den `AssetPersistor` weitergeleitet, der entsprechende Datenbankanweisungen generiert. Jedes Asset-Attribut wird dann einzeln serialisiert in der Datenbank abgelegt. Zusätzlich werden eine menschenlesbare String-Repräsentation, ein Zeitstempel und der Verweis auf das zugehörige Asset und Game mitgespeichert. Das Datenbank-Schema (bzw. die nötigen SQL-Anweisungen, um die Tabellen mit MySQL anzulegen) liegt dem Quellcode bei. Wie schon bei der Game-Persistenz sei auch hier auf mögliche künftige Kompatibilitätsprobleme hingewiesen, indem unter Java 1.4 serialisierte Objekte mit späteren Versionen vielleicht nicht lesbar sind. Assets können in der jetzigen Implementation nicht nachträglich persistent gemacht

werden, sondern nur von Anfang an persistent (vgl. Abb. 15) oder transient angelegt werden.

```
Asset msg = getAssetManager().createPersistentAsset(AT_NEWS);
```

**Abbildung 15: Beispiel: Asset-Persistenz**

### 5.5.4. AssetActions

`dfrank.game.server.impl.AssetActionImpl`

Actions sind aus der Sicht des Servers nichts als Attribute mit speziellen Namenskonventionen, für deren Erstellung ein paar Hilfsmethoden zur Verfügung stehen. Die Klasse `AssetAction` wird dabei (analog zu `Message` und `Asset`) wiederum als 'Wrapper' um eine HashMap implementiert, die alle Eigenschaften einer einzelnen Aktion speichert. Beliebig viele solcher Actions können unter dem reservierten Attribut '_action_list' in jedem Asset gespeichert werden.
Eine Action spezifiziert ähnlich einem HTML-Formular Attribut-Namen und -Typen die vom Client mit Daten gefüllt an den Server zurückgeschickt werden können. Abbildung 16 zeigt ein einfaches Formular, das vom Client dem Benutzer als Textfeld mit der Beschriftung "Your Message:" und mit einer Absende-Taste "Send" präsentiert werden kann.

```
Asset chat = getAssetManager().createAsset("AT_CHAT");
chat.set("_name", "Chat");

    AssetAction input = AssetActionImpl.createAction("MT_CHAT_TXT");
    input.set("_name", "Your Message");
    input.set("_info", "Send a mesage to everybody.");
    input.set("_type", "generic");
    input.set("_submit", "Send");
    input.addArgument("ARG_CHAT_TEXT", "TextArea", null,
        "Your Message:", "Type a text and click 'Send'!");

chat.addAction(input);
chat.synkToAll();
```

**Abbildung 16: Beispiel: 'Chat'-Action an Asset anhängen**

Die Methode `addArgument()` kann wiederholt aufgerufen werden, um kompliziertere Formulare zusammenzustellen. Ihre fünf Argumente bedeuten in der Reihenfolge ihres Auftretens:

- Name des Message-Attributs; als Konvention kann man optionale Attribute dabei in Klammern setzen.
- Typ des Arguments ("TextArea", "ComboBox", "TextField" oder "CheckBox")
- Zusatzdaten (z.B. eine Liste von Strings für die ComboBox)
- Beschriftung
- Zusätzlicher Beschreibungstext (z.B. als ToolTips darstellbar).

Tabelle 3 zeigt, welche Meldung bei der Ausführung der in Abbildung 16 beschriebenen Action verschickt wird.

| **Attribut-Name** | **Attribut-Wert** |
|---|---|
| "_message_type" | "MT_CHAT_TXT" |
| "ARG_CHAT_TEXT" | "Mein ins Textfeld geschriebener Text." |

**Tabelle 3: Aus einer AssetAction generierte Meldung**

## 5.6. GAMESTEPS

`dfrank.game.server.gamesteps.*`

Das Gegenstück zu den Actions, die ja ganz bestimmte Meldungsformate als willkommen anpreisen, sind die Spiel-Schritte, die signalisieren können, welche Meldungstypen sie verarbeiten können. Die Meldungsverarbeitung im Spiel läuft so ab, dass zuerst das Spiel-Objekt Meldungen zum Aussortieren bekommt. Diese Möglichkeit wird momentan ausschliesslich dazu genutzt, um Login-Begehren an den RoleManager umzuleiten. Dann werden sämtliche Messages an den EventProcessor weitergeleitet. Dieser leitet sie an die Wurzel eines Baumes von Spielschritten weiter (`GameStepRootImpl`), die sie wiederum im Baum bis an die Blätter weiterreichen kann. Die Struktur des Baumes kann beim Spielstart durch eine XML-Datei bestimmt werden (vgl. Kap. 6). Die Struktur der Spielschritte wird nachfolgend beschreiben – praktische Anwendungen folgen in Kapitel 6 und 7.

### 5.6.1. Struktur eines einzelnen Spiel-Schrittes

Spiel-Schritte werden in der Regel von `AbstractGameStep` abgeleitet und folgen drei einfachen Prinzipien:

- GameSteps haben einen **Lebenszyklus**

GameSteps werden bei Spielbeginn alloziiert (vgl. Kapitel 5.4.2.). Dabei wird ihr Default-**Konstruktor** aufgerufen und Initialiserungsparameter werden gesetzt – zu dem Zeitpunkt sind jedoch noch keinerlei Dienste (z.B. AssetManger oder RoleManager) aktiv. Bei Spielbeginn, wenn diese Dienste bereitstehen, wird für jeden GameStep die Methode **`_start()`** aufgerufen. Jeder GameStep kann entweder *aktiv* oder *passiv* sein. Bei der Aktivierung wird seine Methode **`_init()`**, bei der Passivierung **`_exit()`** aufgerufen. Über den Zeitpunkt der Aktivierung und Passivierung entscheidet der übergeordnete GameStep. Ein GameStep kann aber auch jederzeit seine Passivierung verlangen, indem er **`childHasFinished()`** des übergeordneten GameStep aufruft.

- GameSteps bestimmen, welche **Meldungs-Typen** sie erhalten wollen

Durch Aufruf von **`addHandledEventType()`** (in der Regel im `_init()`) und **`removeHandledEventType()`** (im `_exit()`) kann ein GameStep bestimmen, welche Meldungs-Typen er zugeteilt erhält. Wenn eine Meldung mit einem entsprechenden Typ eintrifft wird sie durch Aufruf von **`handleEvent(Message, Role)`** dem GameStep übergeben. Jeder GameStep hat jederzeit direkten Zugriff auf das Game-Objekt und auf den AssetManager.

- GameSteps können **andere Spiel-Schritte enthalten**

GameSteps können Container für andere GameSteps sein – dazu leitet man sie am einfachsten von `AbstractGameStepContainer` ab. Container werden dabei als 'Eltern', ihr Inhalt als 'Kinder' bezeichnet. Die wichtigsten bereits implementierten Container sind im nächsten Kapitel aufgeführt.

### 5.6.2. GameStep-Beispiel

Das nachfolgende Code-Beispiel zeigt, wie wenig Aufwand der Server betreiben muss, um auf die in Abbildung 16 vorgestellt 'Chat'-Action zu antworten. Der ChatGameStep registriert sich für die eingehenden Chat-Meldungen vom Typ "MT_CHAT_TXT" und schickt sie an alle Clients weiter.

```
public class ChatGameStep extends AbstractGameStep {
  protected void _init() {
    addHandledEventType("MT_CHAT_TXT");
    // hier sollte die Chat-AssetAction von Abb.16 verteilt werden.
  }

  public void handleEvent(Message inMessage, Role inRole) {
    Asset chatmsg = getAssetManager().createAsset("AT_CHAT");
    chatmsg.set("_name", "Chat-Meldung");
    String titel = inRole.getUserName() + " hat gesagt:";
    chatmsg.set(titel, (String)inMessage.get("ARG_CHAT_TEXT"));
    chatmsg.synkToAll();
  }

  protected void _exit() {
    removeHandledEventType("MT_CHAT_TXT");
  }
}
```

**Abbildung 17: Beispiel: Eingehende Meldung an alle Clients weiterschicken**

### 5.6.3. Wiederverwendbare Spiel-Schritte

Es stehen die folgenden fünf Container-Typen zur Verfügung, die als Inhalt einen oder mehrere GameSteps haben können, die ihrerseits Container sein können:

- `dfrank.game.server.gamesteps.`**`GSIteration`**
  Dieser Spiel-Schritt wiederholt ein einziges Kind eine vorgegebene Anzahl Male. Das Kind bestimmt selbst, wann eine Iteration zu Ende ist.

- `dfrank.game.server.gamesteps.`**`GSSequence`**
  Dieser Spiel-Schritt hängt mehrere Kinder hintereinander. Ein Kind wird Aktiv, sobald sein Vorgänger sich selbst beendet hat.

- `dfrank.game.server.gamesteps.`**`GSIterationSequence`**
  Dies ist eine Kombination der vorangegangenen zwei GameSteps – eine Abfolge von Schritten wird mehrmals wiederholt.

- `dfrank.game.server.gamesteps.`**`GSSwitch`**
  Alle Kinder eines Switch-Containers sind gleichzeitig aktiv. Messages werden dem ersten GameStep übergeben, der den jeweiligen Meldungs-Typ verarbeiten kann. Ruft eines der Kinder `getParent().childHasFinished()` auf, dann werden alle Kinder passiviert.

- `dfrank.game.server.gamesteps.`**`InterruptableGS`**
  Dieser Spielschritt enthält ein einziges Kind und kann entweder durch den Administrator unterbrochen (passiviert) werden oder es kann eine Zeit festgelegt werden, nach der automatisch unterbrochen wird. Clients werden durch ein Asset vom Typ 'AT_CLOCK' über die Restzeit informiert und der Administrator kann den Countdown anhalten, verschnellern, verlangsamen oder einen sofortigen Abbruch erwirken.

## 5.7. CLIENT

### 5.7.1. Allgemeines

Der Client kann wie in Abbildung 10 bereits gezeigt mit dem Server kommunizieren. Dazu benutzt er ein Connection-Objekt, um den Server anzuwählen und ein Session-Objekt, um sich bei einem bestimmten Spiel einzuloggen. In der Folge wird er vom Server die in 5.5.2 erwähnten Asset-Synchronisations-Meldungen erhalten. Der Client kann als ClientSessionUser (d.h. als Verarbeiter einkommender Meldungen) einen (Client-)AssetManger einsetzen. Die eingegangenen Assets und Actions stehen dann dem Client mit den gleichen Schnittstellen zur Verfügung wie dem Spiel. An den AssetManager kann zudem ein `AssetBroker` gebunden werden, der Änderungen der Assets an beliebig viele Benutzerschnittstellen-Elemente (bzw. `AssetObserver`) weiterleitet (vgl. auch Abbildung 5).
Die Benutzerschnittstelle kann vom (Client-)AssetManager durch Aufruf der Methode `getAssets()` auch aktiv eine Liste aller Assets anfordern und sie geeignet darstellen.

### 5.7.2. Asset-Präsentation und -Filterung

Assets können von den bereitgestellten Client-Implementationen einerseits generisch dargestellt werden. Dabei wird erwartet, dass jedes Asset ein Attribut "_name" besitzt, das als Titel in einer Liste aufgeführt wird. Die Detail-Ansicht eines Assets zeigt hingegen alle Attribute an, welche nicht mit einem Unterstrich ('_') beginnen. Im Chat-Beispiel von Abbildung 17 könnten die vom Server verteilten Meldungen demnach als Liste von Elementen angezeigt werden, die alle mit "Chat-Meldung" beschriftet sind. Ein Maus-Click auf ein Listen-Element würde dann den Asset-Inhalt der Chat-Meldung anzeigen. Mit den Daten aus Tabelle 2 könnte der so aussehen: "David hat gesagt: Mein ins Textfeld geschriebener Text."

Eine von `AbstractAssetObserver` abgeleitete Filter-Klasse kann aber auch an den `AssetBroker` gebunden werden und z.B. ausschliesslich Assets vom Typ "AT_CHAT" abonnieren und in einer spezifischen Form (z.B. tabellarisch) darstellen.

### 5.7.3. Swing-Client

Der Swing-Java-Client stellt im Paket `dfrank.game.client.swing` speziell auf Swing-Präsentationsformen (JTable, JList und JTree) zugeschnittene, abstrakte AssetObserver bereit. Sie implementieren DatenModell-Schnittstellen die von den erwähnten Swing-Klassen unmittelbar darstellbar sind. Sie werden im Paket `dfrank.game.rog.swingclient.models` vom Realoptionen-Spiel benutzt.
Im Paket `dfrank.game.rog.swingclient.swingui.action` stellt die Klasse `ActionTreeGUI` eine Liste aller Assets mit den ihnen angehängten Actions als JTree dar und präsentiert Asset-Inhalte tabellarisch und Action-Formulare als benutzbare Dialog-Boxen. Abbildungen dazu finden sich in Kapitel 6 und 7.

Der Swing-Client bietet weiter eine Fenster-Infrastruktur an, mit der sich die unterschiedlichen Asset-Repräsentationen nebeneinander darstellen lassen.
Um das Testen von Spielen zu erleichtern, erlaubt der Swing-Client beliebig viele Client-Threads (d.h. vollwertige Client-Umgebungen) in der gleichen Applikation zu starten. Er kann auch so konfiguriert werden, dass beim Aufstarten eine vorgegebene Anzahl Client-Threads automatisch auf ein Spiel einloggen.
Für die Swing-Benutzerschnittstelle wurden keinerlei GUI-Editoren benutzt.

### 5.7.4. Web-Client

Für den mobilen Client wird im Paket `dfrank.game.client.web` mit der Klasse `WebClient` ein sehr schlanker Wrapper um eine ClientSession und einen AssetManager gelegt. Dieses Objekt biete die Methoden `connect(servername, port)` an, um Verbindung mit einem Spiel-Server aufzunehmen und die Methode `login(spielname, username, passwort)`, um in ein Spiel einzuwählen. Daneben erlaubt es nur noch den Zugriff auf den AssetManager (zur Asset-Darstellung) und auf die ClientSession (zum Versenden von Messages).
Die Java Server Pages konstruieren aufgrund von Formulareingaben ein WebClient-Objekt und speichern es in einer JSP-Session ab, wodurch es auf allen nachfolgenden Seiten verfügbar bleibt.
Der JSP-Output besteht aus reinem, UTF-8 codiertem XHTML für Mobilgeräte und wurde gegen die offizielle Spezifikation [25, 26, 27] validiert. Stylesheets, welche die Ausgabe gerätespezifisch anpassen könnten werden zurzeit nicht eingesetzt.
Der Web-Client wurde erst mit Web-Browsern von Desktop-Rechnern getestet und nicht mit mobilen Geräten.

## 5.8. WERKZEUGE

Ganz kurz sollen hier die Werkzeuge genannt werden, die während der Implementations-Phase benutzt wurden.

- Als Betriebssystem kam *Mac OS X* 10.3.5. zum Einsatz, das sämtliche Servermerkmale bietet, die für den Spiel-Server nötig sind. Die genannte Plattform enthält bereits eine aktuelle *Java*-Umgebung der Version *1.4.2_5.* Für andere Plattformen ist diese frei erhältlich [28].
- Die mitgelieferte Entwicklungsumgebung XCode unterstützte in der anfangs Jahr verfügbaren Version 1.1. Java nur rudimentär, weshalb nach einer Java-Entwicklungsumgebung mit Unterstützung für das Build-System Ant [43] Ausschau gehalten wurde. Nach Versuchen mit NetBeans 3.6 und Eclipse 3.0M7 wurde schliesslich *IntelliJ IDEA 4.0* [29] als Entwicklungsumgebung gewählt. Insbesondere deren Refactoring-Unterstützung erwies sich für das Projekt als äusserst hilfreich. Während der Projektdauer wurde laufend auf jeweils aktuelle Entwicklungsversionen von IDEA 4.5 und später 4.5.1

gewechselt, die stets von erstaunlich hoher Qualität und Stabilität waren, und für dieses Projekt nützliche neue Funktionen (RMI- und Tomcat-Unterstützung) enthielten.

- Während der ganzen Implementations-Phase wurde *Subversion* [30] in der Version 1.0.0. bis 1.0.4 als Versionsverwaltungssystem eingesetzt, das sich mit *Svn-Up* 0.8 [31] nahtlos in IDEA einbinden liess.
- Bei der Verpackung der kompilierten Java-Klassen in JAR-Archive für die Verteilung der Endprodukte leistete *Class Wrangler* 1.5, das zur Metrowerks Entwicklungsumgebung CodeWarrior 8.0 [44] gehört, gute Dienste.
- Als Datenbank wurde *MySQL* 4.0.15 in der Distribution von ServerLogistics [32] verwendet, und mittels phpMyAdmin 2.5.5-pl1 [33] administriert. Von Java aus kann über den bei MySQL mitgelieferten JDBC-3.0 Treiber 'MySQL Connector/J' auf MySQL zugegriffen werden.
- Als Servlet-Container für die JSP-Schnittstelle des mobilen Clients wurde *Tomcat* 5.0.27 eingesetzt [34].

Sowohl die Spielumgebung als auch das Realoptionen-Spiel erfordern zur Laufzeit **nur** eine funktionierende Java 1.4 -Umgebung. Die HTML-Schnittstelle ist ein eigenständiges Programm, das einen Servlet-Container benötigt. Die Persistenzfunktionen arbeiten natürlich nicht ohne Datenbank – das Spiel läuft aber auch ohne sie.

Die Abhängigkeit von Java 1.4 entsteht namentlich aus der Verwendung von:

- `java.beans.XMLDecoder` zum Einlesen der XML-Konfigurationen
- `java.net.InetSocketAddress` und dem argumentlosen Konstruktor von `java.net.Socket` in `dfrank.game.client.common.utils.connection.impl.ClientConnectionSocketImpl`
- `String.replaceAll()` an diversen Orten
- `java.awt.datatransfer.TransferHandler` für Drag-and-Drop im Client
- `java.util.LinkedHashSet` in `dfrank.game.server.impl.AssetManagerImpl`

### 5.9. BIBLIOTHEKEN

Während auf Abhängigkeiten von exotischen bzw. infrastrukturell anspruchsvollen Technologien bewusst verzichtet wurde, soll doch nicht unerwähnt bleiben, an welchen Stellen öffentlich zugängliche Programmteile von Dritten benutzt wurden. Sämtliche der benutzten Bibliotheken liegen im Quellcode dieser Arbeit bei und dürfen für nicht-kommerzielle Zwecke frei verwendet werden. Nähere Angaben zum Copyright finden sich in den angegebenen Referenzen und teilweise direkt im Quellcode.

- Das Logging-System (zur Ausgabe von Fehlermeldungen) wurde mit Hilfe des Apache-Projekts 'commons logging', Version 1.0.4 [35] abstrahiert, sodass unterschiedliche Logging-Bibliotheken verwendet werden können.
- Als Logging-Bibliothek kommt darunter Log4J 1.2.8 [36] zur Anwendung.
- Zur Indexierung von Assets wurde die `MultiMap`-Datenstruktur aus dem Apache-Projekts 'commons collections', Version 3.1 [37] verwendet. Eine MultiMap ist eine Hash-Tabelle, die mehrere Einträge pro Hash-Schlüssel verwalten kann. Assets können damit nach Typ indiziert werden (es kann mehrere Assets vom selben Typ geben).
- Für den Swing-Client wurde der `ScrollableDesktop` von Tom Tessier verwendet [38], um eine Windows-artige MDI-Benutzerschnittstelle anzubieten.
- `TableSorter` zur automatischen Sortierung von `javax.swing.JTable` im Client ist zwar noch im Quellcode-Verzeichnis enthalten, wird aber vom Realoptionen-Spiel nicht benutzt. Es stammt aus den Swing-Tutorials von Sun [39].
- `FileTransferHandler` schliesslich ermöglicht Drag-and-Drop von Dateien in Textfelder im Client und stammt von David Flanagan [40].
- Der Quellcode für die MD5-Codierung von Passwörtern stammt aus einem Online-Forum [41].

# 6. IMPLEMENTATION 'ADMINISTRATIONS-SPIEL'

Um Spiele von überall her starten zu können, wurde ein Administrations-Spiel programmiert, das die Fernsteuerung des in Kapitel 5.4.2. beschriebenen GameManagers ermöglicht. Es stellt ein Asset mit vier Aktionen zur Verfügung, die es erlauben

- Ein Spiel zu starten (dessen XML-Konfiguration der GameSteps muss dabei angegeben werden)
- Ein Spiel zu stoppen
- Einem Spiel eine andere ServerSession zuzuteilen (d.h. den Port zu wechseln)
- Ein abgespeichertes Spiel aus der Datenbank zu reaktivieren.

Sämtliche Funktionalität dazu wird vom GameManager mit der Hilfe des GamePersistors bereitgestellt.



**Abbildung 18: JSP- und Swing-Administrations-Schnittstellen**

Zur Implementation wird ein einziger GameStep namens **`AdminGS`** benutzt. Es ist nicht sinnvoll, bereits beim Spielstart die vier Actions für den Administrator bereitzustellen, da der Administrator ja erst am Server-Zustand zum Zeitpunkt seiner Anmeldung interessiert ist. Dafür kommt uns gelegen, dass der `RoleManger` bei jeder Anmeldung eines Benutzers oder Administrators eine Meldung vom Typ 'MT_INIT_USER' bzw. 'MT_INIT_ADMIN' ins Spiel einspeist. `AdminGS` konstruiert also beim Erhalten einer Meldung vom Typ 'MT_INIT_ADMIN' die entsprechenden Actions und synchronisiert sie zum Administrator. Wie auch später beim Realoptionen-Spiel wird auch hier eine separate Klasse, **`AdminAF`**, verwendet, die als

'AssetFactory' zur Konstruktion der Assets statische Hilfsmethoden anbietet und so die Übersichtlichkeit des Quellcodes verbessert. Ebenso werden die allgemeinen Asset-Hilfsfunktionen benutzt, die `dfrank.game.server.gamesteps.UtilityAF` anbietet.

Eingehende Administrationsbefehle werden in der Methode `handleEvent()` von AdminGS ausgepackt und dem GameManager übergeben, der über das Game-Objekt erreichbar ist (`getGame().getGameManager()`).

Die Konfiguration des Administrations-Spiels ist sehr einfach. Zur allgemeinen Konfiguration wird das XML-Template von Abbildung 12 verwendet. Die zweite nötige XML-Beschreibung zur Komposition und Konfiguration der GameSteps fällt hier noch einfacher aus: als einziger GameStep muss `AdminGS` geladen werden, der als Parameter nur Benutzername und Passwort des Administrators benötigt. Das Passwort wird übrigens wie alle Benutzerpasswörter von der ClientSession nur als MD5-Digest an den Spielserver übertragen.

```
<?xml version="1.0" encoding="UTF-8" ?>
<java version="1.4.0" class="java.beans.XMLDecoder">

  <object class="dfrank.game.server.admin_gamesteps.AdminGS">
    <void method="setAdminUP">
      <string>admin</string>
      <string>topsecret</string>
    </void>
  </object>

</java>
```

**Abbildung 19: Admin-GameSteps: Admin_GS.xml**

# 7. IMPLEMENTATION 'REALOPTIONEN-SPIEL'

## 7.1. EINLEITUNG

Der Spielverlauf im Realoptionen-Spiel wird in allen Details in der Spezifikation (Kapitel 10/Anhang A) beschrieben. Ein Überblick soll hier genügen.
Jeder Spieler wird bei Spielbeginn mit einem Geldbetrag ausgestattet. Bei einer Auktion kann er die Nutzungsrechte an einem Ölfeld erstehen, wobei natürlich weniger Ölfelder existieren als Mitspieler. Spieler, die ein Ölfeld besitzen, können in den nachfolgenden Spielzyklen in Fördertechnologie investieren. Hier kann die Wahl ermöglicht werden zwischen mehreren Technologien, die zu verschiedenen Investitions- und Produktionskosten unterschiedliche Produktionskapazitäten aufweisen. Spieler, die in Technologie investiert haben, können darauf in jeder Spielrunde eine Produktionsmenge bestimmen, die zu Marktpreisen verkauft wird. Der Ölpreis ist abhängig vom Produktionsvolumen aller Teilnehmer und zufälligen Fluktuationen unterworfen und somit nicht genau vorhersehbar. Wenn ein Spieler (beispielsweise wegen tiefen Ölpreisen, welche die Förderkosten nicht decken) auf die Ölproduktion verzichtet, werden Produktionsabbruch-Kosten fällig und es sammeln sich Reparaturkosten an, die bei der Wiederaufnahme der Produktion bezahlt werden müssen. Nach mehreren Investitions-/Produktions-Zyklen können die Ölfelder erneut per Auktion unter den Teilnehmern verteilt werden.

Der zyklische Spielverlauf und das Prinzip, dass getätigte Investitionen (Ölfeld/ Bohrtechnologie) zusätzliche Handlungsmöglichkeiten verleihen, erlauben es in idealer Weise, den Spielverlauf des Realoptionen-Spiels durch Assets, Actions und GameSteps abzubilden. Das Kapitel 7.2. beschreibt diese Realisierung im Detail. Kapitel 7.3. und 7.4. zeigen nachher, wie die verschiedenen Client-Typen die Assets den Spielern präsentieren.

## 7.2. SERVER

### 7.2.1. GameStep-Struktur

Dis Spiel-Struktur lässt sich durch die in Kapitel 5.6.3. vorgestellten GameStep-Container einfach wiedergeben. Abbildung 20 zeigt dies graphisch auf.
In Kapitel 11 (Anhang B) ist die komplette XML-Konfigurationsdatei abgedruckt, welche die in Kapitel 7.2.3. aufgeführten Realoptionen-GameSteps exakt in dieser Art so zusammenfügen.

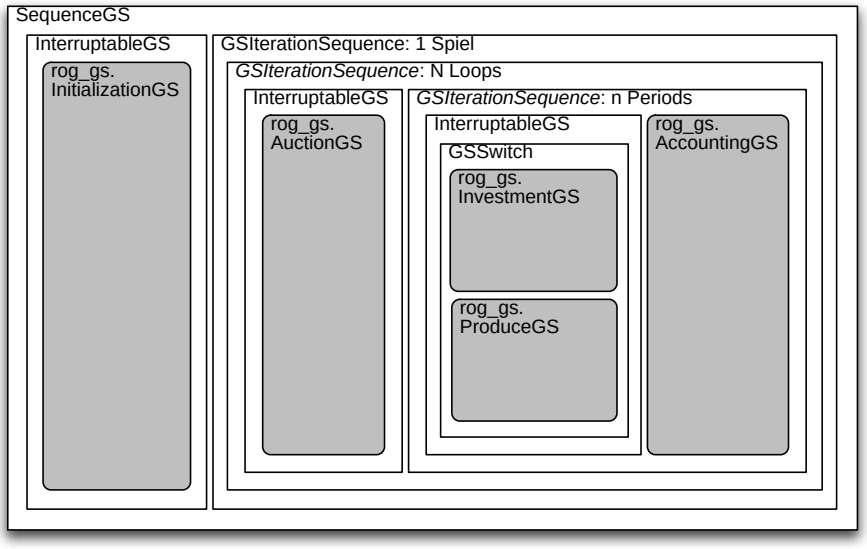



**Abbildung 20: Realoptionen-Spiel: GameStep-Struktur**

Das Spiel beginnt mit einer unterbrechbaren Vorphase, in der alle Spieler ins System einwählen. Danach beginnt das Spiel, das aus N Zyklen (Loops) besteht, die je mit einer Auktion beginnen und n Investitions-/Produktions-Perioden (Periods) beinhalten. Nach jeder dieser unterbrechbaren Perioden wird in einer 'Accounting'-Phase der Ölpreis berechnet und Bilanz gezogen. Spieler können dabei bankrott ausscheiden.

### 7.2.2. Asset-Strategie

Das Realoptionen-Spiel macht bei seiner Implementation ausführlichen Gebrauch von Assets und AssetActions. Dabei unterscheidet es vier Gruppen von Assets:

- **Spieler-Eigentum**

  Jeder Spieler besitzt ein **Bankkonto**, das durch ein Asset vom Typ 'AT_BANK_ACCOUNT' repräsentiert wird und kann im Laufe des Spiels ein **Ölfeld** erlangen, das ein Asset vom Typ 'AT_OILFIELD' ist. Während das Bankkonto nur jeweils den aktuellen Kontostand anzeigt, werden dem Ölfeld die jeweils erlaubten Aktionen angehängt (Investieren oder Fördern). In der Auktionsphase, in der noch kein Spieler ein Ölfeld besitzt, wird mit der **Auktions-Einladung** ein dritter Asset-Typ 'AT_AUCTION_INVITATION' verteilt, der eine AssetAction zur Abgabe von Auktions-Geboten transportiert.

- **Globale Informationen**

  Die zweite Gruppe besteht aus reinen Informations-Assets, die über die Details von Bohrtechnologien, Preisbildungsparametern und den Steuerfuss Auskunft geben, ohne dass ihnen eine Aktion angegliedert ist. Sie tragen die Typenbezeichnungen 'AT_INFO_TECHNOLOGY', 'AT_INFO_PRICE', bzw. 'AT_INFO_GENERAL'.

- **Messages/Log**

  Meldungen über den Spielverlauf (z.B. das Ende einer Periode) sowie persönliche Mitteilungen – beispielsweise über akzeptierte Auktions-Gebote, Konto-Veränderungen oder den Öl-Preis – werden durch Assets vom Typ 'AT_NEWS' repräsentiert.

- **Uhr**

  Schliesslich werden die Spieler vom verwendeten Standard-GameStep `InterruptableGS` durch ein Asset vom Typ 'AT_CLOCK' über den Ablauf von zeitbegrenzten bzw. unterbrechbaren Spielphasen informiert. Sowohl die Auktions-Phase als auch die Investitions-/Produktions-Perioden haben eine zeitliche Begrenzung, über deren Fortschreiten 'AT_CLOCK' informiert.
  Dem Administrator stehen dabei die in 5.6.3. beschriebenen Funktionen zur Zeitmanipulation zur Verfügung.

### 7.2.3. Spielspezifische GameSteps

Zur Bereitstellung und Manipulation der Assets und zur Meldungsverarbeitung werden GameSteps wie schon beim Administrations-Spiel in Kombination mit Asset-Factory-Hilfsklassen verwendet, welche die eigentliche Konstruktion der Assets übernehmen.
Die GameSteps im Realoptionen-Spiels haben folgende Aufgaben:

- **InitializationGS & InitializationAF**
  Informiert den Administrator, welche Benutzer bereits eingewählt sind und teilt jedem einwählenden Spieler ein 'AT_BANK_ACCOUNT' mit dem Startkapital zu. Bei Passivierung des InitializationGS (d.h. Beginn der ersten Auktion) wird die Anmeldung weiterer Benutzer gesperrt.
- **AuctionGS & AuctionAF**
  Verschickt 'AT_AUCTION_INVITATION' und behandelt Meldungen vom Typ 'MT_AUCTION_BID'. Ein Gebot pro Spieler wird gesammelt und bei der Passivierung des AuctionGS wird eine vorgegebene Anzahl Ölfelder nach dem Prinzip einer 'second-price sealed bid auction' vergeben (d.h. 'AT_OILFIELD'-Assets werden zugewiesen).
- **InvestmentGS & InvestmentAF**
  Bei Spielstart verschickt dieser Spiel-Schritt ein 'AT_INFO_TECHNOLOGY'-Asset an alle Mitspieler. Jedem Ölfeld, dem noch keine Technologie hinzugefügt wurde, wird bei jeder Aktivierung von InvestmentGS eine Action angehängt, mit welcher der Spieler eine 'MT_TECHNOLOGY_BUY'-Meldung verschicken kann. Bei eingehenden Meldungen dieses Typs wird – falls genug Geld auf dem Konto ist – dem 'AT_OILFIELD' die gewählte Technologie als Attribut angehängt.
- **ProduceGS & ProduceAF**
  Bei der Aktivierung stellt ProduceGS jedem Ölfeld, das bereits eine Bohrtechnologie hat, eine Action zur Versendung von 'MT_PRODUCE_OIL'-Meldungen bereit. Bei eingehenden Meldungen dieses Typs wird die beabsichtigte Produktionsmenge als Attribut im Ölfeld-Asset für die abschliessende Accounting-Phase notiert.

- **AccountingGS & Price**

  In diesem Spielschritt werden die Produktionsmengen aller Spieler aufsummiert, der davon abhängige Öl-Preis wird nach Spezifikation durch die Hilfsklasse 'Price' berechnet und für jeden einzelnen Spieler wird bilanziert, wie viel Gewinn oder Verlust aus seinem Verhalten resultiert.
  Für Spieler, welche nichts produziert haben, können ggf. Unterhaltskosten belastet werden. Bankrotten Spielern werden die Ölfeld-Assets entzogen.
  Nach Beendigung der Accounting-Phase folgt entweder die nächste Investitions-/Produktions-Phase, eine neue Auktion oder das Spiel-Ende.

### 7.2.4. RoleManager

Der im Realoptionen-Spiel verwendete RoleManager lässt während der Initialiserungs-Phase Benutzer mit beliebigen Benutzernamen/Passwort-Kombinationen zu. Nach dieser Phase können die Benutzer sich mit den gleichen Login-Daten beliebig oft von unterschiedlichen Orten aus einwählen. Nur der zuletzt eingewählte Client gilt dabei als aktiv und bekommt Assets zugeschickt.

## 7.3. JAVA-/SWING-CLIENT

Die folgenden Abbildungen zeigen, wie die Assets im Realoptionen-Spiel dargestellt werden.

### 7.3.1. Scrollable Desktop

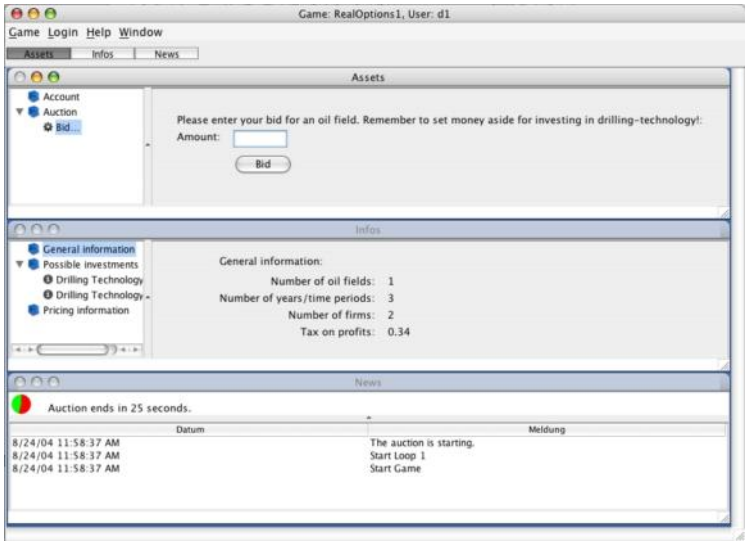



**Abbildung 21: Realoptionen-Spiel: Java-Client**

Die einzelnen Asset-Kategorien werden in getrennten, verschiebbaren Fenstern dargestellt, die sich auch einzeln ausblenden, vergrössern und automatisch anordnen lassen. Menus ermöglichen die Einwahl auf andere Server/Spiele. Dabei lassen sich in der jetzigen Testphase mit einem Client beliebig viele Fenster öffnen, die je eine vollständige Client-Umgebung darstellen.

### 7.3.2. Darstellung der Assets und Actions

Assets werden in einer Baumstruktur hierarchisch dargestellt. Die Auswahl eines Baumknotens zeigt seinen Inhalt an. Reine Informations-Assets und solche mit angegliederten Actions werden in verschiedenen Fenstern, aber ganz ähnlich dargestellt.

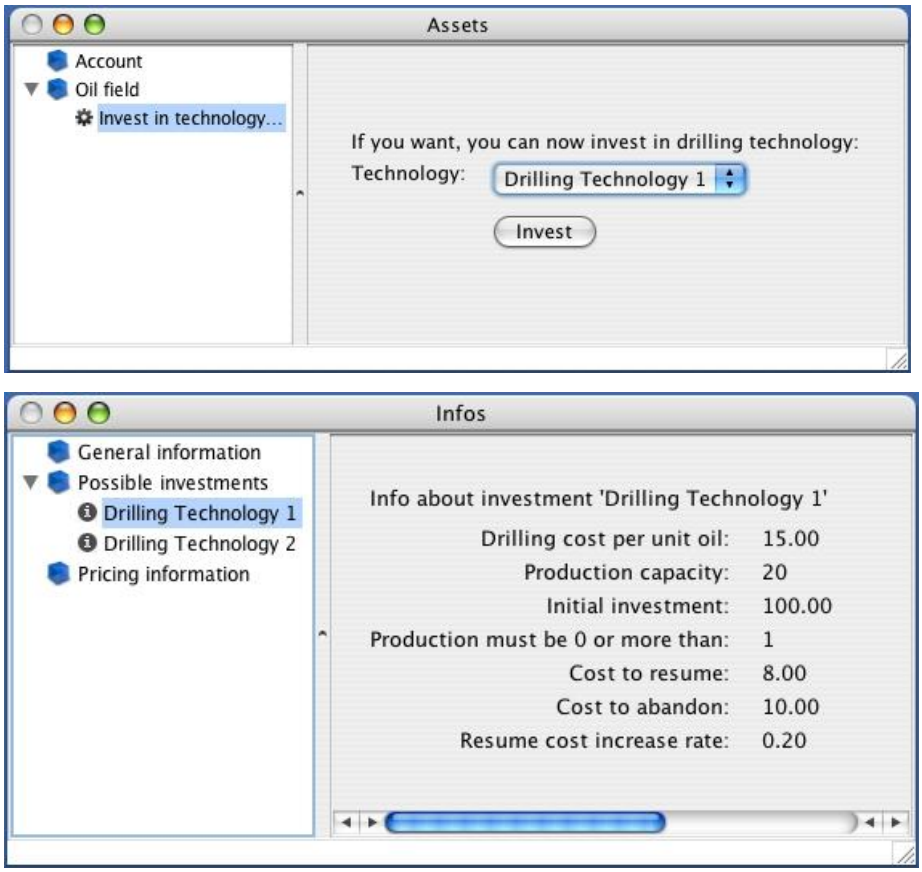



**Abbildung 22: Realoptionen-Spiel: Java-Client: Asset-Präsentation**

### 7.3.3. Darstellung der Meldungs-Tabelle und Uhr

Meldungen und Uhr werden im selben Fenster dargestellt. Bei den Meldungen steht die Aktuellste zuoberst.

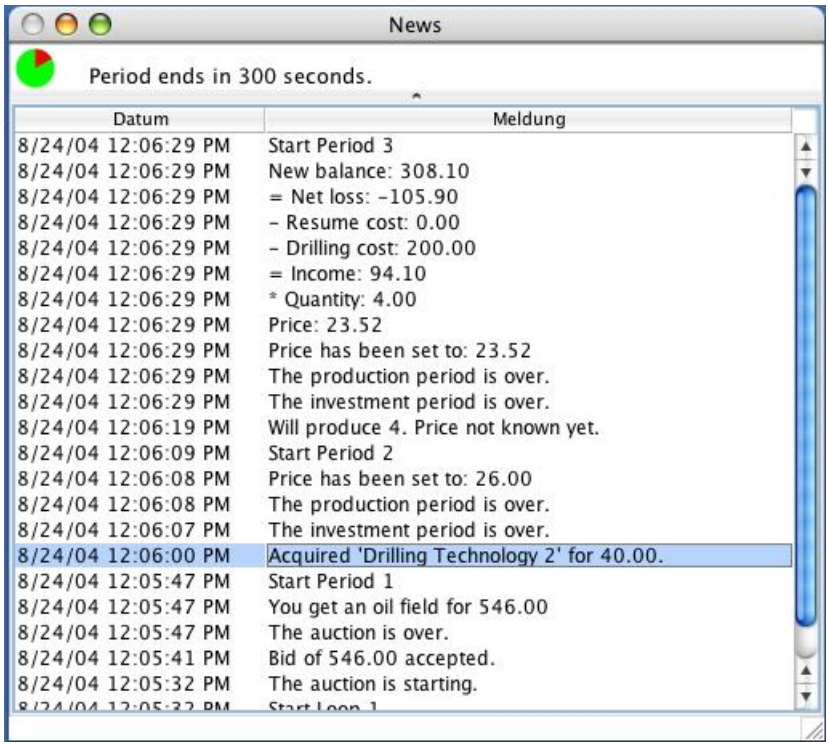
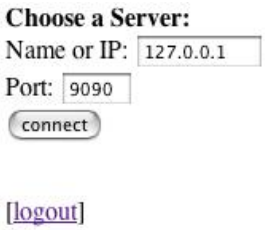



**Abbildung 23: Realoptionen-Spiel: Java-Client: Uhr und Meldungen**

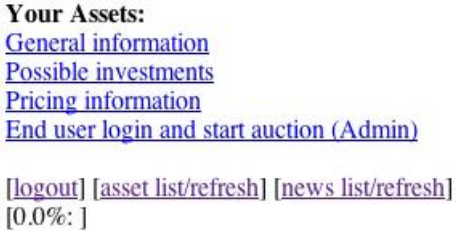
## 7.4. JSP-CLIENT

Abbildung 24 zeigt verschieden Phasen des Spiels in der Darstellung als Web-Seite.




**a) Server-Wahl**


**b) Spiel-Login**




**c) Asset-Überblick**

**Possible investments**
**Drilling Technology 1:**
Drilling cost per unit oil: 15.0
Production capacity: 20
Initial investment: 100.0
Production must be 0 or more than: 1
Cost to resume: 8.0
Cost to abandon: 10.0
Resume cost increase rate: 0.2

**Drilling Technology 2:**
Drilling cost per unit oil: 50.0
Production capacity: 10
Initial investment: 40.0
Production must be 0 or more than: 1
Cost to resume: 8.0
Cost to abandon: 10.0
Resume cost increase rate: 0.05

[logout] [asset list/refresh] [news list/refresh]
[0.0%: ] [asset/refresh]

**d) Investitions-Information**

**News:**
2004-08-24 12:21:22.461: hi admin!
2004-08-24 12:21:22.464: Login: admin
2004-08-24 12:21:22.478: Users in/max: 0/2
2004-08-24 12:15:27.833: Period length: 1.00
2004-08-24 12:15:27.834: Pricing constant: 1.50
2004-08-24 12:15:27.84: p(u): 0.50
2004-08-24 12:15:27.841: p(d): 0.50
2004-08-24 12:15:27.842: PD(o): 15.00
2004-08-24 12:21:22.461: hi admin!
2004-08-24 12:21:22.464: Login: admin
2004-08-24 12:21:22.478: Users in/max: 0/2

[logout] [asset list/refresh] [news list/refresh]
[0.0%: ]

**e) News-Liste**

**Oil field**
Field price: 88.0
**Invest in technology…:**
Technology: Drilling Technology 1 Choose one!
Invest

[logout] [asset list/refresh] [news list/refresh]
[6.941666666666668%: Period ends in 330 seconds.] [asset/refresh]

**f) Investitions-Action**

**Abbildung 24: Realoptionen-Spiel: Web-Client**

# 8. ZUSAMMENFASSUNG

## 8.1. ERREICHTE ZIELE

Innerhalb von einem halben Jahr ist mit rund 10'000 Codezeilen, aufgeteilt auf 900 Methoden, 130 Klassen und 23 Packages eine verteilte Spiel-Plattform entstanden, die verschiedensten Anforderungen gewachsen ist und unterschiedliche Netzwerk- und Präsentations-Technologien unterstützt. Die gewählten Konzepte haben sich bei ihrer Anwendung auf das Realoptionen-Spiel als adäquat erweisen und gezeigt, dass viel Flexibilität in ihnen steckt. Die Bewährung als Plattform für Experimente mit mobilen Clients steht noch aus.

Die riskante Strategie, sehr viel Arbeit in die Infrastruktur zu investieren, um erst am Schluss mit relativ wenig Aufwand das Realoptionen-Spiel darauf aufzubauen, ist voll aufgegangen.

In dieser letzten Phase ist in intensiver Zusammenarbeit mit Felix Morger eine solide "Version 1.0" entstanden, die wohl mit vergleichsweise geringen Anpassungen im Unterricht produktiv eingesetzt werden kann.

## 8.2. MÖGLICHE WEITERENTWICKLUNG

Natürlich ist die vorliegende "Version 1.0" noch keineswegs perfekt. Ihre verschiedenen Komponenten wurden während der Entwicklung ausführlich und systematisch getestet, und sind voll funktionsfähig. Kleinere Einschränkungen oder Weiterentwicklungsideen sind direkt im Quellcode als Kommentar mit dem Kürzel 'TODO' notiert.

In der Folge werden in loser Form bekannte Probleme und potentielle Weiterentwicklungen diskutiert.

- **Performance**
  - Wenn Server und mehrer Clients inklusive Entwicklungsumgebung gleichzeitig auf demselben Rechner laufen, ist die Performance eher mittelmässig. Schuld daran ist jedoch primär der Ressourcen-Hunger

von Java. Getrennte Rechner für Client und Server, sowie genug physischer Speicher lösen das Problem.
  - Es wurden nur informelle Performance-Messungen der Netzwerkschicht gemacht, die sehr positiv ausfielen. Verschiedene Thread-bezogene Parameter (Wartezeiten im Millisekunden-Bereich) könnten wohl noch optimiert werden.
  - Der EventManager speichert jede einzelne eingehende Meldung in eine Datenbank ab, was momentan nicht asynchron geschieht. Der zuständige GamePersistor sollte Meldungen wohl in einer Queue zwischenspeichern und von dort in einem separaten Thread in die Datenbank laden. Am selben Ort wäre auch zu prüfen, ob statt Serialisierung Long-Term-Persistence die Kompatibilität erhöhen könnte. Die Kompatibilität zu anderen Datenbanken ist nicht getestet worden, aber vermutlich mit trivialen Anpassungen herstellbar.
  - Systematische Performancetests mit entsprechenden Werkzeugen und Massentests mit vielen gleichzeitigen spielenden Clients wären sicher angebracht.
  - Die Asset-Synchronisierung erfolgt momentan nicht strikt inkrementell (es wird immer das komplette Asset übermittelt).

- **Server**
  - Weitere standardisierte GameStep könnten sinnvoll sein. Beispielsweise ein ConditionalGS, der abhängig von bestimmten Bedingungen aktiv wird bzw. Meldungen an seine Kinder zuweist.
  - Einbindung von Scripting-Möglichkeiten in die GameStep-Parametrisierung könnte interessant sein.
  - Ein GameStep, der Event-Delegation an seine Kinder anbietet (mit einem Subscribe-Verfahren statt mit Delegation), könnte Sinn machen, falls das Spiel keine hierarchische/rundenbasierte Struktur hat, sondern sehr nebenläufig ist.
  - GameSteps werden momentan nicht dynamisch geladen. Ideal wäre eine Upload-Möglichkeit von GameStep-Klassen durch den Admin-

Client; realistischer ein Server-Verzeichnis, in das zusätzliche GameStep-Klassen abgelegt werden können. Eventuell müssten auch die GameStep-Klassen beim Spielstart in der Datenbank gespeichert werden, um eine korrekte Reaktivierung alter Spiele zu erreichen.

- Das Reaktivieren von unterbrochenen Spielen funktioniert nur eingeschränkt mit dem InterruptableGS zusammen (dem einzigen zeitabhängigen GameStep). Evtl. müsste er sich selber, wenn seine Zeit abgelaufen ist, eine Meldung vom Typ 'MT_INTERRUPT' zuschicken.
- *Netzwerkschicht*: Denial-of-Service-Attacken sind durch den zweistufigen Verbindungsaufbau möglich. Zumindest sollte die Möglichkeit geschaffen werden, explizit aus einem Spiel auszuloggen. Auf Stufe ServerSpooler sollte die Aufbewahrung von Verbindungsinformationen zeitlich limitiert werden.
- Meldungs- und Asset-Typen sind schwierig konsistent und konfliktfrei zu halten. Die Verwendung von typsicheren 'enums', die Java aber erst ab Version 1.5 kennt [42], böte sich als Lösung an.

- **Client**
  - Das Verhalten der Asset-Trees im Java-Client ist nicht immer ganz konsistent. Bei Änderungen von Asset-Attributen wird beispielsweise die Anzeige nicht nachgeführt und erst sichtbar, wenn erneut zum selben Knoten im Baum navigiert wird.
  - AssetActions werden clientseitig nicht validiert. Nützlich wäre bei Texteingabe beispielsweise die Unterscheidung von Text- und Zahlen-Eingaben und bei letzteren die Einschränkung auf bestimmte Wertebereiche.
  - Clientseitig könnte eine Bibliothek zur vereinfachten Erstellung von Avataren (computerisierten Mitspielern) nützlich sein. Etwa in der Form von Regeln, welche die automatische Reaktion auf eingehende AssetActions beschreiben.

- Zur Distribution des Clients stehen noch alle Möglichkeiten offen. Sei es als Applet, WebStart-Applikation oder (wie bisher) als doppelklickbares JAR-Archiv.

- **Realoptionen-Spiel**
  - Der ganze Reporting-Bereich ist verbesserungswürdig.
  - Nett wäre natürlich eine voll-graphisch massgeschneiderte Oberfläche mit animierten Ölborpumpen, graphischer Kurs-Verlaufs-Anzeige, etc.

- **Konfiguration**
  - Die Konfiguration mittels XML ist sehr mächtig, aber für Laien hart an der Grenze des Zumutbaren. Vier Lösungsmöglichkeiten fallen dazu ein:
    - GameSteps als JavaBeans graphisch konfigurieren (ein Ausgangspunkt wäre `java.beans.PropertyEditorSupport`)
    - Einen massgeschneiderten Editor dazu schreiben
    - Ein vereinfachtes Format mittels suchen/ersetzen in XML überführen.
    - XSL zur Konvertierung benutzen oder einen XML-Editor mit entsprechender DTD.

# 9. LITERATURVERZEICHNIS

*Alle angegebenen Internet-Referenzen waren am Abfragedatum 20.08.2004 gültig.*


[1] Eibensteiner, S.: Real Optionen, Diskussion der Anwendbarkeit auf IT-Projekte, Seminararbeit an der Wirtschaftsuniversität Wien, 2003, http://wwwai.wu-wien.ac.at/~koch/lehre/inf-sem-ws-02/eibensteiner

[2] SwingML, Homepage, http://swingml.sourceforge.net

[3] SwiXml, Homepage, http://www.swixml.org

[4] Thinlet, Homepage, http://www.thinlet.com

[5] WebOnSwing, Homepage, http://webonswing.sourceforge.net

[6] ULC, Homepage, http://www.canoo.com/ulc

[7] Thin Client Framework, Homepage, http://www.alphaworks.ibm.com/tech/tcf

[8] BSFramework, Homepage, http://www.bs-factory.org

[9] Jini/JavaSpaces, Homepage, http://wwws.sun.com/software/jini

[10] JMS (Java Message Service), Homepage, http://java.sun.com/products/jms

[11] JSDT (Shared Data Toolkit for Java Technology), Homepage, https://jsdt.dev.java.net/

[12] Kouadri Mostéfaoui, G./Kouadri Mostéfaoui, S.: Java Shared Data Toolkit for Multi-Player Networked Games, 2003, http://diuf.unifr.ch/people/kouadri/publications/JSDTpaper.pdf

[13] Struts, Homepage, http://struts.apache.org

[14] Pushlets, Homepage, http://www.pushlets.com

[15] Apple Computer: Dynamic HTML and XML: The XMLHttpRequest Object, 2004, http://developer.apple.com/internet/webcontent/xmlhttpreq.html

[16] Hibernate, Homepage, http://www.hibernate.org

[18] Brynielsson J./Wallenius, K.: GECCO, Game Environment for Command and Control Operations, 2001: http://www.nada.kth.se/theory/gecco/

[19] Mallet, V.: Jags, The Java Game System, 1999: http://www.jfouffa.com/vmallet/Jags.html

[20] Ataraxia, Homepage, http://ataraxia.sourceforge.net

[21] Gameframe, Homepage, http://gameframe.sourceforge.net

[22] Richter, J.: Yale Game Server, 1999, http://zoo.cs.yale.edu/classes/cs490/99-00a/richter.john.jmr39

[23] SAGSAGA, Swiss Austrian German Simulation And Gaming Association, Homepage, http://www.sagsaga.org

[24] Long-Term-Persistence of JavaBeans Components, http://java.sun.com/products/jfc/tsc/articles/persistence

[25] XHTML-Mobile DTD, http://www.wapforum.org/DTD/xhtml-mobile10.dtd

[26] WAP Forum Ltd., XHTML Mobile Profile, 2001, http://www.openmobilealliance.org/tech/affiliates/wap/wap-277-xhtmlmp-20011029-a.pdf

[27] W3C-Validator, http://validator.w3.org

[28] Download für Java 1.4.2, http://java.sun.com/j2se/1.4.2/download.html

[29] Entwicklungsumgebung IDEA, http://www.jetbrains.com/idea/download

[30] Versionskontrolle Subversion, Homepage, http://subversion.tigris.org

[31] Subversion Plug-in für IDEA, Homepage, http://svnup.tigris.org

[32] Distribution der Datenbank MySQL, http://www.serverlogistics.com/mysql.php
[33] phpMyAdmin, Homepage, http://www.phpmyadmin.net
[34] Tomcat, Homepage, http://jakarta.apache.org/tomcat
[35] Apache Commons Logging, Homepage, http://jakarta.apache.org/commons/logging
[36] Log4J, Homepage, http://logging.apache.org/log4j
[37] Apache Commons Collections, Homepage, http://jakarta.apache.org/commons/collections
[38] Tessier, T.: Create a scrollable virtual desktop in Swing, 2001, http://www.javaworld.com/javaworld/jw-11-2001/jw-1130-jscroll.html
[39] TableSorter, Swing-Tutorial, http://java.sun.com/docs/books/tutorial/uiswing/components/example-1dot4/TableSorter.java
[40] Flanagan, D.: FileTransferHandler, in: Java Examples in a Nutshell, 3rd Ed., O'Reilly, 2004, http://www.davidflanagan.com/javaexamples3
[41] MD5-Digest, http://forum.java.sun.com/thread.jsp?forum=9&thread=531112
[42] 'enums' in Java 1.5, http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html
[43] Ant, Homepage, http://ant.apache.org
[44] Metrowerks CodeWarrior, Homepage, http://www.metrowerks.com

# 10. ANHANG A – SPEZIFIKATION

*Interactive Oil Field Game; Game Design von Prof. Marc Chesney und Felix Morger*





**Contents**







**Interactive Oil Field**

**Game**

**Game Design**

Prof. Marc Chesney, Felix Morger

February 11th, 2004

Version 2.1

# 1 Aim of the Document

In a course on real options an interactive game among students on the valuation of oil fields will be played on computers. In order to do so, first, a design of the game is elaborated. Second, the game is converted into code. Finally, tests of the code are run.

This document contains general guidelines with some remarks on the target groups and the design of the course in the beginning and concentrates then on the design of the game.

**In its first version the game will be realized in a simple setting!** Many interesting but complex features (storage of oil, multiple loops, borrowing of oil) will be realized in later versions.

# 2 General Guidelines

## 2.1 Main Goals

- The game allows the students to study the subject of real options within a practical context. It therefore improves students' intuitive understanding of the real option theory.
- Participant's decisions can be analyzed and compared with the theoretically optimal ones. In order to do so, it must be possible to answer the following questions:
  - Do students value the project according to the net present value (NPV) approach or according to real options theory?
  - Do students choose the amount of oil production, as game theory would predict?
  - How does the quality of students' value assessment evolve as the game goes on?

## 2.2 Requirements

The following requirements help achieve the main goals:

- The students understand the game.
- The setting is realistic.
- The results can be presented shortly after the game ends.

## 2.3 Target groups and curricular integration

### 2.3.1 Licentiate students

The licentiate students study business and/or economics and thereby have had some exposure to finance, and in particular to options and their valuation. They are in the 5$^{th}$ to 8$^{th}$ semester.

The game will be part of a 2-day crash course on real options. This course was first held in May 2003 and will be held again in June 2004. The students will receive a "Seminarschein" for a successful participation. Presumably, the game will be integrated in a regular one-semester course on real options later. In this case, the students will earn APS points for a successful participation.





### 2.3.2 Master students

Masters students follow the program of Masters of Advanced Studies in Finance of the ETHZ and the University of Zurich and have good knowledge of finance already. The game would be part of a course on the valuation of (real) options.

### 2.3.3 Executive MBA students

This target group has a more practical background in finance. The game will possibly be integrated into the existing MBA course at the economic faculty of the university of Zurich. Talks have already been undertaken.

### 2.3.4 Executives

Executives also posses a more practical background. The game will be part of a seminar, which lasts between two days or one week. Prof. Marc Chesney has already organized similar seminars on real options in Paris and Lausanne together with Prof. Rajna Gibson.

## 2.4 Resources

### 2.4.1 Human resources

- Subject matter experts: Prof. Marc Chesney (in-house) and Prof. Abraham Bernstein
- Instructional design: Michael Korner (in-house)
- Game design: Felix Morger (in-house)
- Programmer: David Frank. He is a "Wirtschaftsinformatik" student, who writes his diploma thesis on the topic, will fill the position.
- Designer: This position is open. In a first step a designer is not urgently needed.

### 2.4.2 Financial resources

Cost for software will possibly accrue, which can most likely be funded in-house. Certainly, in 2004 the possibility of an application on funding from the University exists.

## 2.5 Course Design

For all target groups the course will start with theory. This theory can then be applied within the interactive game. A game contains 1 loop. Each loop has 4 to 6 periods. The game can easily be paused to discuss some first results. In the third part the results of the game are analyzed and compared with theoretically optimal ones. In a two-day course such a session will be held once. In longer courses it is possible to have more than one session.

# 3 Game Design

## 3.1 Main Setting

- Number of participants is $i$, where $i \geq 2$
- The government has the rights on $f$ identical oil fields, where $f < i$. It wants to lease these rights to private firms for $n$ years. During these $n$ years the firms can produce oil. $v$% of the profits from the sold oil goes to the government.

- The game is played in discrete time intervals. The next period starts, when all players have made their decision, or when the period is over.
- A virtual newspaper can announce exogenous changes in the spot price, $P_t$.
- Each participant has a bank account, $BA^i$, where the transactions within a loop are summarized.
- In order to develop the participants understanding of the game a test round is played before the real game starts.
- If the game is used for research purposes it is important that the participants reveal their true preferences. Incentives as real money, which is paid out at the end of the game, ensure that the participants reveal their true preferences. The payment is a fraction of the earnings during the game.

### 3.2 Steps of the Game

#### 3.2.1 Auction

In the beginning, a Vickery auction is held. A Vickery auction is a second-price sealed-bid auction, i.e., the second highest price determines the selling price for all players. In contrast to a first-sealed bid-auction, it is optimal for the players in Vickrey auctions to bid their true value, independently of what the others do. Vickery auctions are e.g. used to determine the price of bond issues.

The endowment of the participants is $E$.

**Remark:** Those who do not get a right on an oil field, cannot participate. This is a drawback of this setting and can motivate over- or under pricing depending on, if the students like to participate or not.

#### 3.2.2 Initial Investment and Choice of Production Capacity

Those firms who could lease a field decide in each period, if they pay the initial investment $I$ in order to exploit the oil field.

The firms can choose between $j$, $j = 1,2,$ technologies for the production of crude oil. Each technology is characterized by the investment amount $I_j$, drilling costs $K_j$, and a production capacity $Q_j^{max}$. In the case of two technologies, $l$ and $h$, the relations $I_l < I_h$, $K_l < K_h$, and $Q_l^{max} < Q_h^{max}$ hold.

**This feature shall note be implemented in the prototype version of the game.** It can be added, if the prototype works.

#### 3.2.3 Production Periods

All firms who paid the initial investment amount $I$ can produce oil in the remaining periods. These firms can decide in each period, how much oil they want to produce, $Q$. This decision includes the option to abandon ($Q = 0$) and the option to resume ($Q_t > 0 \mid Q_{t-1} = 0$). The costs to abandon $K_j^A \geq 0$ are constant. The costs of resuming $K_j^R > 0$ depend on how long production was ceased before and increase with rate $r^R > 0$. The longer the break is, the higher the costs, because more facilities must be replaced. If the costs cannot be paid, the production cannot be resumed.





It is possible that firms bankrupt, since the price to which they sell the oil is only fixed after all firms have taken their production decision. A firm that bankrupts is excluded from the game.

### 3.3 Spot Price Process of Crude Oil

The price of crude oil $P_t$ is determined through demand and supply. The demand is simulated through a binomial tree. The supply is determined through the total amount of oil sold in the market per period. The price of crude oil behaves according to

$$P_t = P_t^D \cdot e^{k\left(-\frac{Q_t^{tot}}{Q_{0,J}^{max,Jtot}}+0.5\right)},$$

where $$P_t^D = P_0^D \cdot u^w \cdot d^{t-w},$$

and $$u = e^{\sigma\sqrt{\Delta t}}, \quad d = \frac{1}{u}, \quad p(u) = \frac{e^{(r_f-\delta)\sqrt{\Delta t}} - d}{u - d}, \quad p(d) = 1 - p(u).$$

First, the parameters will be declared, subsequently the equations will be interpreted: $P_t^D$ follows a binomial process and $P_t^D$ is its initial value. $k$ is a constant. $u$ and $d$ are constant factors by which $P_t^D$ is multiplied; $u > 1$, $0 < d < 1$ hold. In each period $P_t^D$ is multiplied by $u$ or by $d$. $w$ counts the periods in which $P_t^D$ was in increased by $u$. The $\sigma$ denotes the volatility of the binomial tree. $\Delta t$ is the time interval and is normally 1 year. $p(u)$ is the probability of $P_t^D$ going up. $r_f$ is the interest rate and $\delta$ the convenience yield.

Interpretation of the equations:

- The first component of $P_t$, $P_t^D$, can be interpreted as the price movement induced by the demand side. It follows a binomial tree.
- The second component is mainly determined by a fraction. This fraction is the ratio between the realized industry production and the maximal industry production, if all firms had chosen the technology with the high capacity. The fraction has a codomain of [0,1]. The constant $k$ can be interpreted as a proxy for market size of the oil fields from the government relative to the total market size. The higher $k$ is, the bigger is the impact of the supply on the price. For $k = 2$, $P_t$ can be increased up to 2.7 times and decreased up to the factor 0.4.

### 3.4 Information Given to the Participants

Every firm has full knowledge about its own and its competitor's financial situation in terms of profits.

A detailed list of the parameters and their availability is given in appendix A. The parameters are grouped as 'Server', 'Public', and 'Private'. 'Server' data are data that are only available on the server. 'Public' means that the data are accessible for all players. 'Private' means that the data is only accessible for the player to whom the data belong.

### 3.5 Announcement

- Announcement of demand shocks, e.g. detection of unknown oil fields or wars. The new price is given exogenously in absolute terms.

Influenced parameter: $P_t$.

### 3.6 Summary

#### 3.6.1 Uncertainties

- Price of a barrel oil, $P_t$.
- Amount of oil produced in a period, $Q_i^{tot}$.

#### 3.6.2 Constants

- Production costs $K_i$, after the choice of the technology, and all other costs.
- All rates.
- Production capacity per period $Q_j^{max}$, after the choice of the technology.
- Percentage of earnings that belong to the government.

## 4 Releases

### 4.1 Optional Feature in the First Release

In a first release the following feature could be set aside:

- the choice of production capacity (see 3.2.2 Initial Investment and Choice of Production Capacity),

### 4.2 Ideas for Later Releases

Ideas for later releases are:

- $l$ loops are played. After a loop the gains $G^i$ are written on a firms personal game accounts, where the gains from each loop are summarized. Firms that bankrupt in a loop have a gain of zero for that loop. There is no other punishment. Each loop starts with a new auction. The auction is held after $n > 3$ periods. Two possibilities exist:
  - All participants can bid.
  - The $m$ firms with the smallest profits from the firms who could lease a field in the previous round, are excluded, i.e., the best $f - m$ firms plus the $i - f$ firms, who couldn't lease a field, bid. This second possibility takes into account that the government has an interest in high profits from the produced oil and therefore excludes bad performing firms.
- Storage of oil: Produced oil can be sold on the market or stored to sell it in some later periods. The amount of oil sold in the market is $Q^M$ and the amount of stored oil is $Q^S$. The storing costs per barrel oil and per period are $K^S > 0$. If the storing costs cannot be paid, the stored oil (or a part of it) must be sold.
- Uncertainty over the quantity of oil in a field. It will be revealed after drilling the field.
- The newspaper announces that a new drilling technology will be developed with a chance of $p$. It would enlarge the capacity to $Z^*$ mio barrel, be available in $x$ periods and cost $K^*$ mio. This allows the analysis of the output development after the announcement. After the $x$ periods an announcement declares if the technology could be developed or not.
- Announcement of a new competitor with a certain capacity.





- Introduction of a bank as a player. The bank can decide if it wants to give money to a firm or not.
- Borrowing and lending of money with the bank as counter party.

## A Parameters and their Availability to the Participants

This part of the appendix includes two tables. In the first table all the parameters are listed. The second table includes the parameters that need more information.

| Parameter | Restriction | Availability |
|---|---|---|
| Number of firms, $i$ | $i \geq 2$ | Public |
| Number of oil fields, $f$ | $f < i$ | Public |
| Number of years, resp. time intervals per loop, $n$ | $2<n$ | Public |
| Actual time period, $t$ | $0<t$ | Public |
| Real time per period, $RT$ | $0<RT$ | Public |
| Remaining real time per period, $Rt$ | $0<Rt \leq RT$ | Public |
| Tax on profits, $v\%$ | $0 \leq v \leq 1$ | Public |
| Bank account, $BA^i$ | $0 \leq BA^i$ | Public |
| Initial endowment, $E$ | $0<E$ | Public |
| Number of production technologies, $j$ | $1 \leq j \leq 2$ | Public |
| Initial investment, $I_j$ | $0 \leq I_j$ | Public |
| Drilling costs per unit oil, $K_j$ | $0 \leq K_j$ | Public |
| Production capacity, $Q_j^{max}$ | $0 \leq Q_j^{max}$ | Public |
| Maximal production capacity (high tech.), $Q_h^{max\ tot}$ | $i * Q_h^{max} = Q_h^{max\ tot}$ | Public |
| Costs to abandon, $K_j^A$ | $0 \leq K_j^A$ | Public |
| Costs to resume, $K_j^R$ | $0 \leq K_j^R$ | Public |
| Rate of annual increase in the costs to resume, $r^R$ | $0 \leq r^R \leq 1$ | Public |
| Price of crude oil, $P_t$ | $0 \leq P_t$ | Public |
| Demand driven price, $P_t^D$ | $0 \leq P_t^D$ | Server |
| Pricing constant, $k$ | $0 \leq k$ | Server |
| Multiplication factors, $u$, $d$ | $0<d<1<u$ | Public |
| Counter of occurrences of $u$, $w$ | $0 \leq w \leq t$ | Server |
| Volatility of the binomial tree, $\sigma$ | $0 \leq \sigma \leq 1$ | Public |
| Period length in years, $\Delta t$ | $0<\Delta t$ | Server |
| Probability of an up resp. down move, $p(u)$, $p(d)$ | $0<p(u)<p(d)<1$ | Server |
| Risk-free interest rate, $r_f$ | $0 \leq r_f \leq 1$ | Public |



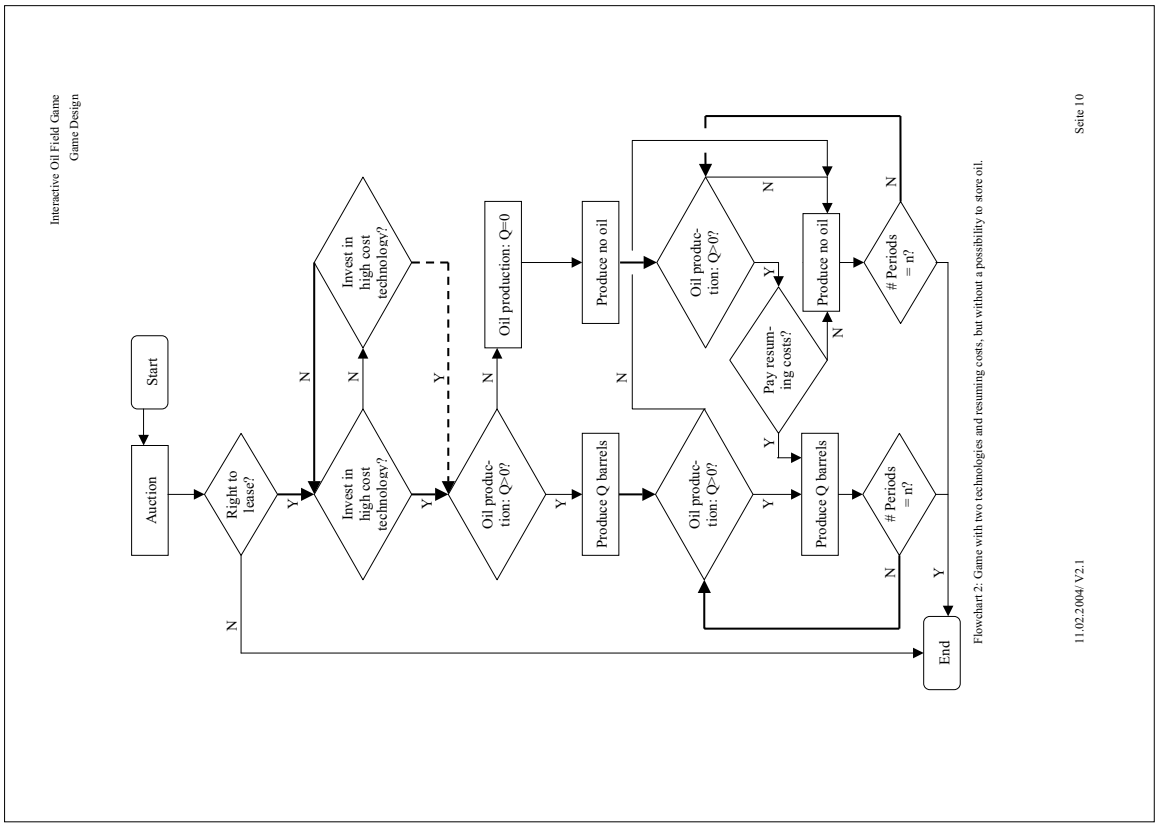


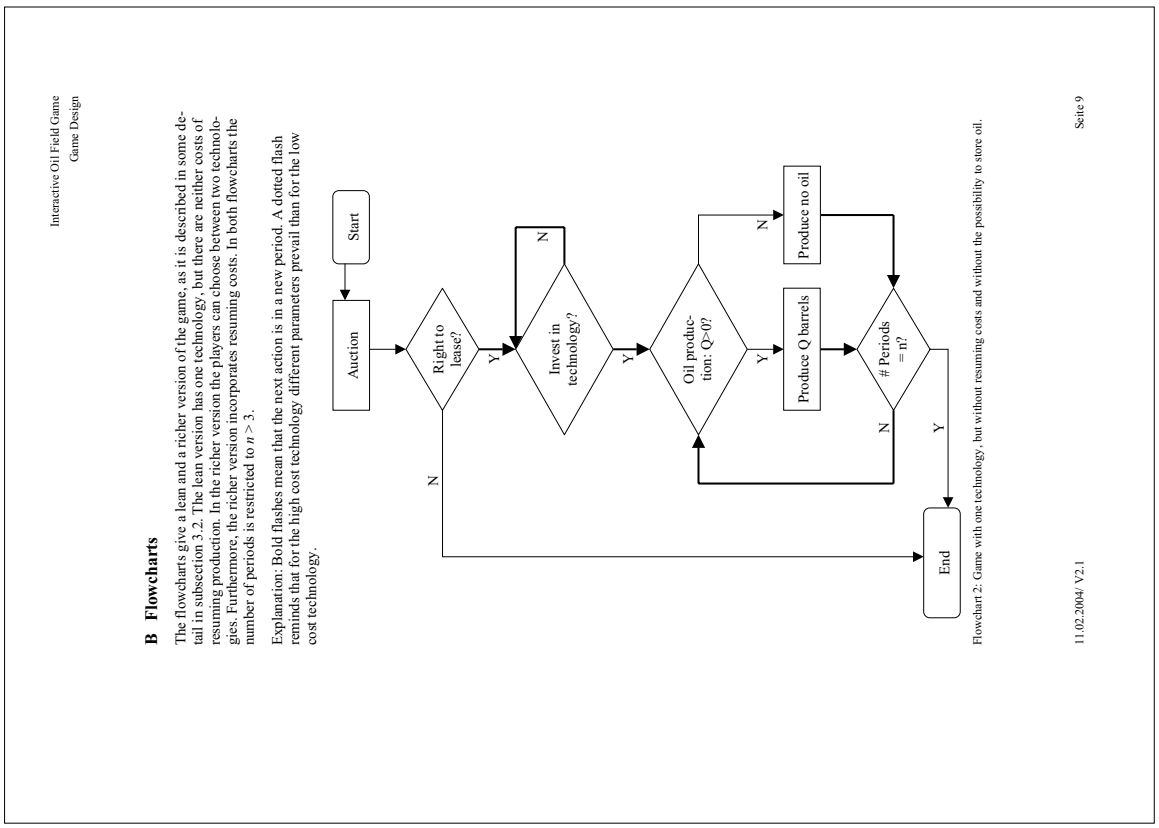
| | | |
|---|---|---|
| Convenience yield, $\delta$ | $0 \leq \delta \leq 1$ | Public |
| Produced oil in a period, $Q_t^i$ | $0 \leq Q_t \leq Q_j^{max}$ | Private* |
| Costs incurred by a firm at time $t$: $K^t$, $K_t^{R,i}$, $K_t^{A,i}$ | $0 \leq K^t$, $K_t^{R,i}$, $K_t^{A,i}$ | Private |
| Profits, $G_t^i$ | $0 \leq G_t^i$ | Public* |

| Parameter | Restriction | Availability |
|---|---|---|
| Market production of oil in the previous period, $Q_{t-1}^{tot}$ | $0 \leq Q_{t-1}^{tot} \leq Q_j^{tot}$ | Public |
| Each firms oil production in the previous period, $Q_{t-1}^i$ | $0 \leq Q_{t-1}^i \leq Q_j^{max}$ | Public |
| Each firms profits in the previous period, $G_{t-1}^i$ | $0 \leq G_{t-1}^i$ | Public |
| Each firms total profits, $G^{tot}{}_{t-1}^i$ | $0 \leq G^{tot}{}_{t-1}^i$ | Public |

Table 2: Availability of some parameters in detail.

Spezifikation (S. 11-12): *Interactive Oil Field Game; Game Design*



# 11. ANHANG B – KONFIGURATION REALOPTIONEN-SPIEL

*Legende*: **GameSteps**
**Spiel-Parameter**
**Spiel-Texte**
Kommentare

```
<?xml version="1.0" encoding="UTF-8" ?>
<java version="1.4.0" class="java.beans.XMLDecoder">

 <object class="dfrank.game.server.gamesteps.GSSequence">

  <!-- global: newUser-Step -->
  <void method="addGameStep">
   <object class="dfrank.game.server.gamesteps.InterruptableGS">

    <void property="StepName">
     <string>End user login and start auction</string>
    </void>

    <!-- global: initialize -->
    <void method="addGameStep">
     <object class="dfrank.game.rog.server.rog_gs.InitializationGS">
      <void property="Endowment">
       <double>1000</double> <!-- E - Endowment: initial amount of money every user becomes -->
      </void>
      <void property="NumberOfFirms">
       <int>2</int> <!-- i - Number of firms: >= 2 -->
      </void>
      <void method="setAdminUP">
       <string>admin</string>
       <string>secret</string>
      </void>
     </object>
    </void>

   </object>
  </void>

  <!-- global: Main Game-Step -->
  <void method="addGameStep">
   <object class="dfrank.game.server.gamesteps.GSIterationSequence">
    <void property="NumberOfIterations">
     <int>1</int> <!-- play game 1 time -->
    </void>
    <void property="BeginIterationsText">
     <string>Start Game</string> <!-- optional: say that game started -->
    </void>
    <void property="NextIterationText">
     <string>Next Game</string> <!-- optional: next game -->
    </void>
    <void property="EndIterationsText">
     <string>End Game</string> <!-- optional: say that game is over -->
    </void>

    <!-- N: Loops - Begin -->
    <void method="addGameStep">
     <object class="dfrank.game.server.gamesteps.GSIterationSequence">
      <void property="NumberOfIterations">
       <int>1</int> <!-- N - Number of Loops >=1 -->
      </void>
      <void property="BeginIterationsText">
       <string>Start Loop 1</string> <!-- optional: say that loop started -->
      </void>
      <void property="NextIterationText">
       <string>Start Loop %iteration%</string> <!-- optional: next loop -->
      </void>
      <void property="EndIterationsText">
```

```
 <string>End Loop</string> <!-- optional: say that loop is over -->
</void>

<!-- what to do in every loop.. -->

<!-- In every Loop: 1.: hold auction -->
<!-- allow auction to be interrupted.. -->
<void method="addGameStep">
 <object class="dfrank.game.server.gamesteps.InterruptableGS">

  <void property="Timeout">
   <long>55</long> <!-- interrupt in Seconds -->
  </void>

  <void property="TimeoutMessage">
   <string>Auction ends in %seconds% seconds.</string>
  </void>

  <void property="StepName">
   <string>Auction</string>
  </void>

  <void method="addGameStep">
   <object class="dfrank.game.rog.server.rog_gs.AuctionGS">
    <void property="NumberOfOilFields">
     <int>1</int> <!-- f - NumberOfOilfields < i-->
    </void>
   </object>
  </void>

 </object>
</void>

<!-- In every Loop: 2.: make n Periods -->
<void method="addGameStep">
 <object class="dfrank.game.server.gamesteps.GSIterationSequence">
  <void property="NumberOfIterations">
   <int>3</int> <!-- n - Number of periods = lease duration for oil field > 2 -->
  </void>
  <void property="BeginIterationsText">
   <string>Start Period 1</string> <!-- optional: say that period started -->
  </void>
  <void property="NextIterationText">
   <string>Start Period %iteration%</string> <!-- optional: next period -->
  </void>
  <void property="EndIterationsText">
   <string>End Period</string> <!-- optional: say that period is over -->
  </void>
  <void property="GeneralInfo">
   <string>Number of years/time periods:</string> <!-- makes NumberOfIterations public -->
  </void>

  <!-- what to make in every period.. -->

  <!-- In every Period: 1. invest in technology or produce -->
  <!-- allow this phase to be interrupted.. -->
  <void method="addGameStep">
   <object class="dfrank.game.server.gamesteps.InterruptableGS">

    <void property="Timeout">
     <long>360</long> <!-- interrupt in Seconds -->
    </void>

    <void property="TimeoutMessage">
     <string>Period ends in %seconds% seconds.</string>
    </void>

    <void property="StepName">
     <string>Period</string>
    </void>

    <void method="addGameStep">
     <object class="dfrank.game.server.gamesteps.GSSwitch">
```

```
    <!-- Either...: choose technology -->
    <void method="addGameStep">
     <object class="dfrank.game.rog.server.rog_gs.InvestmentGS">
      <void method="addTechnology">
       <string>Drilling Technology 1</string> <!-- Description for Technology -->
       <double>100</double> <!-- I_invest - Price for Technology -->
       <double>15</double> <!-- K_drill - Cost of Drilling per unit -->
       <long>20</long> <!-- Q_capacity - Capacity after drilling -->
       <long>1</long> <!-- min_Prod - Minimal production amount -->
       <double>10</double> <!-- KA_cost_abandon - Cost to abandon -->
       <double>8</double> <!-- KR_cost_resume - Cost to resume -->
       <double>0.2</double> <!-- resume_increase_rate - Rate of annual increase in the cost to resume -->
      </void>
      <void method="addTechnology">
       <string>Drilling Technology 2</string> <!-- Description for Technology -->
       <double>40</double> <!-- I_invest - Price for Technology -->
       <double>50</double> <!-- K_drill - Cost of Drilling per unit -->
       <long>10</long> <!-- Q_capacity - Capacity after drilling -->
       <long>1</long> <!-- min_Prod - Minimal production amount -->
       <double>10</double> <!-- KA_cost_abandon - Cost to abandon -->
       <double>8</double> <!-- KR_cost_resume - Cost to resume -->
       <double>0.05</double> <!-- resume_increase_rate - Rate of annual increase in the cost to resume -->
      </void>
     </object>
    </void>

    <!-- ...or: produce oil -->
    <void method="addGameStep">
     <object class="dfrank.game.rog.server.rog_gs.ProduceGS">
     </object>
    </void>

   </object>
  </void>

 </object>
</void>
<!-- end of interrupt-period -->

<!-- In every Period: 2. look who earns what -->
<void method="addGameStep">
 <object class="dfrank.game.rog.server.rog_gs.AccountingGS">

  <void property="TaxOnProfits">
   <double>0.34</double> <!-- v - Tax on Profits 0..1 -->
  </void>

  <void property="ConvenienceYield">
   <double>0.01</double> <!-- d - Convenience Yield 0..1 -->
  </void>

  <void property="RiskFreeInterestRate">
   <double>0.03</double> <!-- rF - Risk-free Interest Rate 0..1 -->
  </void>

  <void property="Volatility">
   <double>0.2</double> <!-- s - Volatility of the binomial Tree 0..1 -->
  </void>

  <void property="PricingConstant">
   <double>1.5</double> <!-- k - Pricing constant k >= 0 -->
  </void>

  <void property="PeriodLength">
   <double>1.0</double> <!-- dt - how many years is one period? -->
  </void>

  <void property="InitialDemandDrivenPrice">
   <double>15.0</double> <!-- PD(0) - initial demand driven price -->
  </void>

 </object>
```

```
      </void>

     </object>
    </void>
    <!-- end of period-iterator -->

   </object>
  </void>
  <!-- end of loop-iterator -->

 </object>
</void>

</object>

</java>
```

# 12. ANHANG C – INHALT DER CD-ROM

Auf der beiliegenden CD-ROM befindet sich:

- Der vollständige Quellcode von
  - Spiel-Framework
  - Administrations-Spiel
  - Realoptionen-Spiel
  - Web-Client (separates Projekt)
- Vorkompilierte, doppelklickbare Client- und Server- Applikationen sowie eine vorkompilierte Web-Applikation (WAR).
- Projektdateien für IntelliJ IDEA 4.5.1 und generierte Ant-Build-Dateien.
- Datenbank-Schemata von Asset-Datenbank und GameManager-Datenbank.
- Alle benutzten Bibliotheken im Quellcode und (wo erhältlich) als kompilierte JAR-Dateien.
- Diese Diplomarbeit im PDF-Format.