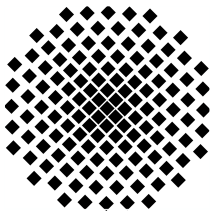


Universität Stuttgart
Fakultät Informatik

Prüfer:.....
Betreuer:..... Thomas Eisenbarth
Beginn am:..... 15.3.2000
Beendet am: 15.9.2000
CR-Klassifikation: .....
D.3.2 Language Classifications
Specialized application languages?
Design languages?
D.3.3 Language Constructs and Features:
Interpreters

Studienarbeit-Nr.

1 Inhaltsverzeichnis

1 Inhaltsverzeichnis 2
2 Projektplan 4
3 Aufgabenstellung 5
3.1 Was sind Rollenspiele? 5
3.1.1 Wie funktionieren Rollenspiele 5
3.1.2 Ablauf eines Rollenspieles 6
3.2 Aufgabe der Diplomarbeit 7
3.3 Anforderungen an die Programmiersprache 7
4 Spezifikation 8
4.1 Design der Sprache 8
4.2 Geschichtsführung 10
4.3 Basistypen 12
4.3.1 Würfelwert 13
4.3.2 Gültige Operationen der Basistypen 14
4.4 Tabellen 15
4.4.1 Definition 15
4.4.2 Beispiel 16
4.5 Aufbau des Charakters 16
4.6 Formeln 17
4.7 Attribute 18
4.8 Statements 19
4.8.1 Attributdeklaration 19
4.8.2 Anweisungen 20
4.8.3 Anweisung "parentall" 20
4.8.4 Anweisungen mit Prioritäten 22
4.8.5 Bedingungen 23
4.8.6 Zusagen (Assertions) 23
4.8.7 Vorbedingungen 24
4.8.8 Events 24
4.9 Typen 25
4.10 Objekte 26
4.11 Spezialfunktionen 27
4.11.1 count/countall 27
4.11.2 sum/sumall 27
4.11.3 countparents 28

4.11.4 istype ... 28
4.12 Charakterbaumerstellung ... 28
4.13 Modifizieren des Charakters ... 31
4.14 Objektbibliothek ... 32
5 Implementierung ... 33
5.1 Allgemein ... 33
5.2 Einlesen ... 33
5.2.1 Statische Analyse beim Einlesen ... 35
5.3 Statements ... 35
5.4 Algorithmus zur Berechnung der Attribute ... 36
5.4.1 Handhabung von Events (ON CHANGE) ... 40
5.4.2 Handhabung von Verzweigungen (IF) ... 40
5.4.3 Das Einfügen von Objekten ... 42
5.5 Implementierung der Berechnungen ... 43
5.5.1 CalcHandler ... 44
5.5.2 Berechnungen der Attributen ... 45
5.6 Einfügen von Objekten ... 46
5.7 Performance ... 48
5.8 Optimierungen ... 50
5.8.1 Verzögerung der Berechnungen ... 50
5.8.2 Caching der Eingabedaten ... 52
5.8.3 Text Ausgabe ... 53
5.8.4 HTML Ausgabe ... 54
5.8.5 HTML Ausgabe (gurps.jsp) ... 55
5.8.6 HTML Ausgabe (Beispiel) ... 59
6 Glossar ... 60
7 Literatur ... 62
8 Anhang ... 63
8.1 Beispiel RPDL Charakter ... 63
8.1.1 McRathgar.rpdl ... 63
8.1.2 GURPS_base.rpdl ... 64
8.1.3 GURPS_tables.rpdl ... 68
8.1.4 GURPS_baseChar.rpdl ... 70
8.1.5 GURPS_equip.rpdl ... 70
8.1.6 GURPS_equip_skills.rpdl ... 71
8.1.7 GURPS_equip_weapons.rpdl ... 71
8.1.8 GURPS_magic.rpdl ... 72

2 Projektplan

Projektplan RPDL

	# Tage	Von	bis	Tätigkeit
	1	15.3	15.3	Projektplan
				Teil 1: Spezifikation
	1,0	16.3	16.3	Festlegung der Art der Sprache: Wo sind Schwerpunkte zu setzten, was ist nicht so wichtig (Geschwindigkeit, kompatibilität, ...)
	1,0	17.3	17.3	Festlegung der Basistypen
	2,0	20.3	21.3	Spezifikation von Formeln & Tabellen; u.a. wie können welche Basistypen miteinander verknüpft werden?
	8,0	24.3	31.3	Beschreibung der Klassen-Hierarchie
	15,0	3.4	21.4	Festlegung der Syntax von RPDL Kommandos, wie ADD ITEM, MODIFY ITEM
	3,0	24.4	26.4	Erstellen von Tests für RPDL: Was muß getestet werden?
	2,0	27.4	28.4	Spezifikation der Testumgebung: Was soll später ausgegeben werden?
	6,0	1.5	8.5	Dokumentation der Spezifikation (Grammatikalische Überarbeitung)
39				**Teil 2: Programmieren**
	2,0	22.5	24.5	Festlegung der Arbeitsweise des Interpreters. Wie wird er aufgerufen, wie sieht die Konfiguration (config-files) aus?
	5,0	25.5	31.5	Struktur des Interpreters: Festlegung der interfaces & abstrakten Java-Klassen
	22,0	1.6	30.6	Programmierung
	5,0	3.7	7.7	Programmieren von GURPS in RPDL
	2,0	10.7	11.7	Erstellen der Tests
	10,0	12.7	25.7	Testen und Fehlerbeheben
46,0				**Teil 3: Ausarbeitung**
	20,0	7.8	31.8	Schreiben der Diplomarbeit.
		15.9		**Abgabe**

Gesamt: **105,0**

3 Aufgabenstellung

Dieses Kapitel enthält die Aufgabenstellung, und die eigentliche Motivation für die Idee von RPDL.

3.1 Was sind Rollenspiele?

Rollenspiele sind in den 70er Jahren in den USA entstanden. Damals hatte man die Idee sich selbst in eine fiktive Welt zu versetzen; in eine Welt, die eigene Regeln hat, und zu einer anderen Zeit spielt. Die meisten Rollenspiele fallen in die Kategorien Fantasywelt (Umgebung des Mittelalters mit Monstern und Magie) oder Science-Fiction. Es gibt aber auch Rollenspiele, die ein Endzeit-Szenario haben, oder sogar in der aktuellen Zeit spielen. Zum Beispiel gibt es Vampires, das in der heutigen Zeit spielt, und von Vampiren handelt.

Das erste Rollenspiel, Dungeon&Dragons [3], war ein Fantasy-Rollenspiel, das im Mittelalter spielt, und viele Monster hat, wie Drachen, Trolle,...

3.1.1 Wie funktionieren Rollenspiele

Bei einem Rollenspiel gibt es normalerweise einen Spielleiter, der praktisch der Gott dieser fiktiven Welt ist. Des weiteren gibt es die Rollenspieler-Gruppe, meist 2-6 Personen, die nun in dieser Welt agieren. Die Welt selber hat meist einen komplexeren Hintergrund, und ist in den Regelwerken beschrieben. Ein Buch, das so eine Welt gut beschreibt ist „Der Herr der Ringe“ von J.R.R. Tolkien. Die Geschichte handelt in einer mittelalterlichen Welt. Häufig werden die dort vorkommenden Fantasiegestalten in die Rollenspielwelt übernehmen, wie Elfen und Trolle. In Tolkiens Welt gibt es den Kontinent Mittelerde, bei dem im Nordenosten die Elfen wohnen, im Norden die Zwerge in den Bergen, und im Süden das Böse (Trolle, Orks,...). In dem Rollenspiel Midgard gibt es den zum Beispiel den Kontinenten Midgard.

Nachdem der Spielleiter, die Spieler über die Verhältnisse in der Welt aufgeklärt hat (oder sich die Spieler die Passagen in dem Regelwerk durchgelesen haben), und sie einen groben Überblick haben, erzeugt sich jeder Spieler eine fiktive Person, den Charakter, mit dem er in Zukunft in dieser Welt agiert. Meist bestimmt man selber, in welcher Richtung dieser Charakter seine Stärken hat. Beliebte Figuren sind zum Beispiel Zauberer, Kämpfer und Diebe. Man kann aber auch ein Allround-Talent spielen, der zwar viel kann, aber nichts wirklich gut. Das Erstellen des Charakters erfolgt nun nach den Regeln des Rollenspiels, und ist stellenweise sehr komplex. So gibt es bei manchen Rollenspielen nur wenige Grundwerte (Stärke, Intelligenz, Konstitution & Geschicklichkeit), bei anderen gibt es dann zusätzlich noch Ausdauer, Geschwindigkeit, Geruchsinn, Hörsinn, Sicht,...

Nachdem man nun ungefähr weiß, was der Charakter können soll, bestimmt man die Grundwerte. Das kann zum einen durch Würfeln passieren (Midgard), oder man verteilt die anfänglichen Charakterpunkte (bei GURPS® [1] kann man 100CPs auf die Attribute & Fähigkeiten verteilen). Bei Midgard würfelt man zweimal mit einem 100er-Würfel (1W100), und nimmt das höhere Ergebnis. Das Ganze macht man für jeden Grundwert. Manche Spielleiter erlauben es, danach die gewürfelten Werte auf die Grundwerte

selber zu verteilen. Damit kann man seiner gewünschten Spielerrolle besser gerecht werden.

Der nächste Schritt ist die Auswahl der Fähigkeiten (Skills) die der Charakter hat. Einen Charakter spielt man normalerweise nicht als Baby, sondern als eine Person mit einer Ausbildung. Nun gilt es, diese festzulegen. Wer einen Dieb spielen will, wird den Grundwert für Geschicklichkeit möglichst hoch wählen, während ein Kämpfer eher stark und ausdauernd sein sollte. Ein Magier wird mehr Intelligenz brauchen, und ist dafür nicht sehr kräftig. Die einzelnen Fertigkeiten die es gibt, stehen wiederum im Regelwerk, und ihre Kosten ebenfalls. Die Fertigkeiten können im Lauf des Spieles gesteigert werden, wenn der Charakter an Erfahrung sammelt. Je besser ein Charakter eine Fertigkeit kann, desto leichter fällt es ihm, sie auszuführen. So wird ein Dieb sehr gut Schlösser knacken können, während ein Kämpfer daran verzweifeln kann. Die Auswahl der Fertigkeiten will gut überlegt sein, denn der Charakter soll ja überlebensfähig sein, und nicht bei der ersten Aktion versagen.

Als letztes kommt noch die Grundausrüstung für den Charakter. Der Kämpfer wird schon ein paar Waffen und eine Rüstung haben, der Dieb evtl. ein paar Dietriche, usw. Außerdem besitzt Ein Charakter meist auch Kleidung oder eine Rüstung. Er muß ja nicht unbedingt nackt herumlaufen.

3.1.2 Ablauf eines Rollenspieles

Nachdem die Spieler jeweils einen Charakter haben, kann es losgehen. Der Spielleiter muß nun erst einmal die Gruppe zusammenführen. Zum Beispiel kann ein Kämpfer gerade einen Wettbewerb gewonnen haben, und während der Feier trifft er auf einen Magier, der zufällig in der Stadt ist. Sie entschließen sich zusammen in die nächste Stadt zu ziehen, ... Was genau passiert, bleibt dem Spielleiter überlassen. Der Spielleiter könnte nun zum Kämpfer sagen, daß ein Plakat an einer Wand hängt bei dem Leute für einen Auftrag gesucht werden, und sich die Gruppe dann spontan entschließt, mal genauer nachzuforschen, und zu der entsprechenden Adresse zu gehen, und den Auftrag anzunehmen, usw. Was geschieht nun, wenn die Gruppe vor einem Problem steht? Zum Beispiel steht die Gruppe (Dieb, Kämpfer, Magier) vor einer verschlossenen Tür. Der Dieb würde wohl erst Fallen suchen, und danach versuchen das Schloß zu knacken. Ein Kämpfer kann seine Waffe nehmen, und aus der Tür Kleinholz machen; der Magier könnte einen Feuer-Spruch zaubern, und die Tür verbrennen. Wer genau was macht, machen die Spieler unter sich aus. Sind sie sich einig geworden, sagen sie dem Spielleiter, was sie genau machen. Zum Beispiel sagt der Spieler des Diebes zum Spielleiter, daß er als erstes nach Fallen sucht. Als nächstes muß der Spieler würfeln, ob er es schafft oder nicht. Abhängig davon wie gut er Fallen finden kann, wird er sie entdecken, oder nicht. Normalerweise würfelt der Spielleiter verdeckt, und sagt es dann dem Spieler das Ergebnis. Hat er schlecht gewürfelt, wird der Dieb keine Falle finden, auch wenn eine vorhanden ist.

Kommt es zu einem Kampf, zum Beispiel, weil ein paar Diebe das Lager überfallen, so kommen die Kampf-Regeln ins Spiel. Diese sind wiederum stellenweise sehr komplex, so daß es schon einmal 10 min dauern kann, bis eine Runde ausgeführt ist, und alle beteiligten Charaktere ihre einzelnen Aktionen gemacht haben. Üblicherweise gibt es während des Kampfes Runden, in denen der Charakter Aktionen machen kann: Zuschlagen, sich bewegen, oder auch nur die Waffe bereit machen. (Einen neuen Pfeil für den Bogen ziehen, mit der Zweihand-Axt ausholen, oder mit der Armbrust genauer zielen, damit man in der nächsten Runde besser trifft). Manche Spielleiter vereinfachen die Regeln etwas, damit das ganze nicht zu „bürokratisch“, sondern flüssiger abläuft.

Ansonsten schaut man nur noch im Regelwerk nach, was man genau gemacht werden muß, wie zum Beispiel wieweit man mit dem Bogen schießen kann, oder um wieviel die Zielgenauigkeit steigt, wenn man länger zielt.

Hat nun eine Gruppe eine Aufgabe erledigt hat (wie zum Beispiel „klaut das heilige Buch aus dem Tempel und bringt es dem Herrscher“), gibt der Spielleiter den Charakteren meistens die Möglichkeit ihre Fertigkeiten zu verbessern. Bei GURPS® gibt es neue Charakterpunkte, mit denen man sich dann neue Fertigkeiten antrainieren oder bestehende verbessern kann (man braucht aber auch noch Geld & Zeit, um das ganze zu lernen!). Außerdem kann man sich von dem verdienten Gold, das man unterwegs findet (ebenfalls vom Spielleiter abhängig), neue Ausrüstung kaufen.

3.2 Aufgabe der Diplomarbeit

Die Erstellung des Charakters ist immer eine langwierige Aufgabe, und kann Stunden dauern. Insbesondere die Anwendung der vielen Regeln und das Nachschlagen in den Tabellen ist sehr zeitraubend. Oft kommen auch Formeln vor, die man ausrechnen muß (z.B. Geschwindigkeit = Geschicklichkeit + Stärke / 4). Des weiteren hängen die Fähigkeiten von den Grundwerten ab, und man muß in Tabellen nachschauen, welchen Wert man letztendlich hat. Bei den meisten Rollenspielen ist diese Prozedur ähnlich, und in dieser Diplomarbeit soll eine allgemeine Programmiersprache entwickelt werden, die das Schreiben eines Charaktereditors vereinfacht, und bei der man diese Regeln einfach eingeben kann. Des weiteren sollen noch folgende Möglichkeiten bestehen:

- ?? Laden von Bibliotheken, aus denen man die Objekte (Ausrüstungsgegenstände, Fertigkeiten, Zaubersprüche, etc.) wählen kann, die man dem neuen Charakter zuweist.
- ?? Festhalten der Entwicklung des Charakters. D.h. nicht nur den aktuellen Stand, sondern auch wie der Charakter sich entwickelt hat. Damit hat man die Möglichkeit, den Lebenslauf des Charakters nachzuvollziehen.
- ?? Eingabe von Kommentaren wenn man den Charakter verbessert. Dies ist für eine Zusammenfassung des Geschehens, die der Charakter während des letzten Spiels erlebt hat, gedacht.

3.3 Anforderungen an die Programmiersprache

Aufgrund der Analyse von einigen Rollenspiel-Regelwerken, sind die Anforderung an eine Programmiersprache folgende:

- ?? Die Eingabe von Formeln in der Berechnung der Werte. Außerdem muß die Möglichkeit bestehen, andere Werte wie Variablen zu benutzen.
- ?? Die Möglichkeit, der Benutzung von Tabellen in Formeln. Die Tabellen sollen eingegeben werden, und dann in Formeln benutzt werden können.
- ?? Hinzufügen von Objekten zu dem Basis-Charakter. Der Charakter soll neue Objekte bekommen; diese Objekte können Fertigkeiten sein, oder aber auch "richtige" Gegenstände wie ein Schwert oder ein Seil.
- ?? Bildung von Klassen von Objekten. Zum Beispiel ist der Spruch Feuerball ein Zauberspruch, bzw. ein Zauberspruch der Kategorie Feuer.

- ?? Eingabe von Bedingungen, für ein Objekt. Zum Beispiel soll der Spruch Feuerball nur gelernt werden können, wenn man den Spruch „Feuer erschaffen“ beherrscht.
- ?? Rolemaster hat die Regel, daß pro Stufe, den der Charakter erreicht, eine Fähigkeit nur um 2 Stufen verbessert werden kann, und nicht mehr. Diese Eigenschaft soll ebenfalls möglich sein.
- ?? Bei vielen Rollenspielen gibt es einen Unterschied, ob der Charakter neu erschaffen wird, oder sich während des Spieles verbessert. Bei GURPS® gibt es die Vor- und Nachteile eines Charakters die nur bei der Erschaffung des Charakters verändert werden können. Während des Spiels ist dies nicht mehr möglich.
- ?? Markieren von Attributen, die zwingend gesetzt werden müssen. Wie zum Beispiel die Stärke eines Charakters. Es ist dann ein Fehler, wenn diesem Attribut beim Anlegen dieses Objektes kein Wert zugewiesen wird.
- ?? Evtl. Möglichkeit eines „Wizards“, der schrittweise die benötigten Werte abfragt.

4 Spezifikation

4.1 Design der Sprache

RPDL ist eine Sprache, die einfache Regeln (definiert durch Rollenspielregelwerke) umsetzt, und diese dann bei der Charaktererstellung anwendet. Es soll keine zeitkritische Anwendung werden. Das wichtige ist, daß alles richtig berechnet wird, und alle Formeln richtig angewendet werden. Des weiteren ist es wünschenswert, wenn das Programm unter verschiedenen Betriebssystemen, wie Windows und diverse Unix-Varianten (Linux, Solaris, HPUX, ...) läuft. Dementsprechend wird die Implementierung auch in Java [8] stattfinden, auf welche diese Anforderungen am besten zutreffen.

Die Sprache selber wird eine Batchsprache werden: Es werden Kommandos der Reihe nach ausgeführt, und so der eigentliche Charakter nach und nach erzeugt, bzw. vervollständigt. Nach jedem Kommando befindet sich der Charakter in einem festen Zustand.

Beispiel:

- ?? [...]
- ?? Gib Charakter einen Rucksack.
- ?? Lege das Seil in den Rucksack.
  -> die Liste der Gegenstände des Rucksacks beinhaltet nun das Seil

?? [...]

Der Batch-Modus hat auch den Vorteil, daß problemlos neue Kommandos hinzugefügt werden können. Eine nachträgliche Bearbeitung von vorherigen Kommandos ist bei Rollenspielen nicht vorgesehen. In der Praxis kommt es allerdings öfters vor, daß der Charakter nachträglich "angepaßt" wird. So zum Beispiel wenn man später eine Fertigkeit in dem Regelwerk entdeckt, die man am Anfang übersehen hat, und die zu dem Charakter eigentlich besser passen würde.

Um beiden Anforderungen gerecht zu werden, muß die *komplette* Erstellung des Charakters gespeichert werden. Im Gegensatz zu anderen Charaktereditoren, die nur den aktuellen Stand abspeichern. Ein Beispiel: Bei dem letzten Update hat der Charakter ein Schwert bekommen, daß jetzt unbrauchbar ist, und deshalb ist es wieder entfernt worden. Der Charakter hat also im Moment kein Schwert. In RPDL sollen beiden Ereignisse gespeichert, und auch ausgewertet werden.

Die Darstellung des Charakters erfolgt über eine Baum-Struktur. Wie detailliert der Baum letztendlich modelliert wird, hängt vom Regelwerk des jeweiligen Rollenspieles ab, und wie umfangreich man selber die Gegenstände erstellt. Zum Beispiel kann man einen Ruchsack in ein linkes und rechtes Fach aufteilen, oder einfach als einen großen Behälter ansehen:

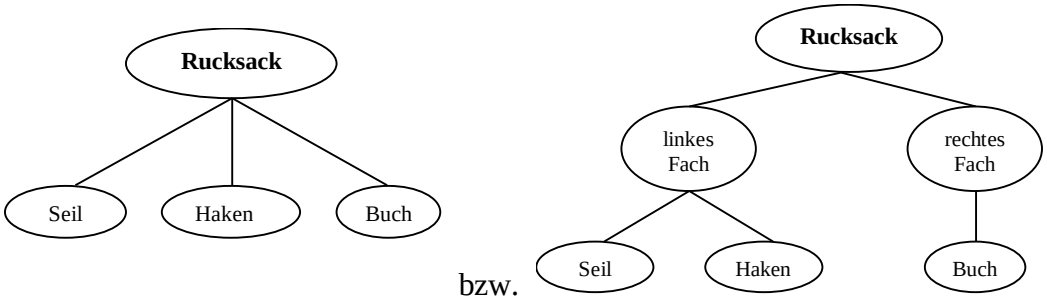


Abbildung 4.1: Baumstruktur des Charakters

Manche Rollenspiele teilen den Charakter in verschiedene Körperteile (Arme, Beine, Kopf) auf, um den Kampf, bzw. die Treffer realistischer zu machen. Bei anderen Systemen gibt es nur den gesamten Körper des Charakters. Das Vereinfacht den Spielablauf während eines Kampfes drastisch.

Eine weitere wichtige Eigenschaft ist die Verwendung von fertigen Objekten. So soll es möglich sein, daß es schon eine Bibliothek mit fertigen Gegenständen gibt, die man nur noch dem Charakter geben muß. Dies ist sinnvoll für die Standard-Ausrüstung, wie Kleidung, Rüstung, Waffen, usw. Aber auch Dinge wie Fähigkeiten und Zaubersprüche sollen vorher eingegeben werden können, und von verschieden Charakteren benutzt werden.

Die Programmiersprache ist damit in 3 Teile aufzuteilen:

?? Beschreibung der Rollenspielregeln.

?? Erstellen von Objektbibliotheken.

?? Die eigentlichen (individuellen) Charakterdaten.

Die Regeln werden für das jeweilige Rollenspielsystem geladen. Die Rollenspiel Regeln sind für jeden Charakter innerhalb dieses Rollenspielsystems die gleichen. Die Regeln müssen nicht unbedingt in einem File stehen, sondern können auch auf mehrere Files verteilt werden. So gibt es auch bei den einzelnen Rollenspielen nicht grundsätzlich ein

Regelwerk, sondern auch Regelerweiterungen. Üblicherweise beschreiben diese spezielle Teilbereiche des Rollenspiels detaillierter. Wie zum Beispiel das erweiterte Kampfsystem von GURPS®, daß Treffer nicht nur allgemein auf den Körper zuläßt, sondern Unterschiede zwischen Kopf, Arm, Bein, etc. macht. Dies gilt auch für den Rüstungsschutz an diesen Körperteilen.

Die Objektbibliothek basiert auf den Rollenspielregeln, und stellt im Prinzip eine Vereinfachung der Eingabe von gleichen Objekten dar. Jedes Rollenspiel hat eine Liste mit den Standardausrüstungsgegenständen, und ihren Eigenschaften. So zum Beispiel ein normales Schwert mitsamt dem Preis, den Schaden den es verursacht, sein Gewicht, ...

Der letzte Teil ist der individuelle Charakter. Dieser bestimmt nun die eigentlichen Werte und Eigenschaften des Charakters. Die Charakterdaten basieren auf den Rollenspielregeln, und binden ggf. verschiedene Objektbibliotheken ein, aus denen die einzelnen Objekte ausgewählt werden.

4.2 Geschichtsführung

Die gesamte Sprache besteht aus einer Reihe von Befehlen (Batch-Dateien), die nacheinander ausgeführt werden.

Das hat den Vorteil, daß man zu jedem Befehl einen Zeitstempel angeben kann. Über diesen Zeitstempel kann später die Geschichte des Charakters erstellt werden. Man kann es auch als persönliches Tagebuch des Charakters ansehen, bei dem nach jedem Rollenspiel-Tag die Ereignisse notiert werden, und die Veränderungen, die der Charakter durchgemacht hat.

Aus implementierungstechnischen Gründen wird die Abarbeitung der Befehle sich nicht an den Zeitstempeln orientieren, sondern an der Position in dem Quell-File. Deshalb muß bei der Erstellung des Charakters auf die Reihenfolge geachtet werden. Der Zeitstempel ist rein informativ, und hat auf die eigentliche Erstellung, und den Objekten mit den Attributen des Charakters keine Auswirkungen.

Des weiteren kann man zu jedem Befehl noch einen Kommentar hinzufügen. Wie der Name schon sagt, dient der Kommentar nicht der eigentlichen Funktionalität bei der Erstellung des Charakters sondern nur der Erstellung der Geschichte.

Es bieten sich nun folgende Möglichkeiten:

1. Der Zeitstempel & Kommentar sind in jedem Kommando enthalten:
2. Der Zeitstempel & Kommentar bilden ein eigenes Kommando.

Vor- und Nachteile der beiden Varianten:

Variante 1:

Bei Variante 1 hat jedes Kommando einen Zeitstempel. Die Kommandos können evtl. nach dem Zeitstempel sortiert werden, und sind nicht auf die Reihenfolge in dem Quell-Code angewiesen.

Beispiel:

```
CREATE OBJECT Schlafsack FROM obj
{
     DATE 1.5.1999
     COMMENT "Habe eingekauft"
}
CREATE OBJECT Seil FROM obj
```

```
{
     DATE 1.5.1999
     COMMENT "Habe eingekauft"
}
```

Variante 2

Bei Variante 2 sind die Kommandos auf die Reihenfolge in dem Quell-Code angewiesen. Es vereinfacht auch die Möglichkeiten der Block-Bildung von Ereignissen: Dazu steht ein Kommentar-Kommando im Quellcode, und alle Kommandos, die danach kommen, beziehen sich auf den Kommentar.

Es macht keinen Sinn, die Kommandos, wie zum Beispiel das Kaufen von Waren detailliert zu Kommentieren. In diesem Falle genügt der Kommentar "Bilbo hat in Kaufhausen eine Winterausrüstung gekauft", für alle folgenden Kommandos, Gib Bilbo einen Schlafsack; Geld=Geld-100; Gib Bilbo Wintermantel; Geld=Geld-120. Bei Variante 1 würde in jedem Kommando der gleiche Kommentar stehen.

Beispiel:

```
[DATE "1.5.1999"; COMMENT "Habe eingekauft"]
CREATE OBJECT Schlafsack FROM obj {}
CREATE OBJECT Seil FROM obj {}
```

Der Kommentar und damit die Geschichte des Charakters sollen nur in groben Zügen darüber informieren, was mit dem Charakter geschehen ist. Die Details sind normalerweise aus dem Quellcode ersichtlich. Deshalb habe ich mich für Variante 2 entschieden.

Die Syntax für einen Kommentar lautet:

```
[ DATE "datum"; TITLE "title"; COMMENT kommentar]
```

EBNF:

```
comment := '[' 'DATE' string ';' ['TITLE' qstring ';']
'COMMENT' string ']'
```

Zuerst wird das Datum angegeben und wird als String geliefert. Das eigentliche Format ist konfigurierbar, da verschiedene Länder eine unterschiedliche Syntax haben. Das Datum muß in Anführungszeichen angegeben werden.

Nach dem Datum erfolgt optional ein Titel. So kann man Die Geschichte in Kapitel Gliedern, oder eine kurze Zusammenfassung angeben.

Damit auch größere Texte über mehrere Zeilen eingegeben werden können, ist auch folgende Variante möglich:

Syntax:

```
[DATE "datum"; TITLE "titel"; COMMENT:START]
... langer Kommentar über mehrere Zeilen
[COMMENT:END]
```

EBNF:

```
comment := '[' 'DATE' qstring ':' ['TITLE' qstring ';']
'COMMENT:START' ']'
          string
          '[' 'COMMENT:END' ']'
```

Anmerkung: Der String in dem Kommentar behält seine Zeilenumbrüche bei! Genauso wird mit Anführungszeichen usw. verfahren. Der Kommentar ist wird durch ein [COMMENT:END] am Anfang einer neuen Zeile beendet.

4.3 Basistypen

Die Basistypen sind die verschiedenen Typen von Attributen, die in RPDL benutzt werden können. In den Basistypen werden die eigentlichen Informationen gespeichert. Es gibt folgende 5 Typen:

NUMBER

NUMBER bedeutet eine Fließkommazahl. Mit Zahlen kann man die übleichen Rechenoperationen durchführen.

INTEGER

INTEGER bedeutet eine Integerzahl. Mit Zahlen kann man die üblichen Rechenoperationen durchführen.

DICE

Eine Attribut vom Typ DICE bedeutet ein Würfelwert. Siehe nächstes Kapitel.

BOOL

Ein boolesches Attribut. Kann den Wert `true` oder `false` annehmen.

STRING

Dies ist einfacher Text. Gibt man einen Text als Konstante an, so muß er in Anführungszeichen gesetzt werden. Damit man auch ein Anführungszeichen in dem String angeben kann, muß man vor dem Anführungszeichen ein Backslash '\' setzten. Z.B: "ein \"text\"". Des weiteren gibt es noch zusätzliche Sonderzeichen, für die man den Backslash benutzen kann. Steht ein Zeichen nach dem Backslash, das nicht hier aufgeführt ist, wird es normal ausgegeben, also der Backslash einfach entfernt:

```
\" -> "
\\ -> \
\t -> <TAB>
\n -> <neue Zeile>
\{ -> {
\} -> }
```

Ein zusätzliches Feature bei einem String ist es, daß man ihn in mehreren Sprachen gleichzeitig angeben kann. Das sieht dann so aus:

```
"{DE} ein deutscher text {EN} a german text"
```

Es wird zuerst der zweistellige Ländercode in geschweiften Klammern angegeben, danach folgt der eigentliche Text in der entsprechenden Sprache. Ist kein Ländercode angegeben, so wird die aktuelle Sprache genommen, die in dem Konfigurations-File eingestellt ist. Der Ländercode ist nicht abhängig von der Groß- und Kleinschreibung.

Im folgenden wird mit folgender EBNF ein Basistyp definiert:

```
basetype := ('NUMBER' | 'INTEGER' | 'DICE' | 'STRING' | 'BOOL')
```

dazu kommen noch die Konstanten, für die direkte Zuweisung:

```
const_integer :=  [0-9]*
const_number := const_integer '.' const_integer
const_string := '"'{ [ '{'[A-Z][A-Z] '}' ] <siehe oben> } '"'
const_bool := ( 'true' | 'false' )
const  :=  (const_integer  |  const_number  |  const_string  |
const_dice | const_bool)
identifier := [a-zA-Z_][a-zA-Z0-9_]*
```

Wie man sieht, ist die Exponentialschreibweise für den type NUMBER nicht erlaubt.

4.3.1 Würfelwert

Der Würfelwert ist ein besonderer Typ von Variablen, der keinen festen Wert hat, sondern im Prinzip eine Anweisung wie ein Wert zufällig mit Hilfe von Würfeln zustande kommt. In RPDL werden keine Würfel simuliert, sondern die werden während des Spielens in der jeweiligen Situation angewendet. Dabei wird mit den entsprechenden Würfeln gewürfelt, und das Ergebnis entscheidet dann über Erfolg/Mißerfolg, oder auch darüber, wie gut eine Aktion war. Welche Würfel, und wieviele, steht in dem Würfelwert.

Ein Beispiel: Trifft der Charakter während eines Kampfes einen Gegner so schaut man nach wieviel Schaden das Schwert verursacht. Z.B "1W+1". Dann nimmt man einen 6er Würfel, addiert 1 hinzu, und das Ergebnis ist dann der eigentliche Schaden, den der Gegner nimmt.

Es gibt bei Rollenspielen nicht nur den Standardwürfel mit 6 Seiten, sondern auch 4er, 8er, 10er, 12er, 20er, 30er und 100er Würfel. (Wobei man bei dem 100er-Würfel meist zwei 10er Würfel nimmt, und der eine die Zehner-Potenz angibt, und der andere den einfachen Wert.). Der endgültige Wert aus dem Würfeln kann noch einen Modifikator haben (z.B. "1W6+1"). Multiplikationen von Würfeln, wie "1W6*2", d.h. mit einem Würfel würfeln, und das Ergebnis mit 2 multiplizieren, sind bei Rollenspielen nicht üblich. In diesem Fall wird gleich mit 2 Würfeln gewürfelt, als "2W6".

Syntax:

X = {4,6,8,10,12,20,30,100}

$n_1Wx_1 + ... + n_mWx_m$, n? INTEGER, x? X, ? i,j? {1,..,m}: x_i ? x_j

Der Würfelwert hat folgende EBNF:

```
dicevalues := '4' | '6' | '8' | '10' | '12' | '20' | '30' |
'100'
dice       := const_integer ('W'|'D') dicevalues | const_integer
const_dice := dice { '+' | '-' dice }
```

Beispiele:

1W6: Ein ganz normaler Würfel

2W6+2: Zwei 6er ('normale') Würfel, und zu dem Ergebnis wird zwei hinzuaddiert

1W4+3W8: ein 4er Wuerfel und drei 8er Würfel.

Anmerkung:

Die Syntax 1D6 ist ebenfalls erlaubt, und ist die englische Schreibweise (von *D* wie Die). Es werden also beide Varianten geparst. Wie letztendlich die Ausgabe erfolgt, mit *D* oder mit *W*, ist konfigurierbar.

4.3.2 Gültige Operationen der Basistypen

Folgende Operationen sind zwischen den einzelnen Basistypen definiert, und welcher Ergebnistyp herauskommt:

?? NUMBER +|-|*|/ NUMBER -> NUMBER:
Das Ergebnis ist das Resultat der mathematische Verknüpfung der Werte.

?? INTEGER +|-|*|/ NUMBER -> NUMBER:
Das Ergebnis ist das Resultat der mathematische Verknüpfung der Werte. Wobei der INTEGER Wert zuerst in einen NUMBER-Wert konvertiert wird.

?? NUMBER +|-|*|/ INTEGER -> NUMBER:
Das Ergebnis ist das Resultat der mathematische Verknüpfung der Werte. Wobei der INTEGER Wert zuerst in einen NUMBER-Wert konvertiert wird.

?? INTEGER +|-|*|/ INTEGER -> INTEGER:
Das Ergebnis ist das Resultat der mathematische Verknüpfung der Werte.

?? STRING + STRING -> STRING:
Das Ergebnis ist die Konkatenation der beiden Strings.

?? STRING + (INTEGER|NUMBER|DICE|BOOL) -> STRING:
Das Ergebnis ist die Konkatenation aus dem String und der ASCII-Darstellung des Wertes der zweiten Variablen.

?? INTEGER|NUMBER|DICE|BOOL + STRING -> STRING:
Das Ergebnis ist die Konkatenation aus dem String und der ASCII-Darstellung des Wertes der ersten Variablen.

?? DICE +|- DICE -> DICE
Das Ergebnis ist die Addition (Subtraktion) der einzelnen Würfel. Beispiel: "2W6" + "1W6+1W8" -> "3W6+1W8"

?? DICE +|- INTEGER -> DICE:
Der INTEGER-Wert wird als der Modifikator angesehen (entspricht dem W1), und zu diesem hinzuaddiert/subtrahiert. Das Ergebnis ist der neue Würfelwert.

?? BOOL '|' | & BOOL -> BOOL:
Entspricht den boolschen Operatoren ODER bzw. UND .

?? ! BOOL -> BOOL:
Entspricht dem boolschen Operator NICHT.

?? INTEGER = | <= | >= | != INTEGER -> BOOL:
Vergleich zweier INTEGER Werte.

?? NUMBER = | <= | >= | != NUMBER -> BOOL:
Vergleich zweier NUMBER Werte.

Die Implementierung wird die Typüberprüfung während der semantischen Analyse vornehmen. Deshalb wird im folgenden mit folgender ENBF gearbeitet:

```
value : = mwert | mwert ('+' | '-') mwert | '(' mwert ')'
mwert := identifier | identifier ('*' | '/') identifier
```

4.4 Tabellen

Eine große Rolle spielen Tabellen. Es gibt viele, nicht durch eine mathematische Funktion ausdrückbare Werte zu berechnen. Diese stehen dann in Tabellen. Tabellen werden global in dem Interpreter gespeichert, d.h. sie sind von allen Objekten aus aufrufbar.

4.4.1 Definition

Eine Tabelle hat folgendes aussehen:

Syntax:

```
TABLE type name (type param)
{ tabledata }
```

EBNF :

```
Table := 'TABLE' basetype identifier '(' basetype identifier
')' '{' tabledata '}'
tabledata := ( range: formula ';' )+
range := const
      |  const '..' const
      |  const ('<'|'<=') identifier ('<'|'<=') const
      |  ('['|']') const, const ('['|']')
      |  ('<'|'<='|'>'|'>=') const
      |  range ',' range
```

Das Schlüsselwort TABLE definiert eine Tabelle. Danach folgt der Typ des Rückgabewertes, dann des Name der Tabelle, und zuletzt der Parameter, der für den Lookup in der Tabelle benutzt wird.

Die Bereiche, die man definieren kann sind folgende:

Konstante (**`const`**):

> Der Wert gilt nur, wenn der Parameter diesem Wert entspricht.

Konstanter Bereich (**`const '..' const`**):

> Der Wert gilt, wenn der Parameter innerhalb des Bereiches liegt. Dies entspricht der Eingabe von a<=wert<=b.

Erweiterter Bereich (**`const ('<'|'<=') identifier ('<'|'<=') const`**):

> Bei dieser Art der Angabe, kann man die Eigenschaften der Ränder des Bereiches zusätzlich definieren (offener oder geschlossener Bereich.) Die mathematische Schreibweise (('['|']') const, const ('['|']') ist ebenfalls erlaubt.

Default Bereich (**`('<'|'<='|'>'|'>=') const`**):

> Kann kein Bereich gefunden werden, auf den der Parameter zutrifft, dann kann man mittels des default Bereiches dafür einen default-Wert angeben.

Mehrere Bereiche (**`range ',' range`**):

> Man kann auch verschiedene Bereiche für einen Wert angeben, indem man die Bereiche durch Kommata trennt.

Auf den Bereich folgt, durch einen Doppelpunkt (':') getrennt, der Wert der in diesem Bereich gültig ist. Der Wert ist eine Formel, wie bei den Statements innerhalb eines Objektes.

Innerhalb einer Formel, kann die Tabelle mit dem Zeichen '#' ausgeführt werden:

```
SET schaden = #damage (strength).
```

4.4.2 Beispiel

Schadenstabelle für die Stärke des Charakters.

```
TABLE NUMBER damage (NUMBER st)
{
     <1:   0;
     1..3: st;
     4..5: st+1;
     6:    st+2;
     >6:   st+4;
}
```

Es wird immer der erste Eintrag genommen, bei dem der Wert innerhalb des Bereiches ist. Wenn also zwei Bereiche überlappen, so wird der erste Bereich genommen. Gibt es keinen gültigen Bereich für den übergebenen Parameter, so wird eine Fehlermeldung ausgegeben.

4.5 Aufbau des Charakters

Um sich an den natürlichen Verhältnissen zu orientieren, ist der Charakter baumartig aufgebaut. Er hat zum Beispiel einen Rucksack, der wiederum ein Seil enthält, etc. Zum Beispiel folgendermaßen:

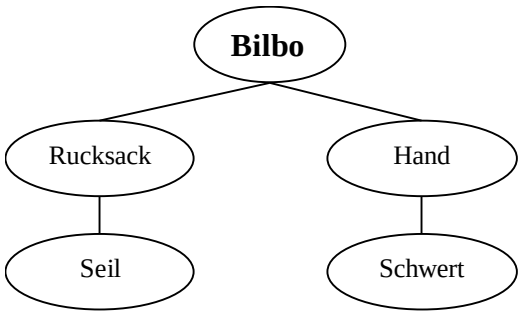


Abbildung 4.2: Beispiel eines Charakterbaumes

Dabei können die einzelnen Objekte, aus denen der Charakter besteht nicht nur reale Dinge wie ein Seil beschreiben, sondern auch abstrakte Dinge wie die Eigenschaften und Fähigkeiten einer Person. In RPDL wird dabei kein Unterschied gemacht. Es handelt dabei immer um Objekte.

Die einzelnen Objekte wiederum sind oft in Kategorien eingeteilt. Wie zum Beispiel Waffen, oder auch einzelne Waffengattungen wie Fernkampfwaffen/Nahkampfwaffen. Des weiteren sind auch meist die Zaubersprüche in Kategorien eingeteilt. Um dies abbilden zu können, ist eine Ableitungshierarchie mit (Mehrfach-) Vererbung an sinnvollsten.

Es gibt in RPDL Typen, die den abstrakten Klassen in anderen Programmiersprachen entsprechen, und die eigentlichen Objekte, die dann Instanzen diesen Typen sind.

Oft ist es auch sinnvoll, daß ein Objekt von mehreren Typen abgeleitet wird. Dies ist in RPDL ebenfalls möglich. Dadurch kann man zum Beispiel die Verzauberungen eines Objektes modellieren, indem man das Objekt zusätzlich von einer weiteren Klasse ableitet. Eine wichtige Eigenschaft von in RPDL ist es, daß sich die Typen eines Objektes ändern können. So können Typen hinzugefügt und entfernt werden. Damit

wird dem Umstand Rechnung getragen, daß ein Objekt verzaubert werden kann, und damit zusätzlich alle Eigenschaften der Verzauberung erhält. (siehe Kapitel 4.13, Modifizieren des Charakters)

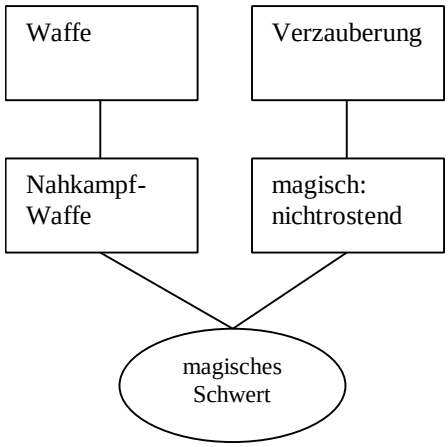


Abbildung 4.3: Beispiel Typenvererbung

Grundsätzlich kann man davon ausgehen, daß Typ-Definitionen die Implementierung der Rollenspiel-Regeln sind, und die Instanziierung der Typen, die Objekte, die eigentlichen Dinge sind, aus denen der Charakter, mit seinen Eigenschaften und Ausrüstung, besteht.

Der Charakter wird letztendlich ein gerichteter azyklischer Graph von Objekten sein, der an der Basis beginnt (mit den Basiseigenschaften, usw.) und schließlich weitere Objekte bekommt, wie Ausrüstung, etc. Das setzt voraus, daß jedes Objekt wieder Unterobjekte enthalten kann. Der gesamte Graph beschreibt dann den Charakter.

Der Vorteil eines gerichteten azyklischen Graphen ist, daß ein Objekt sich an mehrere Objekte binden kann, wie zum Beispiel ein Zweihandschwert, daß sich in beiden Händen befindet.

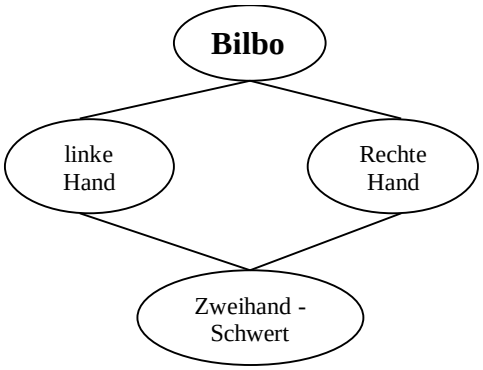


Abbildung 4.4: Beispiel mehrere Vater Objekte

4.6 Formeln

Um Werte Berechnen zu können müssen Formeln eingegeben werden können. Neben den üblichen Operationen zwischen den Basistypen, können noch Attribute, Tabellen und Spezialfunktionen benutzt werden.

Eine Formel ist wie folgt (in EBNF) definiert:

```
formula : = wert | wert op wert
wert := attribute | tablecall | const | specialfunction
tablecall := '#' identifier '(' identifier [',' identifier ')'
```

Der Zugriff auf ein Objekt bzw. Attribut erfolgt folgendermaßen:

```
object := '$' ( 'parent' | 'base' | 'this' | identifier )
attribute := [object '.'] identifier
```

Die Spezialfunktionen sind in Kapitel 4.11, Seite 27 beschrieben.

Eine Formel besteht aus den Verknüpfungen der einzelnen Werte. Siehe Basistypen Kapitel 4.3, Seite 12 für eine genauere Beschreibung der zulässigen Operationen zwischen den Werten. Die Werte können auf folgende Arten angegeben werden:

1. Direkt als Konstante
2. Als Ergebnis eines Aufrufs einer Tabelle.
3. Als Wert eines Attributs
4. Eine Spezialfunktion. Siehe Spezialfunktionen (Kapitel 4.11, Seite 27).

Der Zugriff auf ein Attribut erfolgt über ein Objekt, oder direkt, wenn man sich auf das aktuelle Objekt bezieht. Um ein Objekt anzusprechen muß man diesem das Dollar-Zeichen voranstellen. Wie zum Beispiel bei `$Schwert`. Will man ein Attribut dieses Objektes angeben, so muß man einen Punkt zwischen Objekt und Attribut setzten (zum Beispiel `$Schwert.Gewicht`).

Neben dem direktem Zugriff auf ein Objekt, kann man auch 3 Spezielle Objekte über reservierte Schlüsselwörter ansprechen. Normalerweise werden die Objekte ausschließlich darüber angesprochen, und nicht direkt:

- base: Damit wird das Basis-Objekt angesprochen.
- this: Damit wird das aktuelle Objekt angesprochen.
- parent: Damit wird das übergeordnete Vater-Objekt angesprochen.

Bei parent gibt es daß Problem, wenn mehrere Väter zu einem Objekt existieren, wie bei dem Zweihand-Schwert. In diesem Fall bezeichnet parent ein beliebiges Vater-Objekt. Dies ist ein Kompromiß mit dem man auf einfache Weise dieses Problem bewältigen kann, wenn man davon ausgeht, daß man wirklich nur einen Baum hat, und keinen azyklischen Graphen.

4.7 Attribute

Damit es keine Mehrdeutigkeiten bei den Attributen der einzelnen Typen gibt, muß jedes verwendete Attribut separat deklariert werden. Es gibt damit auch keine Verwechslung bei der Vergabe von Attributen die den gleichen Namen haben. Wie zum Beispiel bei folgendem Problem:

```
TYPE Waffe {
     INT G; // Gewicht
}

TYPE Schuh {
     INT G; // Größe
     INT Gewicht; // Gewicht
```

```
}
```

Bei diesem Code ist nicht klar, was jetzt G jeweils darstellt. Daher gibt es eine globale Liste mit allen Attributen. Da es nur Attribute gibt, die von jedem anderen Objekt angesprochen werden können, und keine "private" Attribute, hat es der Programmierer in diesem Fall schwer, die richtige Bedeutung des Attributs zu erkennen.

Ein weiteres Problem ist die Mehrfachvererbung. Würde nach diesem Beispiel ein Objekt aus diesen beiden Typen erzeugt werden, so wäre das Attribut G mehrdeutig. Bei einer globalen Deklaration der Attribute kann in diesem Fall das Attribut nur einmal angelegt werden, ohne daß es zu semantischen Mehrdeutigkeiten kommt.

Syntax zum Erzeugen eines Attributs:

```
CREATE ATTRIBUTE name BASETYPE basetype [DEFAULT dice] [COMMENT
comment];
```

EBNF:

```
cmd_attr := 'CREATE' 'ATTRIBUTE' identifier 'BASETYPE' basetype
[DEFAULT dice_value] [ 'COMMENT' qstring ]';'
```

Der Basetype gibt den Basistyp des Attributs an (wie "string" oder "int"). Der default-Wert in Form eines Würfelwertes gibt an, wie das Attribut durch Würfeln erzeugt werden kann. Dies ist bei der Implementierung eines Wizards zur automatischen Charaktergenerierung von Bedeutung. Der Kommentar ist für eine Beschreibung des Attributs vorgesehen.

4.8 Statements

Typen und Objekte bestehen aus einer Liste von Statements, die sie beschreiben. Dies ist zum Beispiel die Deklaration von Attribute, die sie besitzen, und die Anweisungen, die Auswirkungen dieses Objekts auf den Charakter beschreiben. Wenn zum Beispiel ein Manaring das Attribut Mana des Charakters erhöht, so enthält das Objekt Manaring eine entsprechende Anweisung:

```
CREATE OBJECT Manaring FROM Ring {
     SET $base.Mana = $base.Mana 10;
}
```

Es gibt folgende verschiedene Arten von Statements:

- ?? Attributdeklaration (nur bei Typen)
- ?? Anweisungen
- ?? Bedingte Anweisungen
- ?? Vorbedingungen
- ?? Zusagen
- ?? Events

4.8.1 Attributdeklaration

Die Deklaration von Attributen ist nur bei Typen erlaubt. Ein Objekt kann keine neuen Attribute definieren.

Syntax:

```
ATTRIBUTE name [ = formula];
REQUIRED ATTRIBUTE name;
```

EBNF:

```
Stmt_attr := 'ATTRIBUTE' identifier [ '=' formula ];
Stmt_attr := 'REQUIRED' 'ATTRIBUTE' identifier;
```

Dadurch wird ein neues Attribut für diesen Typ definiert. Attribute sind grundsätzlich für alle Typen/Objekte sichtbar, d.h. es gibt keine *protected* oder *private* Attribute.

Ein Attribute kann auf einen Wert initialisiert werden. Dabei kann eine beliebige Formel angegeben werden.

Das Schlüsselwort REQUIRED bedeutet, daß dieses Attribut bei der Erzeugung eines Objektes aus diesem Typ initialisiert werden muß. Damit kann man das setzten eines Attributes erzwingen. Wird das Attribut nicht initialisiert, so wird eine Fehlermeldung ausgegeben.

Fehlt beim Anlegen eines Attributs die Initialisierungsanweisung, so wird das Attribut auf folgenden Standard-Wert gesetzt:

Attributtyp	**Standard Wert**
STRING	"" (leerer String)
INTEGER	0
NUMBER	0
BOOL	true
DICE	0

4.8.2 Anweisungen

Eine Anweisung ist das Ausführen einer Formel, und die Zuweisung an ein Attribut.

Die Syntax für eine Anweisung sieht folgendermaßen aus:

Syntax:

```
SET Strength = formel;
```

EBNF:

```
stmt_set := 'SET' attribute '=' formula ';'
```

Ein Problem, ist, wie berechnet man das Gewicht, wenn zwei Hände ein Schwert halten?

4.8.3 Anweisung "parentall"

Es gibt die Möglichkeit, eine Formel auf **alle** Väterobjekte anzuwenden. Das geschieht mit folgender Anweisung:

Syntax:

```
SET $parentall.Gewicht = 10;
```

EBNF:

```
stmt_set := 'SET' '$parentall' '.' identifier '=' formula ';'
```

Bei dieser Art von Anweisung wird in jedem Vater-Objekt, das angegebene Attribut gesetzt. Existiert das Attribut in einem Vater-Objekt nicht, so wird eine Fehlermeldung ausgegeben. Mit Hilfe der Spezialfunktion countparents (Siehe Kapitel 4.11.3, Seite 28), kann man damit eine Aufteilung eines Gewichts erreichen.

```
TYPE obj {
```

```
        ATTRIBUTE GG;       // Gesamtgewicht
        ATTRIBUTE G;        // Gewicht
        SET GG = G;         // das eigene Gewicht
        SET $parentall.GG += GG/$countparents();
    }
```

Mit diesem Konstrukt, hat jedes Objekt ein Eigengewicht, und ein Gesamtgewicht. Fügt man ein Objekt an ein anderes hinzu, so wird das Gesamtgewicht aller übergeordneten Objekte verändert. In diesem Fall, wird das Gewicht gleichmäßig auf die übergeordneten Objekte verteilt.

Ohne Schwert:

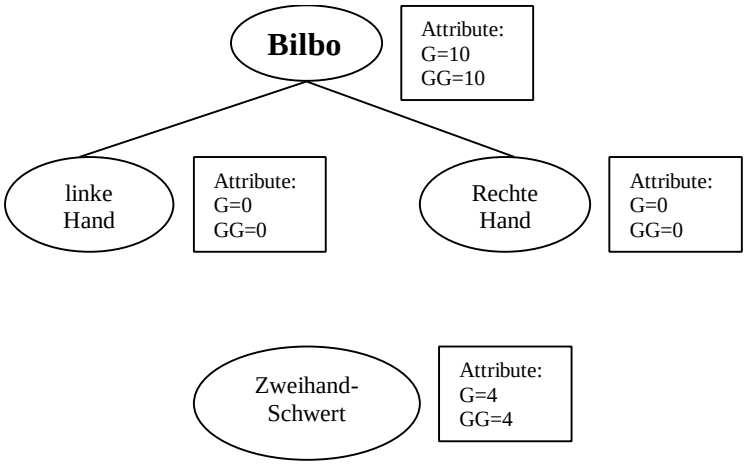


Abbildung 4.5: Charakter ohne Schwert

Nachdem das Schwert an die linke Hand hinzugefügt wurde:

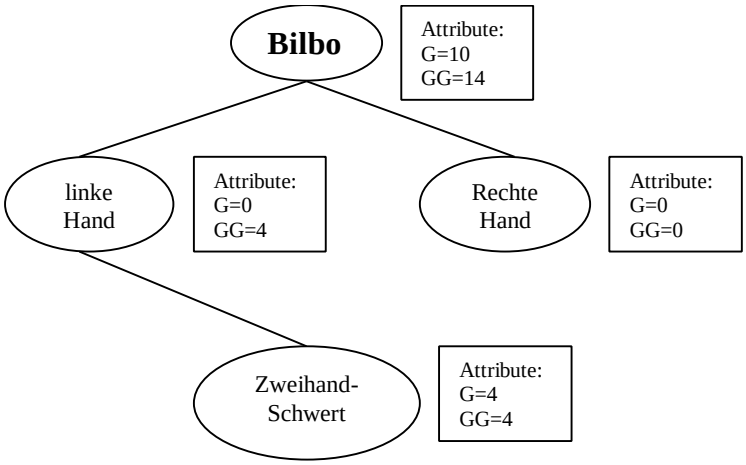


Abbildung 4.6: Charakter mit Schwert in einer Hand

Nachdem das Schwert an beide Hände hinzugefügt wurde:

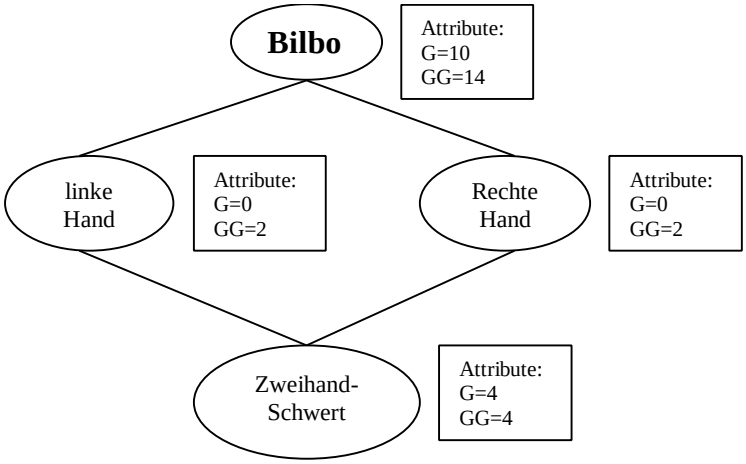


Abbildung 4.7: Charakter mit Schwert in zwei Händen

4.8.4 Anweisungen mit Prioritäten

Ein Attribut wird üblicherweise von mehreren SET-Anweisungen berechnet. Dabei spielt die Reihenfolge der Anweisungen eine bedeutende Rolle. Es geht nicht nur um das Problem der Reihenfolge von Addition und Multiplikation, sondern meist um das Problem Setzten/Verändern, wie zum Beispiel folgende zwei Anweisungen:

```
(1) SET A=10;
(2) SET A=A+1;
```

Das Problem tritt dann auf, wenn Anweisungen von Typen kommen. Sie werden üblicherweise vor den Anweisungen in dem Objekt ausgeführt. Will man aber eine Modifikations-Anweisung bei einer Typ-Deklaration vornehmen, so hat man das Problem, daß diese Anweisung bei einem späteren Setzten des Attributs verloren geht. Hier ein Beispiel:

```
CREATE TYPE Spruch {
ATTRIBUTE Level=0;
}
CREATE TYPE Feuerspruch FROM Spruch {
     // Ein Feuerspruch hat grundsätzlich einen höheren Level.
     SET Level = Level +1;
}
CREATE OBJECT Feuerball FROM Feuerspruch {
     SET Level = 10;
}
```

In diesem Beispiel soll ein Feuerspruch grundsätzlich einen Bonus von 1 auf den Level geben. Dies wird allerdings bei der CREATE OBJECT-Anweisung überschrieben.

Eine Lösung des Problems wäre es, wenn aus der Anweisung "SET Level=10" die Anweisung "SET Level=Level+10" machen würde. Allerdings hat man keinen rechten überblick mehr, wie das ganze nun berechnet würde. Außerdem müßte man bei einer Änderung des Grundwertes (hier bei der Attributsdeklaration) dann den Level von jedem abgeleiteten Objekt ändern. Letztendlich ist es auch ein semantischer Unterschied ob der Level gesetzt oder verändert wird. Es soll ja zu Schluß der Bonus von 1 vergeben werden, und nicht vorher.

Eine bessere Lösung ist die Verwendung von Prioritäten bei der SET-Anweisung. Dabei definiert man die Anweisung "SET Level=Level+1" als Anweisung von niedriger Priorität, so daß sie immer nach den Anweisungen mit normaler Priorität ausgeführt wird.

Die Priorität einer SET-Anweisung wird durch die Verwendung der Schlüsselwörter "SETFIRST" bzw. "SETLAST" anstatt "SET" definiert. Eine SETFIRST-Anweisung wird vor einer SET-Anweisung ausgeführt, und diese wiederum vor einer SETLAST-Anweisung. Anweisungen mit gleicher Priorität werden in der üblichen Reihenfolge ausgeführt. Es gibt also 3 verschiedene Prioritäten.

Zur Lösung des obigen Beispiels man die Anweisung "SETLAST Level=Level+1" benutzen.

4.8.5 Bedingungen

Es gibt auch die Möglichkeit, Kommandos nur bedingt auszuführen. Das kann hilfreich sein, wenn das aktuelle Objekt eine Sonderfunktion hat, die auf einem anderem Objekt basiert. Zum Beispiel könnte ein Schutzschild besser sein, wenn man ein passendes Energiepack eingebaut ist.

Die Syntax ist folgende:

```
IF (bool-formula) {commands} [ else {commands} ]
```

EBNF:

```
stmt_cond := 'IF' '(' boolean formula ')' '{' {ifstatement} '}'
[ 'else' '{' {ifstatement} '}' ]
ifstatement := stmt_set
```

Für das Schutzschild könnte das folgendermaßen aussehen:

```
// dieses Objekt enthält mind. 1 Reaktor
IF ($count(Reaktor) > 0) {
     SET $base.shield += 10;   // der Schild erhöht sich um 10
} else {
     SET $base.shield += 2;    // auf reserve…
}
```

Um die Implementierung zu vereinfachen, sind keine geschachtelten if-Anweisungen möglich. Siehe Kapitel 5.4.2, Handhabung von Verzweigungen (IF).

4.8.6 Zusagen (Assertions)

Zusagen sind dazu vorhanden, um gewisse Aussagen abprüfen zu können. Zum Beispiel kann ein Rucksack nicht unbegrenzt Dinge aufnehmen. Deshalb kann man mittels einer Zusage prüfen, ob zum Beispiel das Gewicht der enthaltenen Gegenstände nicht überschritten ist. Syntax:

```
ASSERT WARN formel "fehlertext";
```

EBNF:

```
stmt_assertion := 'ASSERT' ('ERROR' | 'WARN' ) '(' formula ')'
[ formula ];
```

Die Formel muß einen BOOL-Wert ergeben, und wenn dieser `false` ist, wird ein Fehler (bzw. eine Warnung) ausgegeben.

Zusagen werden immer am Ende der Berechnung ausgewertet, egal an welcher Position sie stehen. Sie haben die niedrigste Priorität von allen Statements.

Trifft eine Zusage nicht zu, so wird als Fehlermeldung das Ergebnis der Formel (als STRING) zurückgeliefert. Da grundsätzlich jeder Ausdruck in einen String gewandelt werden kann ist das eine komfortable Lösung um eventuell Werte von Attributen in der Fehlermeldung angeben zu können.

4.8.7 Vorbedingungen

Vorbedingungen sind von der Syntax her wie Zusagen. Der Unterschied besteht darin, daß sie vor dem Hinzufügen zu dem eigentlichen Basis-Objekt geprüft werden. Schlägt eine Vorbedingung fehl, so kann das Objekt nicht hinzugefügt werden, und eine Fehlermeldung wird ausgegeben. Die Fehlermeldung wird als Formel implementiert, die in einen String konvertiert wird. Damit ist es möglich auch Werte von Attributen mit auszugeben.

Der zweite Unterschied zu einer Zusage besteht darin, daß eine Vorbedingung nur **einmal** geprüft wird, während ein Zusage immer geprüft wird.

Die Syntax lautet:

```
REQUIRE (bool-formel) "fehler";
```

EBNF:

```
REQUIRE '(' formula ')' [ formula ];
REQUIRE OBJECT '(' '$' object ')' [ formula ];
```

Die erste Variante ist die allgemeine Syntax, und benötigt einen booleschen Ausdruck. Ist die Bedingung nicht erfüllt, so wird eine Fehlermeldung mittels der Formel errormessage (üblicherweise ein String) ausgegeben.

Eine besondere Variante ist REQUIRE OBJECT. Diese Variante ist dafür vorhanden, wenn dieses Objekt ein anderes Objekt voraussetzt. Dies ist bei einer hierarchischen Struktur der Fall, wie z.B. bei Zaubersprüchen: Eine Bedingung für den explosiven Feuerball ist, daß man den Spruch Feuerball schon kennt. Das sähe dann so aus:

```
CREATE OBJECT explosiverFeuerball FROM Feuerspruch
{
     REQUIRE OBJECT ($Feuerball) "Explosiver Feuerball benötigt den
Feuerball-Spruch”;
}
```

Das ganze könnte man auch mit der ersten Variante machen:

```
REQUIRE (%countall(Feuerball, $base)>0);
```

Der Grund für diese Sonderbehandlung in diesem Fall liegt darin, daß später das Frontend die Möglichkeit haben soll, alle fehlenden Objekte einfügen zu können. Also wenn der Benutzer den explosiven Feuerball lernen will, und er kann noch keinen Feuerball zaubern, dann soll mittels eines Requesters der Benutzer die Möglichkeit haben, alle Objekte, die Vorbedingung sind, auf einmal hinzufügen zu können. Dies wäre bei der alternativen Darstellung (`REQUIRE (%countall(Feuerball, $base)>0)`) nicht möglich.

4.8.8 Events

Es gibt die Möglichkeit, Anweisungen auszuführen, wenn eine Variable sich ändert. Das kann man dazu benutzen, um bei Erreichen eines neuen Levels einen Zähler zurückzusetzen. Der Zähler kann dann die Anzahl der erlernten Sprüche pro Stufe begrenzen. Dies ist in dem Rollenspiel Rolemaster der Fall.

Eine weiteres Beispiel für Events ist, beim Umsetzten eines Flags einige Attribute zu speichern.

Als Anweisungen in dem Event-Block sind nur SET-Anweisungen zugelassen. Der Grund dafür ist, daß die Zyklenerkennung bei einer IF-Anweisung erschwert werden würde.

Syntax:

```
ON CHANGE (attr) {changestatements}
```

EBNF :

```
stmt_event := 'ON' 'CHANGE' '(' attribute ')' '{'
{changestatement} '}'
changestatement := stmt_set
```

Beispiel :

```
BASE Character;
CREATE OBJECT Bilbo FROM Character {
     ATTRIBUTE experience;
     ATTRIBUTE level = #LevelTab(experience);
     ATTRIBUTE gelernt = 0;
     ON CHANGE (level) {
          SET gelernt = 0;
     }
     ASSERT (gelernt < 3) "Bilbo hat in diesem Level schon genug
gelernt”;
}
TYPE Fähigkeit {
     ATTRIBUTE skill;
     ON CHANGE (skill) {
          SET $base.gelernt += gelernt;
     }
}
```

4.9 Typen

Jedes Objekt, daß später erzeugt wird, wird aus einem (oder mehreren) Typ(en) generiert. Ein Typ kann mehrere Statements enthalten, und wiederum von anderen Typen abgeleitet werden.

Ein Typ wird folgendermaßen erzeugt:

Syntax:

```
CREATE TYPE type FROM parenttype, parent2 {typestatements}
```

EBNF:

```
cmd_type := 'CREATE' 'TYPE' identifier ['FROM' identifier
{,identifier} ] '{' typestatements '}'
```

Der FROM Parameter gibt an, welches die Oberklasse(n) des neuen Typs ist.

Beispiel:

```
CREATE TYPE MissileWeapon FROM Weapon
{
     ATTRIBUTE range;
     ...
}
```

Ein Typ kann folgende Statements enthalten:

- ?? Attribute
- ?? Anweisungen

?? Zusagen
?? Vorbedingungen
?? Events

Das folgende Diagramm gibt eine Übersicht über die Beziehungen zwischen Attributen und Typen:

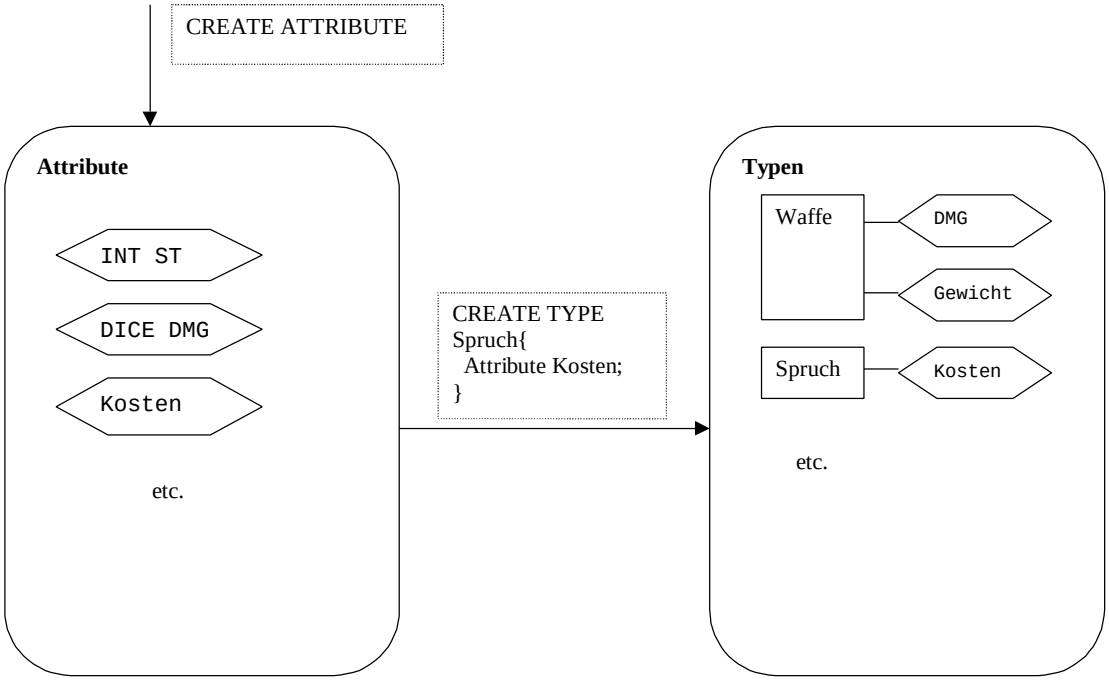


Abbildung 4.8: Beziehung zwischen Attributen und Typen

4.10 Objekte

Objekte werden aus einem oder mehreren Typen gebildet. Faßt man einen Typ als Eigenschaft auf, so kann man mittels der Typen verschiedene Eigenschaften für ein Objekt festlegen. So kann zum Beispiel ein Typ "magical" festgelegt werden, der einem Objekt das Attribut "Magie" erzeugt. Dann kann man relativ einfach prüfen, ob ein Objekt magisch ist, indem man schaut, ob es vom Typ "magical" ist.

Die Syntax zur Erzeugung eines Objektes ist folgende:

```
CREATE OBJECT identifier FROM type,type2… { objstatements }
```

EBNF:

```
cmd_object := 'CREATE' 'OBJECT' identifier 'FROM' identifier
{',' identifier } '{' { objstatement } '}'
objstatement := stmt_set | stmt_cond | stmt_assert |
stmt_event;
```

Grundsätzlich gilt folgendes bei Objekten:

?? Die Statements aus den Typen werden zuerst ausgeführt. Die Statements aus dem Typ, der zuerst angegeben ist, werden vor den anderen ausgeführt.
?? Gibt es für ein Attribut mehrere Deklarationen, so wird das Attribut nur einmal angelegt.

4.11 Spezialfunktionen

Manche Rollenspiele verlangen noch nach weiterer Funktionalität. Diese gibt es dann über die Spezialfunktionen, wie zum Beispiel das Zählen der Anzahl von Objekten eines Typs.

Spezialfunktionen beginnen alle mit einem Prozent-Symbol (%).

In den folgenden Unterkapiteln ist die Syntax der einzelnen Spezialfunktionen jeweils angegeben.

4.11.1 count/countall

Die Funktion count bzw. countall zählt, wie viele Objekte eines Typs ein Objekt enthält. countall zählt die Objekte inklusive der Unterobjekte (rekursiv), während count nur die Unterobjekte des aktuellen Objekts zählt. Wird zusätzlich ein Objekt angegeben, so startet die Zählung ab diesem Objekt. Die Zählung beinhaltet nicht das Startobjekt!

Syntax:

```
%count(type [, Objekt])
%countall(type [, Objekt])
```

EBNF:

```
specialfunction := ('%count' | '%countall' ) '(' identifier
[',' object] ')'
```

Diese Spezialfunktion liefert einen Wert vom Basistyp INT zurück.

Beispiel:

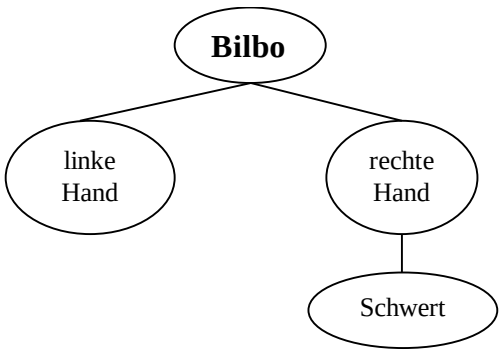


Abbildung 4.9: Beispiel Charakter

Gehen wir davon aus, daß alle Objekte den Typ "obj" haben. Dann ergibt sich folgendes:

%count(obj, base) => 2

%countall(obj, base) => 3

4.11.2 sum/sumall

Die Funktion sum bzw. sumall bildet die Summe aus einem Attribut, von allen Unterobjekten. sumall berechnet die Summe rekursiv über alle Unterobjekte, sum macht das nur über die Objekte, die direkt mit dem aktuellen Objekt verknüpft sind.

Syntax:

```
%sum(attribut [, Objekt])
%sumall(attribut [, Objekt])
```

EBNF:

```
specialfunction := ('%sum' | '%sumall' ) '(' identifier [',' object] ')'
```

Diese Spezialfunktion liefert einen Wert vom Basistyp des übergebenen Attributs zurück.

Hinweis: Diese Funktionen berücksichtigen, wenn ein Objekt mehrere Väter besitzt.

4.11.3 countparents

Die Funktion countparents zählt, wie viele Väterobjekte ein Objekt besitzt.

Syntax:

```
%countparents([Objekt])
```

EBNF:

```
specialfunction := '%countparents' '(' [ object ] ')'
```

Diese Spezialfunktion liefert einen Wert vom Basistyp INTEGER zurück.

4.11.4 istype

Die Funktion istype prüft, ob ein Objekt von einem bestimmten Typ ist, und liefert entsprechend true oder false zurück. Diese Funktion kann dazu benutzt werden, um zum Beispiel einen Ring zu "aktivieren" wenn er an der Hand getragen wird.

Die Syntax lautet:

```
%istype(objekt, typ)
```

EBNF:

```
specialfunction := '%istype' '(' object ',' identifier ')'
```

Beispiel:

```
CREATE OBJECT ManaRing FROM Ring
{
     IF (%istype($parent, Hand)) {
          $base.Mana += 10;
     }
}
```

4.12 Charakterbaumerstellung

Bis jetzt gibt es nur eine Liste mit dem Objekten, die dem Charakter gehören. Damit die Objekte miteinander interagieren können (also die Anweisungen usw. ausgeführt werden können) müssen sie zu einem Baum, bzw. einem gerichteten azyklischem Graph zusammengefügt werden. Dies geschieht mittels des LINK-Kommandos:

Syntax:

```
LINK objekt1 TO objekt2;
```

EBFN:

```
cmd_link := 'LINK' identifier 'TO' identifier ';'
```

Damit wird ein Objekt an ein anderes gebunden. Dabei dürfen keine Zyklen entstehen. Dies wird automatisch erkannt, und eine entsprechende Fehlermeldung ausgegeben.

Das Basis-Objekt wird implizit angelegt, indem ein Objekt von einem bestimmten Typ angelegt wird. Welcher Typ das ist, wird mit dem BASE-Kommando festgelegt.

Syntax:

```
BASE Character;
```

EBNF:

```
cmd_base := 'BASE' identifier;
```

Da nur ein Charakter auf einmal bearbeitet werden kann, gibt es eine Fehlermeldung, wenn versucht wird, mehrere Objekte des Base Typs anzulegen.

Für die erzeugten Objekte gilt grundsätzlich:

Entweder sie sind mit dem Basis-Objekt verbunden, dann werden die Statements ausgeführt, oder sie befinden sich noch frei im Objektpool, dann werden die Statements nicht ausgeführt.

Um eine Verknüpfung zu lösen, gibt es das UNLINK Kommando:

Syntax:

```
UNLINK objekt1 FROM objekt2;
```

EBFN:

```
cmd_unlink := 'UNLINK' identifier 'FROM' identifier ';'
```

Damit wird eine Verbindung dieser beiden Objekte aufgehoben.

Ein Objekt kann mit dem DESTROY-Kommando entfernt werden,:

Syntax:

```
DESTROY OBJECT objekt;
```

EBNF:

```
cmd_destroy := 'DESTROY' 'OBJECT' identifier ';'
```

Im Gegensatz zu UNLINK wird das Objekt gelöscht, und auch alle Objekte die an dem gelöschten Objekt sich befinden! Das Objekt ist danach auch nicht mehr im Objekt-Pool vorhanden.

Beispiel:

```
CREATE OBJECT LinkeHand FROM Hand {}
CREATE OBJECT RechteHand FROM Hand {}
CREATE OBJECT Schwert FROM Schwert {}
CREATE OBJECT Bilbo FROM Character {}
LINK LinkeHand TO Bilbo;
LINK RechteHand TO Bilbo;
```

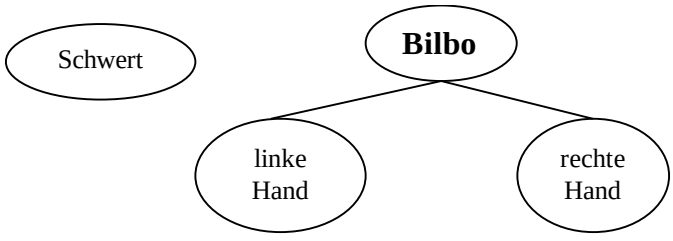


Abbildung 4.10: Charakter ohne verknüpftem Schwert

```
LINK Schwert TO RechteHand;
```

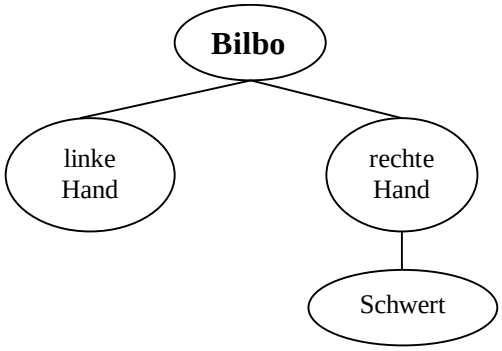


Abbildung 4.11: Charakter mit Schwert in rechter Hand

```
// Jetzt in die linke Hand geben:
UNLINK Schwert FROM RechteHand;
LINK Schwert TO LinkeHand;
```

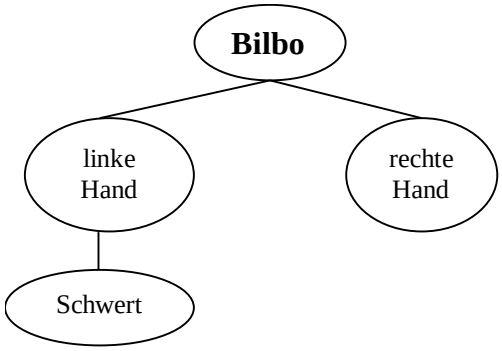


Abbildung 4.12: Charakter mit Schwert in linker Hand

```
// Jetzt in beide Hände nehmen:
LINK Schwert TO RechteHand;
```

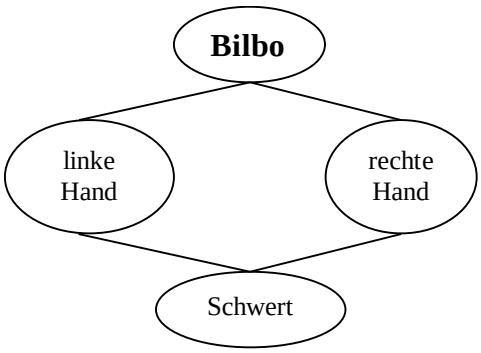


Abbildung 4.13: Charakter mit Schwert in beiden Händen

```
// Das Schwert wird weggeworfen:
DESTROY Schwert;
```

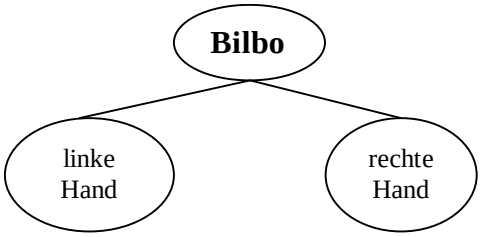


Abbildung 4.14: Charakter ohne Schwert

Dadurch, daß es einen einzigen Pool mit den Objekten gibt, sind alle Objekte eines Charakters eindeutig bestimmt. Es ist dadurch nicht möglich, daß es ein Objekt mit der gleichen Bezeichnung zweimal gibt.

Die Folgende Abbildung verdeutlicht die Beziehung zwischen Typen, Objekten und den Verknüpfungen:

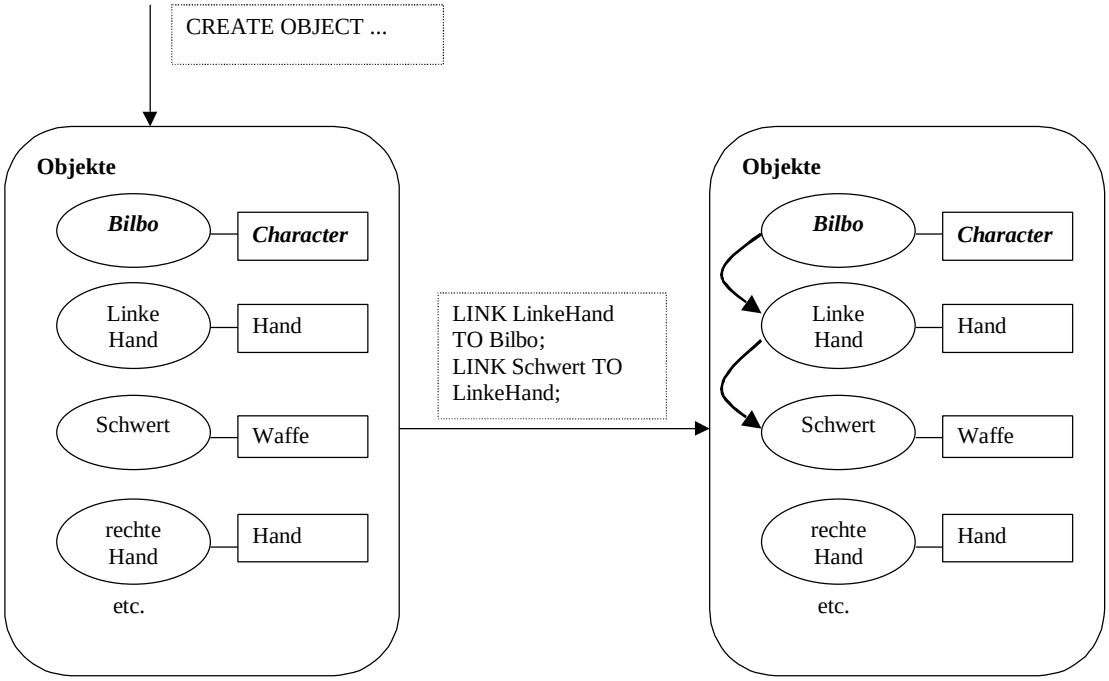


Abbildung 4.15: Verknüpfen von Objekten

4.13 Modifizieren des Charakters

Eine wichtige Funktionalität ist die Modifikation von Objekten.

Syntax:

```
MODIFY OBJECT object { modstatements }
```

EBNF:

```
cmd_modify := 'MODIFY' 'OBJECT' object '{' { modstatement } '}'
modstatement := stmt_set | stmt_addtype | stmt_removetype
```

Es wird immer ein Objekt modifiziert. Die Statements werden einfach an die Liste der Statements des Objektes angefügt.

Zusätzlich kann man mit Hilfe des Modify-statements den Typ des Objektes verändern, indem man einen zusätzlichen Typ hinzufügen, oder entfernen kann.

Dazu gibt es zwei weitere Statements:

Syntax:

```
ADD TYPE typ;
REMOVE TYPE typ;
```

EBNF:

```
stmt_addtype := 'ADD' 'TYPE' identifier;
stmt_removetype := 'REMOVE' 'TYPE' identifier;
```

Wird ein neuer Typ zu dem Objekt hinzugefügt, so bekommt es alle seine Statements. Wird ein Typ von dem Objekt entfernt, so werden alle Anweisungen, die dieser Typ hinzugefügt hat entfernt. Dabei wird Rücksicht auf evtl. mehrfach definierte Attribute genommen.

4.14 Objektbibliothek

Es gibt die Möglichkeit, schon vorgefertigte Objekte zu benutzen. Diese befinden sich in der Objektbibliothek. Die Objektbibliothek hat prinzipiell die gleichen Eigenschaften, wie ein Charakter, mit der Einschränkung, daß es kein Basisobjekt gibt, und es entsprechend auch keine aktiven Statements der einzelnen Objekte geben kann.

Die entsprechenden Kommandos für die Objektbibliothek sind folgende:

Charakter Kommando	Bibliothek Kommando
CREATE OBJECT	CREATE LIBRARY OBJECT
MODIFY OBJECT	MODIFY LIBRARY OBJECT
DESTROY OBJECT	DESTROY LIBRARY OBJECT

Um ein Objekt aus der Objektbibliothek zu benutzen, gibt es den Befehl

Syntax:

```
CREATE OBJECT name FROM LIBRARY name;
```

EBNF:

```
cmd_clone := 'CREATE' 'OBJECT' identifier 'FROM' 'LIBRARY'
identifier;
```

Damit wird eine Kopie des Objektes aus der Objektbibliothek erzeugt, und dem Charakter hinzugefügt. Damit man ein Objekt auch mehrfach benutzen kann, muß ein neuer Name für das Objekt angeben werden. So kann man ohne Probleme mehrere Seile haben, die aus dem gleichen Bibliotheks-Objekt erzeugt wurden.

5 Implementierung

5.1 Allgemein

RPDL in Java besteht aus den Klassen in der Package *de.mutschler.rpdl*. Da die Packagekonvention von Java sich aus einem Domain-Namen herleitet, und mir, Michael Mutschler, die Domain mutschler.de gehört, bietet sich der Name an.

Des weiteren basiert die Implementierung auf Java 1.2. Mittlerweile ist es auf den meisten Plattformen verfügbar, und es besteht deshalb kein Grund noch Java 1.1 kompatibel zu bleiben. Ein Vorteil von Java 1.2 ist auch das Collection API, mit dem man die verschiedenen benutzten Container für die Objekte (wie Listen, Sets, Maps) elegant implementieren kann.

Prinzipiell gibt es eine Basisklasse (RpdlBase), die die komplette RPDL-Umgebung handhabt. Will man mehrere Charaktere erstellen, kann man mehrere Instanzen von RpdlBase bilden. Sie enthält die verschiedenen Container für die Elemente (Tabellen, Typen, Bibliothek, etc.).

Generell habe ich folgende Regeln bei der Implementierung berücksichtigt:

- ?? Der Zugriff auf Attribute erfolgt immer über Wrapper-Methoden (z.B. getString(), setString())
- ?? Die Methode toString(), die jedes Java-Objekt enthält wird nur für debug-Zwecke benutzt.
- ?? Fehler werden als RpdlException geworfen.
- ?? Zur Vereinfachung des Debugging ist jede Klasse von LogObj abgeleitet, die Methoden für die Ausgabe zur Verfügung stellt.

5.2 Einlesen

Das Einlesen der Files geschieht über einen Lex/Yacc Importer. Dazu wird die Java-Implementierung dieser Tools benutzt: Für den Lexer JFlex [11], und für Yacc das Programm CUP [12]. In CUP werden die einzelnen Regeln reduziert, und die Kommandos an das RpdlBase-Objekt mittels addCommand() hinzugefügt. Als zentrale Klasse zum Einlesen dient de.mutschler.rpdl.fileimporter.FileImporter. Sie bekommt ein RpdlBase Objekt, und einen Filenamen, und importiert dieses File mittels JFlex & CUP.

Jedes Kommando, das es in Rpdl gibt, hat eine eigene Klasse, die einen Container für die entsprechenden Parameter bildet. All diese Klassen werden von der abstrakten Klasse abgeleitet.

Einen Vorteil bietet die einfache Schnittstelle zu dem Basis Objekt: Es läßt sich problemlos ein include-statement einbinden: Es wird nur einen neue Instanz der Klasse FileImporter mit dem einzubindenden File erzeugt, und das entsprechende File eingelesen. Das ganze kann rekursiv benutzt werden, um so alle weiteren Files einlesen zu können. Die eingelesenen RPDL-Kommandos werden dabei an das gleiche Basisobjekt gesendet.

Das folgende Diagramm verdeutlicht den Ablauf.

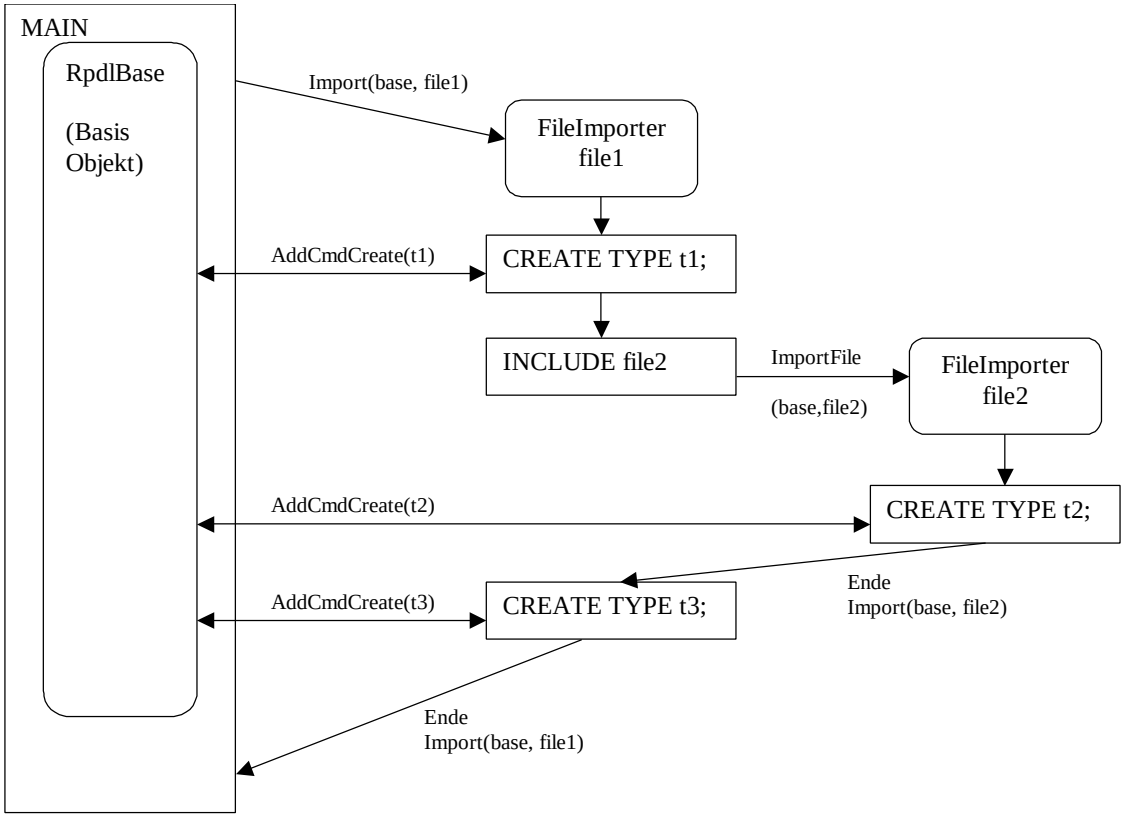


Abbildung 5.1: Ablauf des Einslesens der Kommandos

In der Implementierung sieht das so aus: Der FileImporter, ist aus einem Lexer & einem YACC erstellt, und generiert Objekte des Typs RpdlCommand. Dies ist die Basisklasse, von der alle Kommandos abgeleitet sind. Wie zum Beispiel CmdCreateAttr, CmdLink, ... Alle diese Kommandos befinden sich in dem Package de.mutschler.rpdl.command. Die Klasse RpdlCommand verfügt nur über eine Methode: add(RpdlBase). Diese kann Kommando-spezifisch implementiert werden. Und wird von der RpdlBase während der addCommand() Methode aufgerufen, um mittels der Parameter die RpdlObjekte zu generieren, und in das RPDL-System einzubinden.

Des weiteren wird am Ende von addCommand() der CalcHandler aufgerufen, um neue Berechnungen durchzuführen. Dies passiert nur, wenn eine Veränderung des Charaktergraphen stattgefunden hat.

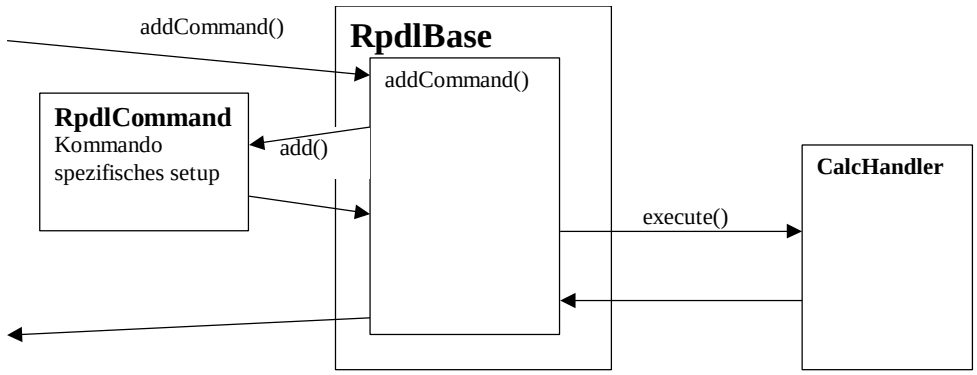


Abbildung 5.2: Ablauf der Analyse der Kommandos

5.2.1 Statische Analyse beim Einlesen

Wird ein Kommando zu der RpdlBase hinzugefügt, findet die erste statische Analyse statt. Dazu wird die Methode add() der Klasse RpdlCommand aufgerufen. Diese erledigt die eigentliche Arbeit für das Eintragen in die RpdlBase-Struktur. z.B. Fügt CmdCreateAttr.add() dort das Attribut hinzu. Des weiteren findet in dieser Methode die eigentliche semantische Analyse von RPDL statt.

Ein weiteres Beispiel für eine Analyse, die hier stattfindet, ist die Typ-Prüfung von Formeln. Eine Ungültige Zuweisung (z.B: Multiplikation von Strings) wird hier schon erkannt.

5.3 Statements

Eine wichtige Rolle bei RPDL spielen Statements. Sie bestimmen letztendlich die Eigenschaften der Rpdl-Objekte. Der folgende Baum zeigt die Klassenhierarchie der Package de.mutschler.rpdl.statement:

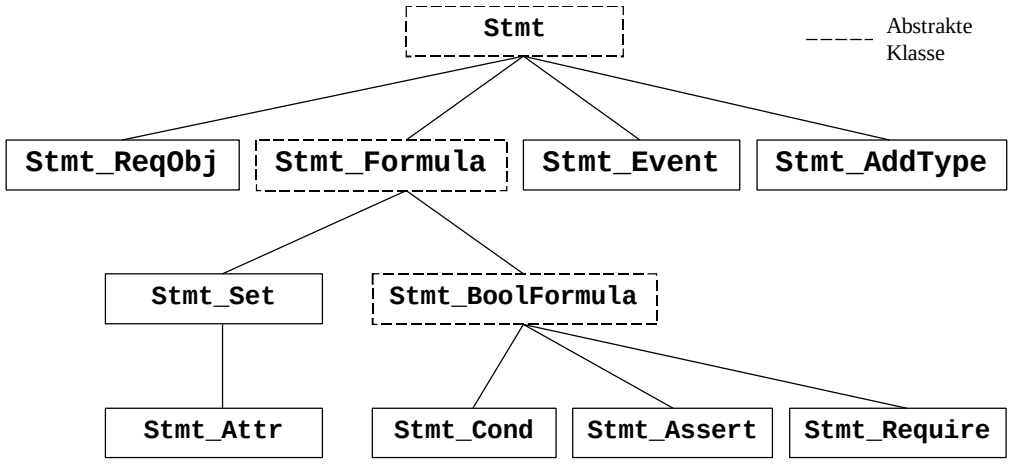


Abbildung 5.3: Klassenhierarchie der Statements

Stmt, die Basisklasse aller Statements, definiert die Schnittstelle für den Zugriff auf die einzelnen Statements. Hier die wichtigsten Aufgaben von Statements:

- ?? ID-Manager: Jedes Statement bekommt eine eindeutige, aufsteigende ID. Dadurch können die Statements später in dem Charakterbaum sortiert werden, wenn sie gleiche Priorität haben. Außerdem vereinfacht dies die Implementierung von equals(), dem Test auf Gleichheit.
- ?? Die Methode checkStatement(): Sie wird zum Zeitpunkt von addCommand() aufgerufen. Genauer von RpdlCommand.add() der jeweiligen Kommandoimplementierung (z.B. von CmdCreateObject().add()). In checkStatement() findet die statische Analyse des Statements statt. Sie bekommt ein Entity (Die Basisklasse von RpdlObject und RpdlType) übergeben, und kann sich so auf das zugehörige Objekt beziehen.
- ?? Die Methode setDerivedEntity(): Damit wird das RpdlObject bzw RpdlType gesetzt in dem das Statement definiert ist.
- ?? Die Methode onBaseLink(): Diese wird aufgerufen, wenn das Statement aktiv wird, also zu einem Objekt gehört, das neu in den Charakterbaum aufgenommen

(oder entfernt) wird. Üblicherweise wird diese Methode durch den Link-Befehl getriggert. Hier werden die Events für die Berechnung aufgesetzt (Eintrag in die CalcList & TriggerList von Attributen.) etc.

?? Die Methode execute(): Sie führt das Statement aus. Hier werden keine Abängigkeiten usw. geprüft, sondern nur die Anweisung ausgeführt. Die Abhängigkeiten werden über onBaseLink() gesetzt.

?? Die Methode trigger(): Mit dieser Methode werden die notwendigen Aktionen erledigt, damit dieses Statement neu berechnet wird. Z.B. Eintrag der Zielattribute in den CalcHandler.

?? Die Methoden getDependencies() & getTargets() liefern die entsprechenden Abhängigen Objekte zurück. Dadurch kann ein Großteil der Berechnungen in der Ausgangsklasse Stmt erledigt werden. Es vereinfacht die Implementierung der Unterklassen.

?? Für das Sortieren der Statements bei der Berechnung kann den einzelnen Typen von Statements individuelle Prioritäten gegeben werden. Diese wird von der Methode getPriority() geliefert. Z.B. hast Stmt_Assert die niedrigste Priorität, da sie **nach** dem Ausführen von SET-Anweisungen ausgeführt werden muß.

5.4 Algorithmus zur Berechnung der Attribute

Im Folgenden erfolgt die Entwicklung eines Algorithmus, der die Berechnung der Attribute vornimmt. Es werden auch gleichzeitig die jeweiligen Probleme der Lösungen diskutiert.

Als Beispiel soll dieser einfache Charakter dienen:

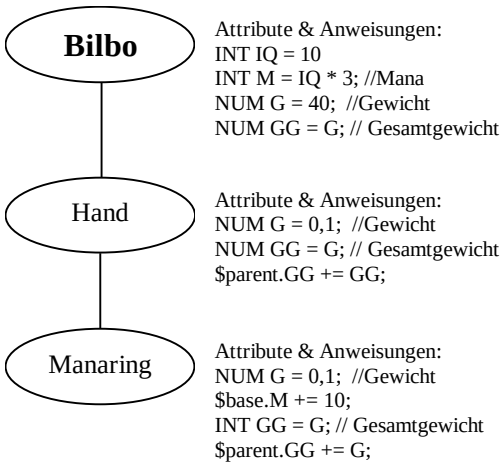


Abbildung 5.4: Beispielcharakter für Berechnung

Die Anweisungen um diesen Charakter zu erzeugen sind folgende:

```
CREATE ATTRIBUTE G BASICTYPE number COMMENT "Gewicht";
CREATE ATTRIBUTE GG BASICTYPE number COMMENT "Gesamtgewicht";
CREATE ATTRIBUTE IQ BASICTYPE int COMMENT "Intelligenz";
CREATE ATTRIBUTE M BASICTYPE int COMMENT "Zauberpunkte";
CREATE TYPE obj {
     ATTRIBUTE G;
     ATTRIBUTE GG;
     SET $parent.GG += G;
```

```
}

CREATE TYPE char {
     ATTRIBUTE G;
     ATTRIBUTE GG;
     SET GG = G;
     REQUIRED ATTRIBUTE IQ;
     ATTRIBUTE M;
     SET M = IQ * 3;
}

BASE char;

CREATE OBJECT Bilbo FROM char {
     SET IQ = 10;
}

CREATE OBJECT Hand FROM obj {
     SET G = 0,1;
}
CREATE OBJECT Manaring FROM obj {
     SET G = 0,1;
     SET $base.M += 10;
}

LINK Hand TO Bilbo;
LINK Manaring TO Hand;
```

Betrachten wir zuerst nur die Berechnung von M ohne den Manaring:
Dabei sind folgende Anweisungen im Spiel:
(1) IQ = 10;
(2) M = IQ * 3;

Nehmen wir als erstes einen einfachen Algorithmus:

1. Berechne die Anweisung, und trage den Wert in das Zielattribut ein.

Das ergibt als Ergebnis: IQ=10; M=30

Das Problem sind nun Anweisungen die Attribute hinterher ändern. Da Attribute auch nachträglich verändert werden können, und die Auswirkungen auch auf vorhergehende Formeln sich auszuwirken haben, schlägt der Algorithmus nach dieser Anweisung fehl:

```
MODIFY OBJECT Bilbo {SET IQ = 12;}
```

So erhalten wir in der Berechnung nun
(3) IQ = 12

Das Ergebnis sähe dann so aus: IQ = 12; M=30;

Da aber die Anweisung (2) noch immer gültig ist, muß sie bei der Berechnung von M weiterhin berücksichtigt werden

Das Ergebnis soll IQ=12; M=36 sein. Daher muß Anweisung (2) noch einmal ausgeführt werden.

Versuche, das Ganze über slicing-Techniken zu realisieren sind fehlgeschlagen. Die üblichen Algorithmen, wie zum Beispiel vom M. Weiser [7], Arbeiten auf Analysen von Quellcode, und ihren Abhängigkeiten. RPDL dagegen bildet die Abhängigkeiten erst, und benötigt die Ergebnisse weniger für eine Gültigkeitsdauer-Beschreibung der Attribute als vielmehr der Berechnung der abhängigen Attribute.

Es ist also notwendig, daß eine Formel neu berechnet wird, sobald sich eins ihrer Attribute ändert. Eine Formel, die von anderen Attributen abhängig ist, (z.B. "M=IQ*3" ist von IQ abhängig), trägt sich in eine Liste, die Triggerliste, aller abhängigen Attribute ein. Diese Formeln werden dann nach der Berechnung des Attributs noch einmal ausgeführt.

Die Triggerlisten von IQ & M sehen so aus:

IQ: {2}

M: {}

Der Algorithmus zum Ausführen einer neuen Anweisung lautet nun:

1. Berechne die Anweisung, und trage den Wert in die Ergebnisvariable ein.
2. Berechne alle Anweisungen neu, die in der Triggerliste für die Ergebnisvariable stehen.

Da die Formel (2) von IQ abhängig ist, wird M nach der Berechnung von (3) ebenfalls neu berechnet.

Dadurch ergibt sich ein neues Problem: Es dürfen keine Zyklen vorkommen.

Ein Zyklus wäre zum Beispiel

(4) IQ = IQ +1;

Dies ist ein lokaler Zyklus, der allerdings zugelassen werden soll.

Es könnte ja sein, daß der Charakter permanent seinen IQ mit folgender Anweisung gesteigert hätte:

```
MODIFY OBJECT Bilbo {SET IQ = IQ + 1;}
```

Die Berechnungen wären nun :

(1) IQ = 10;

(2) M = IQ * 3;

(3) IQ = 12

(4) IQ = IQ +1;

Nachdem (4) nun als Anweisung zum Basis-Objekt hinzugefügt wurde, sehen die Triggerlisten folgendermaßen aus :

IQ: {2, 4}

M: {}

Da aber (4) selber den IQ berechnet, würde das eine Endlosschleife in der Berechnung ergeben. (4) würde sich immer selber triggern.

Abhilfe schafft hier eine Liste mit Anweisungen zur Berechnung der Variablen, nach deren Abarbeitung erst die Triggerliste ausgeführt wird. Die Berechnungsliste enthält alle Anweisungen, die notwendig sind, um die Berechnung für dieses Attribut durchzuführen.

Die neue Liste ist die Berechnungsliste, und wird folgendermaßen erzeugt:

Der neue Algorithmus sieht dann so aus:

1. Füge die Formel zur Berechnungsliste der Ergebnisvariable hinzu.
2. Für alle Variablen in der Formel: Wenn Ergebnisvariable ? ?Formelvariablen, dann füge die Formel zur Triggerliste der Ergebnisvariable hinzu.
3. Berechne alle Formeln der Berechnungsliste.
4. Berechne alle Formeln neu, die in der Triggerliste für die Ergebnisvariable sind.

Damit hat ein Attribut zwei Listen, die Triggerliste, und die Berechnungsliste.

Schauen wir nun das Beispiel an:

(1) IQ = 10;

(2) M = IQ * 3;

(3) IQ = 12

(4) IQ = IQ +1;

In den Spalten bedeutet ist die erste Menge die Berechnungsliste, und die zweite Menge die Triggerliste

Variable	Nach (1)	Nach (2)	Nach (3)	Nach (4)
IQ	{1}/{}	{1}/{2}	{1,3}/{2}	{1,3,4}/{2}
M	{}/{}	{2}/{}	{2}/{}	{2}/{}
Berechnete Formeln	1 IQ=10,M=0	2 IQ=10,M=30	1,3,2 IQ=12,M=36	1,3,4,2 IQ=13,M=39

Bis jetzt werden nur Zyklen erkannt, die sich in einer Formel befinden. Zyklen über mehrerer Variablen werden nicht gehandhabt. Das läßt sich folgendermaßen lösen:

Ein Attribut bekommt eine dritte Liste, in der alle abhängigen Attribute enthalten sind. Und zwar nicht nur die Attribute die in den Formeln der Berechnungsliste sind, sondern auch deren Abhängigkeiten.

Damit alle Abhängigkeiten richtig propagiert werden, und auch Zyklen erkannt werden, sind folgende Schritte für eine neue Formel zu machen:

1. Füge die Formel zur Berechnungsliste der Ergebnisvariable hinzu.

2. Erzeuge eine Liste mit abhängigen Attribute der Formel (neue Abhängigkeiten)
3. Entferne das eigene Attribut aus der Liste der neuen Abhängigkeiten. (erlauben von lokalen Zyklen!)
4. Prüfe, ob sich das aktuelle Attribut in der Liste mit neuen Abhängigkeiten befindet: Falls ja, gibt es einen Zyklus.
5. Füge die Liste mit neuen Abhängigkeiten zu allen Attributen der Triggerliste hinzu. Dazu führe für die Attribute Schritt 4 und 5 aus.

Der Algorithmus läuft rekursiv ab. Er terminiert, da ja alle Zyklen erkannt werden.

Eine weiterer großer Vorteil bei dem Aufbau der Abhängigkeitsliste besteht darin, daß man später eine schnelle Möglichkeit des Sortierens der Berechnungen durchführen kann: Eine Attribut muß vor dem anderen berechnet werden, wenn es in der Abhängigkeitsliste des anderen enthalten ist.

Damit kann eine weitere Zyklenerkennung zur Laufzeit eingebaut werden (die allerdings niemals zuschlagen sollte!). Es gibt einen CalcHandler, der alle Berechnungen von Attributen zwischenspeichert. Werden die Berechnungen dann durchgeführt, so wird die Liste erst sortiert, und in der entsprechenden Reihenfolge ausgeführt. Triggert eine Ausführung ein weiteres Attribut so wird die Berechnung wiederum dem CalcHandler zugeführt, und wenn dieses Attribut schon berechnet wurde, so hat es einen Zyklus gegeben.

5.4.1 Handhabung von Events (ON CHANGE)

Events lassen sich sehr einfach implementieren, wenn man noch folgende Erweiterungen macht:

a) Grundsätzlich läßt sich die Berechnung optimieren, wenn man bei der Berechnung des Attributs den Wert vorher und nachher vergleicht, und wenn sich das Attribut nicht verändert hat, werden die Attribute aus der Triggerliste nicht neu berechnet.

Die Implementierung sieht dann folgendermaßen aus:

Die Formel innerhalb der ON CHANGE Anweisung bekommt den Wert an ein Pseudo-Attribut zugewiesen, und dieses Pseudo-Attribut bekommt jedes Statement innerhalb des ON CHANGE Blocks als zusätzliche Abhängigkeit.

5.4.2 Handhabung von Verzweigungen (IF)

Das Problem bei Verzweigungen ist, daß unterschiedliche Statements benutzt werden, je nach Bedingung.

Zuerst einmal wird die IF-Anweisung wie eine gewöhnliche Formel behandelt. Die Ausführung der IF-Bedingung bereitet nun folgende Probleme:

1. Möglichkeit:

In den Berechnungsbaum werden die aktuellen Anweisungen eingefügt. Ändert sich die Bedingung, so werden die alten Anweisungen entfernt, und die neuen eingefügt.

Ändert sich die Bedingung durch ein neues Kommando, so müssen die alten Anweisungen entfernt, und die neuen eingefügt werden. Da die Anweisungen nach der IF-Anweisung berechnet werden ist das kein großes Problem. Nur wann findet die Berechnung der neuen Anweisungen statt? Gleich, oder nach der Berechnung der anderen Formeln? Wenn sich die Bedingung zweimal ändert, haben wir wieder die

gleichen Formeln im Baum stehen, nur an einer anderen Position. Dadurch gibt es aber kein konsistentes Ergebnis.

2. Möglichkeit.

Die IF-Anweisung merkt sich, welche Variablen in den Zweigen benutzt werden, und trägt sich bei diesen Variablen ebenfalls in die Triggerliste ein. Die Formeln in den Zweigen selber dürfen sich dabei nicht bei den Variablen registrieren, sondern müssen über die IF-Anweisung ausgeführt werden.

Diese Variante liefert ein Konsistentes Ergebnis, da sich die Reihenfolge der Berechnung, wie es bei Möglichkeit 1 der Fall sein kann, nicht ändert. Problematisch ist hier das Abfangen der Registrierung der Attribute in den Zweigen. Wenn sich alle Anweisungen bei der IF-Anweisung in die Triggerliste eintragen, so kann es sein, daß eine Formel zu oft ausgeführt wird.

Beispiel:

```
[…]
IF (true) {
     A = A + 1;   // (1)
     B = B + 1;   // (2)
}
[…]
```

(1) **und** (2) werden ausgeführt, wenn sich A **oder** B ändert. Wenn sich als A und B ändern, werden beide Anweisungen zweimal ausgeführt, anstatt nur einmal. Die IF-Anweisung muß also schauen, für welche Variable gerade die Formel berechnet wird, und nur diese entsprechende Anweisung ausführen.

3. Möglichkeit

Jede Anweisung, bekommt ein Flag, ob sie ausgeführt werden soll, oder nicht. Ist diese Flag gesetzt, so wird die Formel berechnet, ansonsten nicht. Bei dieser Variante werden **beide** Zweige in den Formel-Baum eingetragen. Die IF-Anweisung **muß** vor den Anweisungen in den Zweigen berechnet werden, um in dem Zweig-Anweisungen das Flag zu setzten.

Diese Variante ist ebenfalls konsistent in der Berechnung, denn die Anweisungen werden nur einmal im Berechnungsbaum eingetragen, und nicht mehr verschoben. Außerdem gibt es nicht das Problem der doppelten Ausführung von Anweisungen, wie es bei Möglichkeit 2 der Fall sein kann. Ein weiterer Vorteil gegenüber 2. ist, daß die IF-Anweisung nur ausgeführt wird, wenn sich eine ihrer Variablen ändert, und nicht bei jeder Änderung eines Attributs der Anweisungen in allen Zweigen. Ein Nachteil ist natürlich, daß auf einmal Formeln mit angeschaut werden, die im Moment gar nicht benötigt werden, und evtl. weitere Berechnungen mit einfließen lassen. Da die Berechnungslisten rekursiv aufgebaut werden, werden auch die Formeln aus der Berechnungsliste von (momentan) inaktiven Formeln aufgenommen.

Zusätzlich muß die Abhängigkeit der Zweiganweisung von der IF-Anweisung berücksichtigt werden: Das kann durch ein Pseudo-Attribut geschehen.

Ein weiteres Problem hat sich an dieser Stelle ergeben: Bei geschachtelten IF-Anweisungen, gibt es Probleme, wenn sich auf einmal die äußere Bedingung ändert. Dann muß auch eine Neuberechnung der IF-Anweisungen innerhalb der beiden Zweige durchgeführt werden.

Beispiel:

```
OBJECT test
{
      NUMBER y = 0;
      NUMBER z = 0;
      NUMBER x = 0;
      IF (y = 1) {
            IF (z=2) {          // Fehler
                  SET x=1;
            } else {
            SET x=2;
      }
      } else {
            IF (y =2) {         // Fehler
                  SET x=3;
            } else {
                  SET x=4;
            }
      }
}
```

Ändert sich die Variable y auf 1, so müssen rekursiv, alle Anweisungen in dem else-Block auf unaktiv gesetzt werden, und in dem true-Block muß rekursiv alles erst einmal auf aktiv gesetzt werden. Dann muß überprüft werden, ob weitere IF-Anweisungen aktiv geworden sind, und falls ja, müssen für **beide** Zweige der Bedingung die Formeln überprüft werden.

Um dies in der Implementierung zu vereinfachen, wird es in dieser Implementierung nicht möglich sein, geschachtelte IF-Anweisungen zu benutzen.

5.4.3 Das Einfügen von Objekten

Wird ein neues Objekt zu dem Charaktergraphen hinzugefügt, so kann daß überall passieren. Die Frage ist nun, in welcher Reihenfolge die Berechnung stattfindet:

a) Werden die Formeln der Objekte in der Reihenfolge wie die Objekte in den Baum hinzugefügt werden ausgewertet, oder

b) Findet die Berechnung anhand der Position in dem Baum statt? Und wenn ja, in welcher Reihenfolge?

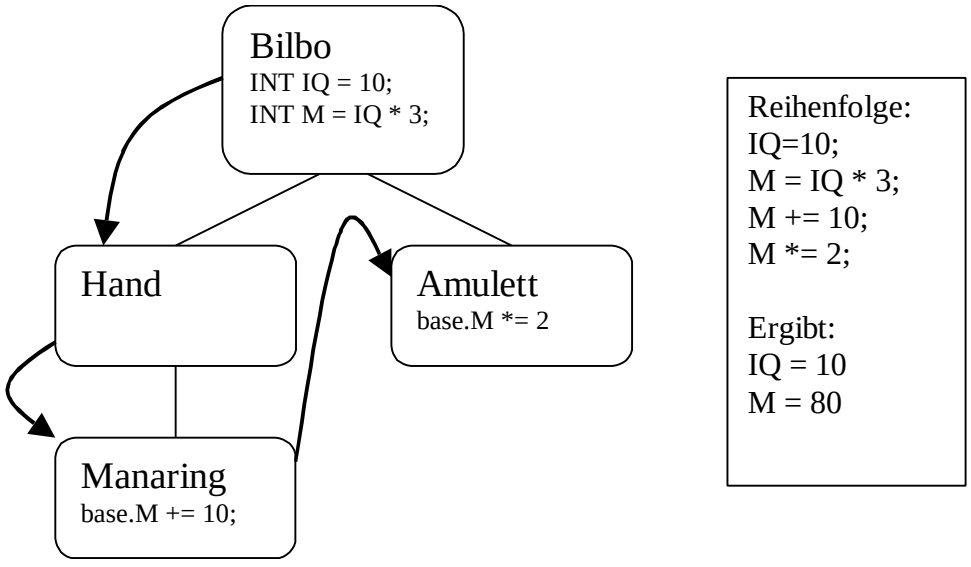


Abbildung 5.5: Tiefensuche

Für a) spricht die einfachere Handhabung in der Berechnung. Man braucht bloß die Formeln des neuen Objektes in den Graphen einzufügen. Das Problem ist, wenn ein Objekt, daß einige Schritte früher angelegt wurde, entfernt wird, und gleich wieder eingefügt wird. Dabei kann sich die Reihenfolge der berechneten Anweisungen ändern. Außerdem können dann gleiche Graphen, mit gleichen Objekten unterschiedliche Ergebnisse liefern.

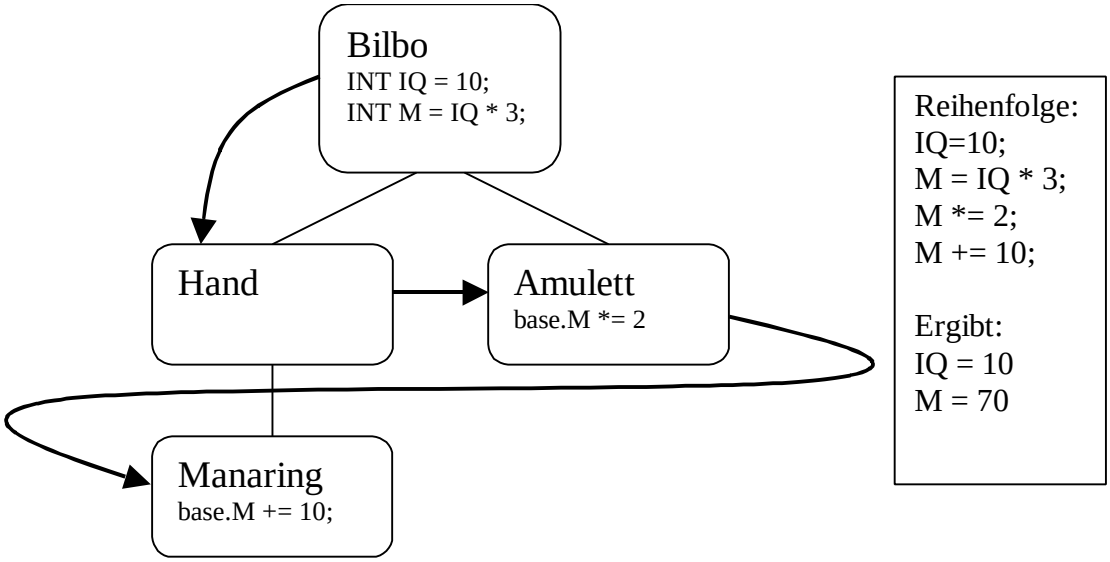


Abbildung 5.6: Breitensuche

Daher ist Variante b) zu bevorzugen: Die Auswertung der Anweisung findet flach statt, also nicht rekursiv von links nach rechts, sondern von oben nach unten. Dadurch werden die Objekte, die näher am Basisobjekt sind zuerst ausgewertet. Auf diese Weise hat jeder äquivalente Baum auch die gleiche Auswertung, und die Entstehung des Baumes spielt keine Rolle bei den Berechnungen. Man kann also auch eine komplette Berechnung mehrmals durchführen, und kommt zu dem gleichen Ergebnis.

Etwas schwieriger ist dabei das Einfügen und Löschen eines Objektes. Dabei muß berücksichtigt werden, daß in den Berechnungslisten die Anweisungen des Objektes an die richtige Stelle eingefügt werden.

Jetzt stellt sich die Frage, welcher Algorithmus der bessere ist. Tatsache ist, daß bei Rollenspielen diese Frage selten auftaucht, da die Charaktere selten so komplex aufgebaut sind, wie es hier möglich ist. Gibt es dennoch Konflikte, werden sie auf die übliche Art und Weise gelöst: Man einigt sich mit dem Spielleiter...

Da dies nicht möglich ist, bleibt nur die Möglichkeit, dem Benutzer die Wahl zu überlassen.

5.5 Implementierung der Berechnungen

Wird ein Objekt mit dem Basis Objekt verknüpft (LINK-Kommando) dann werden die Statements des Objektes "aktiv". Dieses "aktiv schalten" passiert in der Methode checkBase() von RpdlObject. Hier wird geprüft, ob das Objekt ein Vaterobjekt mit Verbindung zum Basisobjekt hat. Falls ja, wird da Objekt entsprechend initialisiert:

Aus allen Typen, aus denen das Objekt besteht, werden die zugehörigen Statements initialisiert. Dazu wird eine Liste mit StmtHandles angelegt, die eine Verknüpfung des

aktuellen Objektes mit einem Statement darstellen. Grundsätzlich werden alle Berechnungen über StmtHandles erledigt.

Wie schon in Kapitel 5.4, Algorithmus zur Berechnung der Attribute, beschrieben arbeitet der Algorithmus zur Berechnung von Attributen auf zwei Listen, die diesen zugeordnet sind: Eine Anweisung, die ein Attribut verändert, fügt sich in die CalcList hinzu, und berechnet dann das Attribut komplett neu, anhand der CalcList.

5.5.1 CalcHandler

Der CalcHandler übernimmt die Ausführung der Berechnungen. Er verwaltet eine Liste mit CalcObject Objekten, die hinzugefügt werden können. Ein CalcObject implementiert das Java Compareable Interface, und bildet so eine Ordnung. Darüber kann nun das CalcObject, das zuerst ausgeführt werden soll, herausgefunden werden.

Die Sortierung der Berechnungen für Attribute wird folgendermaßen gemacht:

Jedes Attribut hat eine Liste von Attributen, von denen es abhängt (siehe nächstes Kapitel). Es wird ein Attribut ausgesucht, daß das kleinste ist. Das kleinste wird mit dem nächsten verglichen: Ist das neue Attribut in der Abhängigkeitsliste des kleinsten Attributs, so ist das neue Attribut das kleinste.

Des weiteren enthält der CalcHandler eine Liste, in die Anweisungen kommen, die schon ausgeführt wurden. Wird ein CalcObject hinzugefügt, so wird zuerst geprüft, ob es schon in der performedList ist, und falls ja, wird eine Fehlermeldung ausgegeben. Dies hat den Sinn, daß zur Laufzeit auf Zyklen geprüft werden kann. Normalerweise sollten sie nicht auftreten, da Zyklen schon vorher erkannt werden.

Aus Performance-Gründen ist die Queue als Set implementiert, d.h. Berechnungen, die das gleiche bewirken, (zum Beispiel die Berechnung von Attribut "A" in Objekt x), werden nur einmal ausgeführt.

Die Arbeitsweise beim Abarbeiten der Liste veranschaulicht folgendes Schaubild:

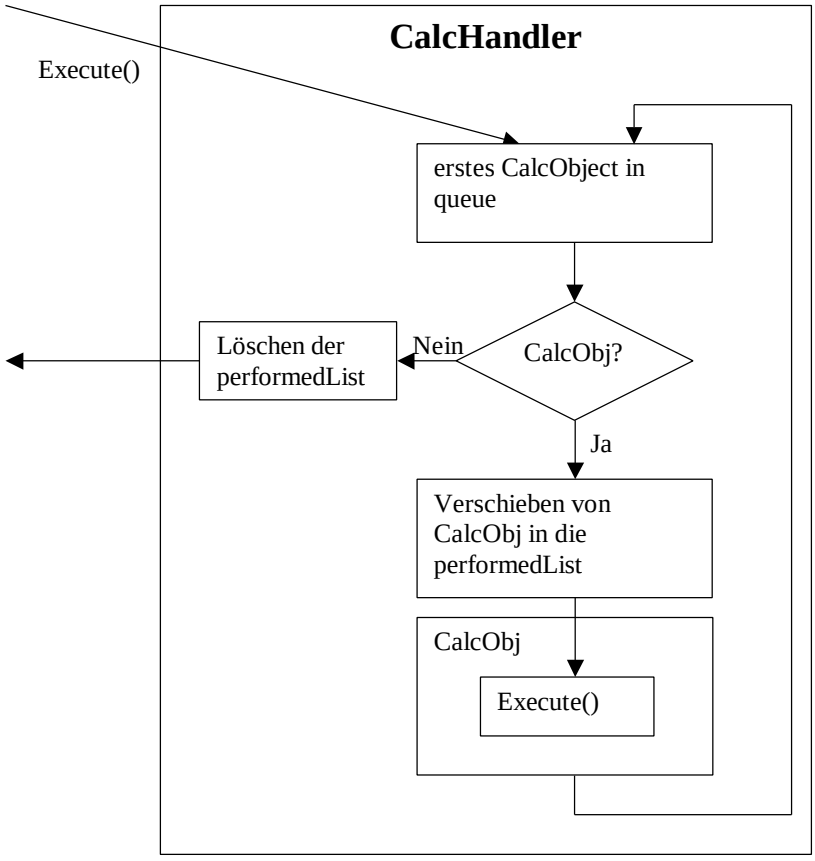


Abbildung 5.7: Arbeitsweise des CalcHandler

5.5.2 Berechnungen der Attributen

Jedes Attribut verwaltet zwei Listen: Die CalcList und die Triggerlist. Jedes besteht aus StmtHandles. Die CalcList ist dabei eine sortierte Liste, die StmtHandles in einer bestimmten Reihenfolge enthält: Die Sortierung erfolgt dabei nach folgenden Kriterien:

1. Priorität der Statements (z.B. SET vor ASSERT)
2. Position des Objektes im Baum (Ebenensuche, Tiefensuche)
3. ID des Statements (Reihenfolge beim Einlesen)

Wird ein Statement zu der CalcList hinzugefügt, so erfolgt automatisch ein Eintrag in den CalcHandler für eine Neuberechnung des Attributs. Des weiteren erfolgt die statische Überprüfung auf Zyklen während der Berechnung:

Jedes Attribut hat eine Liste mit Attributen, von denen sie abhängt. Auch die Abhängigkeiten dieser Attribute sind in dieser Liste enthalten. Diese Liste wird bei der Berechnung benötigt um das erste zu berechnende Statement zu finden.

Beispiel:

1. B=2*A
2. C=3*B

Dann hat sind die Abhängigkeiten: B_d={A}, C_d={A,B}

Nach dem Kommando

3. B=B+D

Sieht das ganze so aus:

B_d={A,D}, C_d={A,B,D}

Die Implementierung ist folgende:

1. Wird ein Statement hinzugefügt, so wird aus dem Statement die Liste der Abhängigkeiten (getDependencies()) erzeugt.
2. Von dieser Liste wird das aktuelle Attribut entfernt (lokale Zyklen sind erlaubt!).
3. Die Liste wird dem aktuellen Attribut mittels addDependencies() hinzugefügt:
4. addDependencies(): Es wird geprüft, ob das aktuelle Attribut sich in der Liste der neuen Attribute befindet.
5. Die Liste der neuen Attribute wird der Liste des Attributs hinzugefügt.
6. Die Liste der neuen Attribute wird an alle Attribute, die sich aus den Ziel Attributen der Statements der Triggerliste ergeben hinzugefügt: 4. addDependencies().

In dem Beispiel oben bei Statement (3) wäre das:

B_d={A},

C_d={A,B}

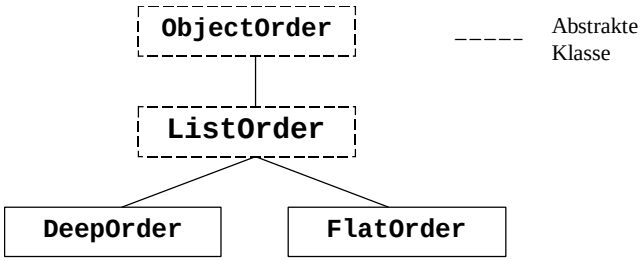
(3) B=B+D

1. Abhängigkeiten der Formel: 3_d={B,D}
2. Entfernen des eigenen Attributs: 3_d={D}
3. B.addDependencies(3_d)
4. B ? 3_d: kein Zyklus!
5. B_d=B_d ? 3_d = {A,D}
6. Liste der zu propagierenden Attribute: Statement (2) C=3*B: weitere Attribute sind {C}, also C.addDependencies(3_d):
7. C ? 3_d: kein Zyklus!
8. C_d=C_d ? 3_d = {A,B,D}
9. Triggerliste von C ist leer: fertig.

5.6 Einfügen von Objekten

Wie schon in Kapitel 5.4.3, "Das Einfügen von Objekten" beschrieben gibt es verschiedene Möglichkeiten den Baum zu traversieren. Um die verschiedenen Algorithmen (Tiefensuche, Breitensuche) in das System einzubinden, gibt es die Klasse ObjectOrder. Mit ihrer Hilfe können zwei Objekte in dem Charakter-Graphen verglichen werden. Dazu wird eine Liste mit allen Objekten des Graphen generiert, und bei dem Vergleich nur noch die Position in der Liste nachgeschaut. Die Liste wird neu aufgebaut, sobald ein neues Objekt hinzukommt.

Die zugehörigen Klassen befinden sich in dem Package de.mutschler.rpdl.tree:



Wobei ListOrder die Verwaltung der Liste usw. erledigt, während DeepOrder & FlatOrder nur noch die Erstellung der Liste implementieren:

FlatOrder:

```
public class FlatOrder extends ListOrder
{
	/** create the sequential list of objects. */
	protected void updateObjectList(List list) {
		list.clear();
		debug("creating new treelist");
		// add the starting object
		list.add(getMain().getBaseObject());
		// now scan all elements
		for(int i=0; i<list.size(); i++) {
			// get the next object
			RpdlObject ro = (RpdlObject)list.get(i);
			// we should not get any null-objects, but who knows...
			if (ro == null) continue;
			// now add all subobjects to the list
			for (Iterator it = ro.getSubObjects(); it.hasNext();) {
				Object o = it.next();
				// prevent adding duplicate items, since we have
				// and acyclic graph and not a real tree
				if (!list.contains(o))
					list.add(o);
			}
		}
	}
}
```

DeepOrder:

```
public class DeepOrder extends ListOrder
{
	/** create the sequential list of objects. */
	protected void updateObjectList(List list) {
		debug("creating new treelist");
		list.clear();
		// start with the base object
		addObject(list, getMain().getBaseObject());
	}

	private void addObject(List list, RpdlObject obj) {
		// we should not get any null-objects, but who knows...
		if (obj == null) return;
		// ignore this object, if it is already in the list
		if (list.contains(obj))	return;
		// not there, so add it
		list.add(obj);
		// now add the subobjects
		for (Iterator it = obj.getSubObjects(); it.hasNext();) {
			RpdlObject ro = (RpdlObject)it.next();
			// add it recursively (!)
			addObject(list,ro);
		}
	}
}
```

5.7 Performance

Als Grundlage für die Tests dient ein Pentium II 450Mhz unter Windows und als Virtual Machine kommt Suns JDK 1.3 & Hotspot 2 zum Einsatz.

Als Eingabe dient das File McRathgar.rpdl.

Getestet wurde das wiederholte Einlesen der Files in einer Endlosschleife, und die jeweilige Ausgabe der benötigten Zeit in Millisekunden. Damit der Garbage-Collector von Java die Ausgabe nicht allzusehr beeinflußt, wird er explizit vor jedem Durchgang aufgerufen, und Anschließend eine Pause von 20ms gemacht.

Des weiteren wurde der Scanner vorher initialisiert, da die verwendeten Tabellen komprimiert sind, und sie beim ersten Zugriff automatisch entpackt werden. Das beeinflußt die Ausführungszeit beim ersten Durchgang ziemlich stark (~10% langsamer)

Hier erfolgt die Ausgabe:

Die erste Zeit ist die Gesamt-Dauer. Die Parameter hinter "calcs:" sind:

1. Anzahl der ausgeführten Berechnungen (Neuberechnungen eines Attributs)
2. Anzahl der gesamten, auszuführenden Berechnungen. Inkl. derer, die doppelt berechnet worden wären.
3. Die Gesamtzeit, die für die Berechnung benötigt wurde.

```
Time to execute: #01: 1062ms calcs: 151/258/95ms
Time to execute: #50: 281ms calcs: 151/258/32ms
```

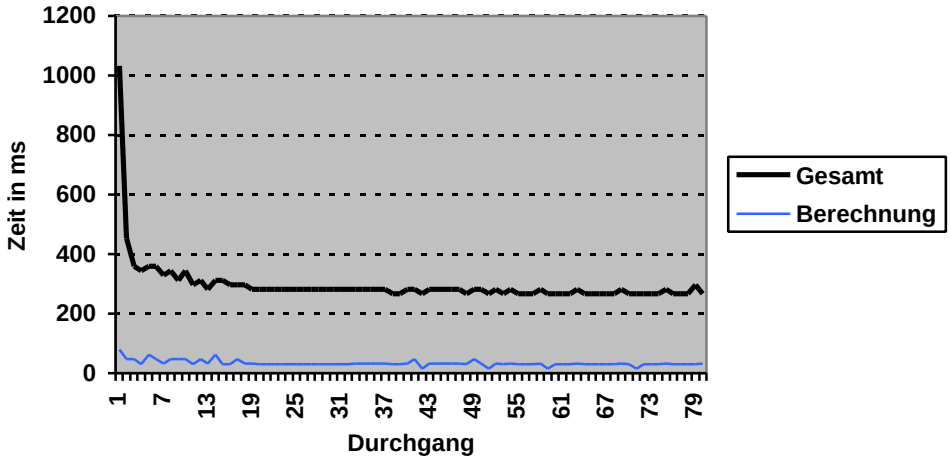


Abbildung 5.8: Laufzeitdiagramm JDK 1.3

Wie man sieht, pendelt sich die Ausführungsgeschwindigkeit bei 281ms ein. Man kann sehr gut die lokalen Optimierung von Hotspot verfolgen. Hotspot erstellt Statistiken während der Laufzeit, und optimiert die häufig benutzten Stellen im Java Bytecode. Daher wird das Programm mit jedem Durchgang schneller, bis der Punkt erreicht wird, in dem keine weiteren Optimierungen mehr möglich sind.

Für die weiteren Tests wird nun die Laufzeit des ersten und des 50. Durchgangs angegeben.

Im folgenden Diagramm sieht man die Laufzeiten der verschiedenen Virtual Machines von JDK 1.3.

1. Die Claassitz VM ("java -classic ...")
2. Die Standard VM (hotspot) ohne Just In Time Compiler ("java -Xint …")
3. Die Standard VM (hotspot) mit Just In Time Compiler ("java -hotspot …")
4. Die Server VM von Hotspot 2.0 ("java -server …")

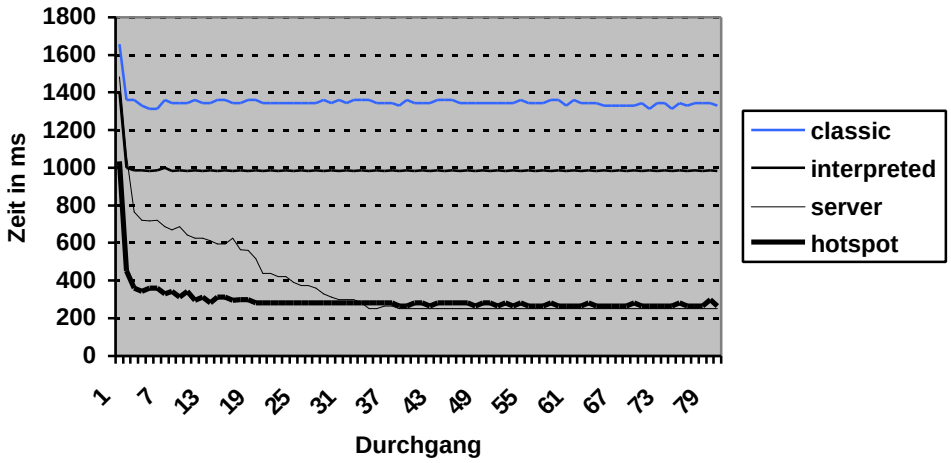


Abbildung 5.9: Laufzeitdiagramm der verschiedenen Virtual Machines

Im folgenden wird immer die Standard Einstellung benutzt, und sollte also mit der ersten Beispielangabe oben verglichen werden.

Zum Vergleich eine HP-Workstation 712 mit einem 80Mhz PA-RISC 1.1 Prozessor:

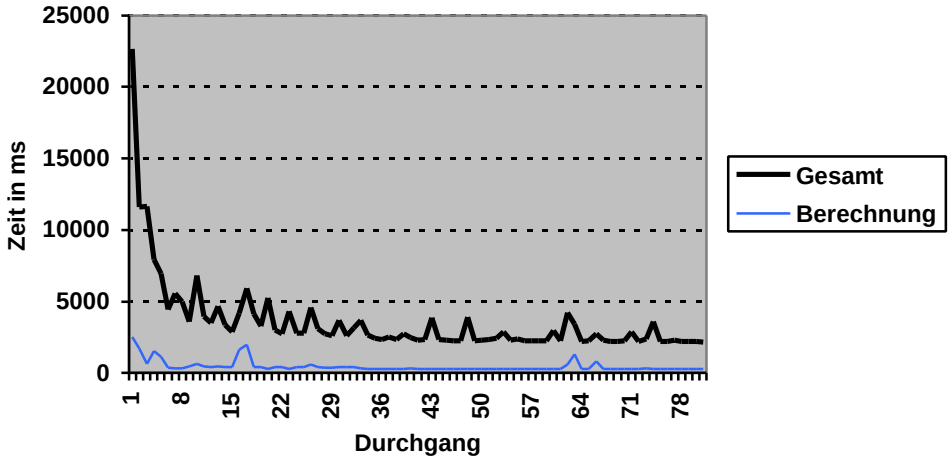


Abbildung 5.10: Laufzeitdiagramm HP 712

Wie man sieht, ist der Rechner ~ 10x langsamer als ein Pentium 450Mhz. Der Grund warum ich den Test mache, ist folgender: Dies ist mein "Server" im Internet (www.mutschler.de) auf dem die Diplomarbeit veröffentlicht wird.

5.8 Optimierungen

5.8.1 Verzögerung der Berechnungen

Eine Optimierung die eingebaut ist, ist die Möglichkeit Berechnungen während des Einlesens zu speichern, und nicht auszuführen. Außerdem werden dadurch keine

Berechnungen doppelt ausgeführt. Zum Beispiel würden folgende Kommandos dadurch das Attribut A nur ein mal Berechnen:

```
MODIFY OBJECT xy {SET A = A+1;}
MODIFY OBJECT xy {SET A = A+3;}
```

Schaltet man diese Optimierung ein, so ist das Ergebnis für das Referenz-Beispiel (McRathgar.rpdl) folgendes:

```
ERR  (main) Time to execute: #01: 1031ms calcs: 105/252/62ms
ERR  (main) Time to execute: #50: 281ms calcs: 105/252/31ms
```

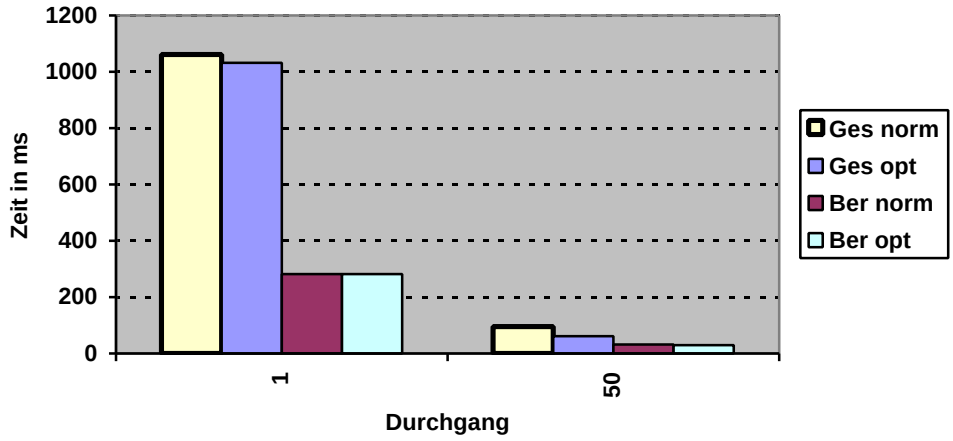


Abbildung 5.11: Laufzeitvergleich mit Berechnungsoptimierung (1)

Wie man sieht, werden insgesamt weniger Attribute neu berechnet (47 weniger, das entspricht ~30%). Die Ausführungszeit ist dagegen gleich geblieben. Das liegt daran, daß die Größe der Berechnungslisten eine Rollen spielen, und bei dem Beispielcharakter im wesentlichen neue Berechnungslisten erstellt werden, und weniger Modifikationen an Attributen.

Fügt man 10x folgende Anweisung an das Beispielfile, so sieht man, daß die Optimierung doch Vorteile hat:

```
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
```

Ohne Optimierung:

```
ERR  (main) Time to execute: #01: 1312ms calcs: 471/578/253ms
ERR  (main) Time to execute: #50: 390ms calcs: 471/578/78ms
```

Mit Optimierung:

```
ERR  (main) Time to execute: #01: 1078ms calcs: 105/272/78ms
ERR  (main) Time to execute: #50: 297ms calcs: 105/272/32ms
```

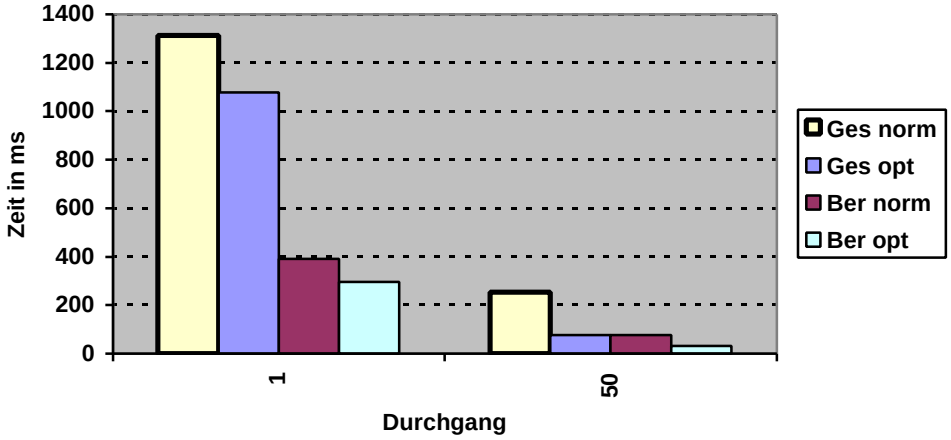


Abbildung 5.12: Laufzeitvergleich mit Berechnungsoptimierung (2)

5.8.2 Caching der Eingabedaten

Aus den Laufzeiten sieht man sehr deutlich, daß die meiste Zeit bei dem Einlesen der Daten verbraucht wird. Daher macht es Sinn, die Eingabe zu optimieren. Da die Eingabe über Kommandos erfolgt, und diese über eine einzige Methode an das eigentlich RPDL-System übergeben werden (RpdlParser.addCommand()) kann man an diesem Punkt ansetzten, und die von dem Parser gelieferten Kommandos zwischenspeichern. Es bietet sich dafür die Serialisierungsmöglichkeiten von Java an, mit denen man Objekte laden und speichern kann.

Wenn die Cache-Files erzeugt werden müssen, dauert es etwas länger:

```
ERR  (main) Time to execute: #01: 1500ms calcs: 105/252/63ms
```

Nun sind die Files erstellt. Hier das Ergebnis beim Lesen aus den Cache Files:

```
ERR  (main) Time to execute: #01: 1219ms calcs: 105/252/62ms
ERR  (main) Time to execute: #50: 500ms calcs: 105/252/32ms
```

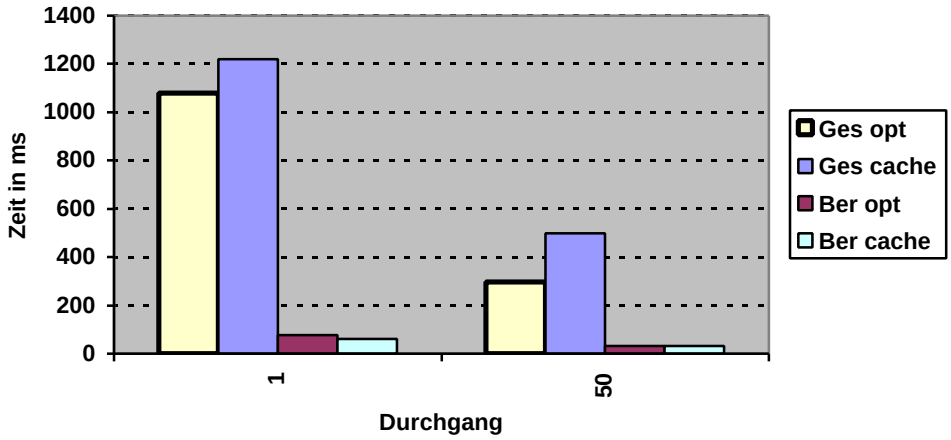


Abbildung 5.13: Laufzeitvergleich mit Caching

Das Einlesen ist dadurch nur halb so schnell wie mit Lex/Yacc (500ms? 281ms)! Diese Optimierung bringt also nichts. Deshalb habe ich sie auch wieder aus dem Code auskommentiert.

Man sieht also, daß das Einlesen mittels Lex/Yacc nach entsprechender Zeit für die VM zum Optimieren des Bytecodes, wesentlich schneller ist, als das Serialisieren von Objekten. Daher sollte man hergehen, und den Lex/Yacc Code zu optimieren, und dadurch eine bessere Performance zu bekommen. Dies ist allerdings nicht Ziel dieser Diplomarbeit.

Eine Möglichkeit zum Caching sollte jedoch in Erwägung gezogen werden:

Wenn man viele Objekte in der Objektbibliothek hat, werden die meisten für einen Charakter nicht benötigt. Man könnte es sich also sparen, die komplette Bibliothek einzulesen, sondern nur die Objekte, die mittels "CREATE OBJECT xxx FROM LIBRARY" erzeugt werden. In diesem Fall bietet sich zum Beispiel eine Datenbank an.

5.8.3 Text Ausgabe

Die Text-Ausgabe erfolgt in der Konsole, und dient in erster Linie dem Debugging im Fehlerfall. Es gibt zum einen die Meldungen über die aktuelle Aktivität, und zum Schluß eine detaillierte Beschreibung der einzelnen Objekte.

Im folgenden wird das Beispiel aus Kapitel 5.4, "Algorithmus zur Berechnung der Attribute", als Beispiel genommen. Der Quelltext sieht folgendermaßen aus:

```
CREATE ATTRIBUTE G BASICTYPE number COMMENT "Gewicht";
CREATE ATTRIBUTE GG BASICTYPE number COMMENT "Gesamtgewicht";
CREATE ATTRIBUTE IQ BASICTYPE int COMMENT "Intelligenz";
CREATE ATTRIBUTE M BASICTYPE int COMMENT "Zauberpunkte";
CREATE TYPE obj {
     ATTRIBUTE G;
     ATTRIBUTE GG;
     SET $parent.GG += G;
}

CREATE TYPE char {
     ATTRIBUTE G;
     ATTRIBUTE GG;
     SET GG = G;
     REQUIRED ATTRIBUTE IQ;
     ATTRIBUTE M;
     SET M = IQ * 3;
}

BASE char;

CREATE OBJECT Bilbo FROM char {
     SET IQ = 10;
}

CREATE OBJECT Hand FROM obj {
     SET G = 0,1;
}
CREATE OBJECT Manaring FROM obj {
     SET G = 0,1;
     SET $base.M += 10;
}

LINK Hand TO Bilbo;
LINK Manaring TO Hand;
```

Die Ausgabe ist ~22kB groß, und deshalb sind im folgenden nur Ausschnitte beschrieben.

Der Generierte Charakterbaum sieht so aus:

```
Basetype = char
Base-Object Tree:
```

```
Bilbo
+ Hand
  + Manaring
```

Das Objekt Hand sieht so aus:

```
Object #1:   ------------------------------------
OBJECT  Hand FROM obj
{
  (11) SET (Attr: G) = (const:0.1);
}

Parenttypes:
+ obj

ParentObjects:
+ Bilbo

active Statements:
Stmt #01: StmtHandle obj=Hand, stmt=(0) ATTRIBUTE G = null;
Stmt #02: StmtHandle obj=Hand, stmt=(1) ATTRIBUTE GG = null;
Stmt #03: StmtHandle obj=Hand, stmt=(2) SET (Attr: ($parent).GG) =
((Attr: ($parent).GG)  + (Attr: G));
Stmt #04: StmtHandle obj=Hand, stmt=(11) SET (Attr: G) = (const:0.1);

Attributes:
Attr #01: (NUMBER) G = '0.1'
calclist:
  #1: StmtHandle obj=Hand, stmt=(11) SET (Attr: G) = (const:0.1);
triggerlist:
  #1: StmtHandle obj=Hand, stmt=(2) SET (Attr: ($parent).GG) = ((Attr:
($parent).GG)  + (Attr: G));
Attr #02: (NUMBER) GG = '0.1'
calclist:
  #1: StmtHandle obj=Hand, stmt=(1) ATTRIBUTE GG = null;
  #2: StmtHandle obj=Manaring, stmt=(2) SET (Attr: ($parent).GG) =
((Attr: ($parent).GG)  + (Attr: G));
triggerlist:
```

Zuerst wird das eigentliche Objekt, angezeigt. Dabei werden Formeln in einer eigenen Syntax angezeigt. Sie ist aussagekräftiger, als die normale Darstellung, da hier noch zusätzliche Informationen über die Basistypen, und die Priorität dargestellt wird.

Als nächstes kommt der Baum mit Typen, von denen das Objekt abgeleitet wird, danach die Väterobjekte innerhalb des Charakterbaums.

Die nächste Liste ist die Liste der aktiven Statements, welche nicht nur die eigenen Statements enthält, sondern auch die Statements aus den Typen.

Als nächstes folgt die Liste der Attribute, und ihre Berechnungs- und Triggerlisten. Wie man bei dem Attribut GG sieht, können hier auch Statements von anderen Objekten stehen. Wie das zweite Statement der Berechnungsliste zeigt: Dieses Statement kommt von dem Manaring.

Es gibt noch Listen mit den Typen, Attribut Definitionen, der Geschichte, usw. Sie werden hier nicht im einzelnen vorgestellt, da die Ausgabe ähnlich wie die der Objekte ist.

5.8.4 HTML Ausgabe

Die zweite Möglichkeit, die Berechneten Charakterdaten auszugeben, ist eine HTML formatierte Ausgabe. Die erzeugte HTML-Seite soll dynamisch generiert werden. Da mit diesen Vorgaben eine Java-Einbindung unerläßlich ist, kommen nur noch Servlets,

als der verbreitetste Standard, in Frage. Allerdings ist die Layout-Erstellung bei Servlets nicht so einfach, und deshalb gibt es Java Server Pages (JSP) [13]. Diese basieren auf Servlets, und erlauben es, Java-Code in die Seite einzubinden.

Als JSP-Engine gibt es zum Beispiel Resin [15], bei der auch schon ein HTTP Server integriert ist.

Im nächsten Kapitel (5.8.5) ist der Beispielcode für eine JSP Seite angegeben, und in Kapitel 5.8.6 die entsprechend formatierte Ausgabe, anhand des Beispielcharakters McRathgar.rpdl.

5.8.5 HTML Ausgabe (gurps.jsp)

```
<%@ page session="false"%>

<%@ page import="de.mutschler.rpdl.*" %>
<%@ page import="de.mutschler.rpdl.entity.*" %>
<%@ page import="de.mutschler.rpdl.fileimport.*" %>
<%@ page import="de.mutschler.util.Log" %>
<%@ page import="java.util.*" %>

<%-- creation of the character --%>
<%! RpdlBase main;
      RpdlObject base;
      RpdlObject robj;      // used as the iterator over the subitems in the lists
      long execTime, calcTime;      // time for execution
%>
<%
      main = new RpdlBase();
      Log.setLogLevel(Log.DEBUG);
      Log.setLogFile("rpdl.log");
      main.getConfig().setLocale(Locale.GERMAN);

      // The file-parameter of the request specifies the file to read
      String file = request.getParameter("file");
      if (file == null) throw new IllegalArgumentException("file-parameter not
specified. Requires an absolute path!");
      FileImporter fi = new FileImporter(main, new File(file));
      //"S:\\Save\\DA\\gurps\\McRathgar.rpdl"));
      long starttime = System.currentTimeMillis();
      main.suspend(true);
      fi.scanfile();
      long midtime = System.currentTimeMillis();
      main.suspend(false);
      long endtime = System.currentTimeMillis();
      execTime = endtime - starttime;
      calcTime = endtime - midtime;
      if (main.hasError()) {
%>
<%@include file="RpdlError.jsp" %>
<%
      } else {
      base = main.getBaseObject();
%>
<html>
<head><title>GURPS Charakterbogen von "<%= base.getAttribute("Name").getStringValue()
%>"</title></head>
<body bgcolor="Silver">
<H1>GURPS® Charakterbogen von <%= base.getAttribute("Name").getStringValue() %></H1>
<p>Execution time: <%=execTime%>ms  (calculation time: <%=calcTime%>ms)</p>
<TABLE border=2><TR>
<TD valign=top align=left>
  <TABLE cellpadding=3>
  <TR>
<!-------- Basic Attributes ------------------------------------->
    <TD><big><b>ST:<BR>GE:<BR>IQ:<BR>KO:</B></BIG></TD>
    <TD align=left><big>
        <%= base.getAttribute("ST").getStringValue() %><BR>
        <%= base.getAttribute("DX").getStringValue() %><BR>
        <%= base.getAttribute("IQ").getStringValue() %><BR>
        <%= base.getAttribute("CO").getStringValue() %>
        </BIG></TD>
    <TD><b>Erschöpfung:</B><BR>
```

```
      <%= base.getAttribute("Fatigue").getStringValue() %><BR>

<!-------- Damage ------------------------------------------------------------>
      <b>Grundschaden:</B><BR>
      <b>Schw:</B> <%= base.getAttribute("SwingDamage").getStringValue() %><BR>
      <b>Stoß:</B> <%= base.getAttribute("ThrustDamage").getStringValue() %></TD>
  </TR>
  <TR valign=top>
<!-------- Movement ---------------------------------------------------------->
    <TD><b>Bewegung:</B></TD>
    <TD align=middle><b>Grundgeschw:</B><BR><%=
base.getAttribute("Speed").getStringValue() %></TD>
    <TD align=middle><b>Bewegung:</B><BR><%= base.getAttribute("Move").getStringValue()
%></TD>
  </TR>
  <TR>
<!-------- Endurance --------------------------------------------------------->

<TD><b>Belastung:<BR></B>Keine(0):<BR>Gering(1):<BR>Mittel(2):<BR>Stark(3):<BR>Extrem(4
):</TD>
    <TD><BR>
      <%= base.getAttribute("Load0").getStringValue() %><BR>
      <%= base.getAttribute("Load1").getStringValue() %><BR>
      <%= base.getAttribute("Load2").getStringValue() %><BR>
      <%= base.getAttribute("Load3").getStringValue() %><BR>
      <%= base.getAttribute("Load4").getStringValue() %>
      </TD>
<!-------- Defense ----------------------------------------------------------->
    <TD><b>Passive Verteidigung:</B><BR>
<!--          Rüstung: -<BR>
              Schild: -<BR> -->
              Gesamt: <%= base.getAttribute("PD").getStringValue() %><BR>
              <b>Schadensresistenz:</B><BR>
<!--          Rüstung: -<BR> -->
              Gesamt: <%= base.getAttribute("DR").getStringValue() %></TD>
  </TR>
  <TR><TD colSpan=3><b>Aktive Verteidigung:</B></TD></TR>
  <TR>
    <td align=center>Ausweichen:<BR><%= base.getAttribute("Dodge").getStringValue()
%><BR><small>=BW</SMALL></TD>
    <TD align=center>Parieren:<BR><%= base.getAttribute("Parry").getStringValue()
%><BR><small>=Waffe/2</SMALL></TD>
    <TD align=center>Abblocken:<BR><%= base.getAttribute("Block").getStringValue()
%><BR><small>=Schild/2</SMALL></TD>
  </TR>
  </TABLE>
</TD>
<TD>
<!-------- Advantages/Disadvantages/Quirks ---------------------------------->
  <TABLE border=1>
  <TR><TH colspan=2>Vorteile, Nachteile</TH></TR>
  <TR><TH>CP</TH><TH></TH></TR>
<%    for (Iterator it = main.getObject("AdvantageList").getSubObjects();
it.hasNext();) {
              robj = (RpdlObject)it.next();
%>
  <TR>
    <TD><%= robj.getAttribute("CPCost").getStringValue() %></TD>
    <TD><%= robj.getAttribute("Name").getStringValue() %></TD>
  </TR>
<% } %>
  </TABLE>
  <p></P>

<!-------- Skills ------------------------------------------------------------>
  <TABLE border=1>
  <TR><TD colspan=3 align=middle><b><big>Fertigkeiten:</BIG></B></TD>
  <TR><TH>CP</TH><TH></TH><TH>FW</TH></TR>
<%    for (Iterator it = main.getObject("SkillList").getSubObjects(); it.hasNext();)  {
              robj = (RpdlObject)it.next();
%>
  <TR>
    <TD align=right><%= robj.getAttribute("CPCost").getStringValue() %></TD>
    <TD align=middle title="<%= robj.getAttribute("Description").getStringValue()
%>"><%= robj.getAttribute("Name").getStringValue() %></TD>
```

```
    <TD align=right><%= robj.getAttribute("Skill").getStringValue() %></TD>
  </TR>
<% } %>
  </TABLE>
</TD></TR>
<TR><TD>

<!-------- Weapons ------------------------------------------------------->
  <TABLE border=1>

<TR><TH>Waffe</TH><TH>Stoß</TH><TH>Schwung</TH><TH>SS</TH><TH>ZG</TH><TH>1/2s</TH><TH>M
ax</TH><TH>Kosten</TH><TH>Gewicht</TH></TR>
<%    for (Iterator it = main.getObject("ItemList").getSubObjects(); it.hasNext();) {
            robj = (RpdlObject)it.next();
            if (!robj.isType("Weapon")) continue;
%>
  <TR>
  <TD title="<%= robj.getAttribute("Description").getStringValue() %>"><%=
robj.getAttribute("Name").getStringValue() %></TD>
  <TD>
  <%-- Display the thrust damage --%>
  <% if (robj.isType("ThrustDamage")) { %>
       <nobr><%= robj.getAttribute("TotalThrustDamage").getStringValue() %></nobr>
  <% } else { %>
       --
  <% } %>
  </TD>
  <TD>
  <%-- Display the swing damage --%>
  <% if (robj.isType("SwingDamage")) { %>
       <nobr><%= robj.getAttribute("TotalSwingDamage").getStringValue() %></n obr>
  <% } else { %>
       --
  <% } %>
  </TD>
  <%-- Display the attributes for ranged weapons --%>
  <% if (robj.isType("RangedWeapon")) { %>
  <TD><%= robj.getAttribute("Snapshot").getStringValue() %></TD>
  <TD><%= robj.getAttribute("RangeAim").getStr ingValue() %></TD>
  <TD><%= robj.getAttribute("HalfDmg").getStringValue() %></TD>
  <TD><%= robj.getAttribute("RangeMax").getStringValue() %></TD>
  <% } else { %>
  <TD> </TD><TD> </TD><TD> </TD><TD> </TD>
  <% } %>
  <TD>$<%= robj.getAttribute("Price").getStringValue() %></TD>
  <TD><%= robj.getAttribute("Weight").getStringValue() %>kg</TD>
  </TR>
<% } %>
  </TABLE>

<!-------- Spells -------------------------------------------------------->
   <TABLE border=1>
  <TR><TH>Spruch</TH><TH>CP</TH><TH>FW</TH><TH>Kosten</TH><TH>Dauer</TH</TR>
<%    for (Iterator it = main.getObject("MagicList").getSubObjects(); it.hasNext();) {
            robj = (RpdlObject)it.next();
%>
  <TR>
  <TD title="<%= robj.getAttribute("Description").getStringValue() %>">< %=
robj.getAttribute("Name").getStringValue() %></TD>
  <TD><%= robj.getAttribute("CPCost").getStringValue() %></TD>
  <TD><%= robj.getAttribute("Skill").getStringValue() %></TD>
  <TD><%= robj.getAttribute("SpellCost").getStringValue() %></TD>
  <TD><%= robj.getAttribute("SpellDuration").getStringValue() %></TD>
  </TR>
<% } %>
  </TABLE>

<!-------- Items --------------------------------------------------------->
  <TABLE border=1>
  <TR><TH>Gegenstand</TH><TH>Kosten</TH><TH>Gewicht</TH><TH>Beschreibung< /TH></TR>
<%    for (Iterator it = main.getObject("ItemList").getSubObjects(); it.hasNext();) {
            robj = (RpdlObject)it.next();
            if (robj.isType("Weapon")) continue; // no weapons
%>
  <TR>
```

```
  <TD><%= robj.getAttribute("Name").getStringValue() %></TD>
  <TD>$<%= robj.getAttribute("Price").getStringValue() %></TD>
  <TD><%= robj.getAttribute("Weight").getStringValue() %>kg</TD>
  <TD><%= robj.getAttribute("Description").getStringValue() %></TD>
  </TR>
<% } %>
  </TABLE>
</TD>
<TD valign=bottom>
  <TABLE border=1>
  <TR><TD></TD><TD>CP</TD></TR>
  <TR><TD>Attribute:</TD><TD><%= base.getAttribute("AttributeCP").getStringValue()
%></TD></TR>
  <TR><TD>Vorteile:</TD><TD><%=
main.getObject("AdvantageList").getAttribute("UsedCP").getStringValue() %></TD></TR>
  <TR><TD>Fertigkeiten:</TD><TD><%=
main.getObject("SkillList").getAttribute("UsedCP").getStringValue() %></TD></TR>
  <TR><TD>Magie:</TD><TD><%=
main.getObject("MagicList").getAttribute("UsedCP").getStringValue() %></TD></TR>
  <TR><TD><b>Gesamt CP:</B></TD><TD><%=
base.getAttribute("CharacterPoints").getStringValue() %></TD></TR>
  </TABLE>
</TD></TR>
</TABLE>
<!-------- History ------------------------------------------------------------>
<h2> Die Geschichte von <%= base.getAttribute("Name").getStringValue() %>< /h2>
<table border=0>
<%! RpdlHistoryEntry hist; %>
<%    for (Iterator it = main.getHistoryEntries(); it.hasNext();) {
            hist = (RpdlHistoryEntry)it.next();
%>
<tr><td colspan=2><big><b>Datum: <%= hist.getDateFormatted() %>: <%= hist.getTitle()
%></b></big></td></tr>
<tr><td width=20></td><td><%= hist.getText() %></td></tr>
<% } %>
</body></html>
<%-- end no error --%>
<% } %>
```

5.8.6 HTML Ausgabe (Beispiel)

GURPS Charakterbogen von McRathgar

Execution time: 3047ms (calculation time: 157ms)

ST: 12
GE: 13
IQ: 12
KO: 11

Erschöpfung: 12
Grundschaden:
Schw: 1W6 + 2
Stoß: 1W6 – 1

Bewegung:	Grundgeschw:	Bewegung:
	6	6

Belastung:

Keine (0):	12
Gering(1):	24
Mittel (2):	36
Stark(3):	72
Extrem(4):	120

Passive Verteidigung:
Gesamt: 0
Schadensresistenz:
Gesamt: 0

Aktive Verteidigung:

Ausweichen:	Parieren:	Abblocken:
6	0	0
=BW	=Waffe/2	=Schild/2

Vorteile, Nachteile	
CP	
10	Lesen und Schreiben
22	Magiebegabung (nur Feuer) +3
15	Kampfreflexe
-10	Rechtschaffenheit
-10	Ehrenkodex
-5	Pyromanie
-5	Leichter Schlaf
-5	Pflichtgefühl
-1	Vorsichtig
-1	Mag nichts Hochprozentiges
-1	Mag keinen Fisch essen

Fertigkeiten:

CP		FW
2	Tarnen	13
8	Breitschwert	15
0.5	Schnelladen	12
1	Schnellziehen	13

Waffe	Stoß	Schw.	SS	ZG	1/2s	Max	Kosten	Gew.
Spitzes Breitschwert	1W6 + 1	1W6 + 3					$600	1.5kg
Armbrust	1W6 + 3	--	12	4	240	300	$150	3kg

Spruch	CP	FW	Kosten	Dauer

	CP
Attribute:	80
Vorteile:	9
Fertigkeiten:	11.5
Magie:	4

Feuer entzünden	2	14	1-4	1 s
Feuer erschaffen	2	14	1-4	1 min

Gegenstand	Kosten	Gewicht	Beschreibung
Seil	$5	0.75kg	Seil der Länge 10

Gesamt CP:	104.5

Die Geschichte von McRathgar

Datum: 13.07.2000: Charakter angefangen

Charakter erstellt

Datum: 20.07.2000: Vorgeschichte

McRathgar hat seine Jugend in einem Dorf verbracht. Er hat gelernt, mit dem Schwert umzugehen, und hat eine Ausbildung als Schmid beendet. Auf dem Fest nach dem Abschluß der Ausbildung, hat er eine Gruppe mit Leuten kennengelernt, den Schamanen "Adler", einen Stummen, und einen kleinen, vorlauten Burschen. Nachdem sich alle auf der Party kennengelernt haben, hat McRathgar von seinem Vater den Auftrag bekommen, in einer Burg, die hier in der Nähe ist, einen Dudelsack zu holen. Dummerweise kommt kein Einheimischer ohne Schutz in die Burg, da es dort spukt. Deshalb sollen die Fremden McRathgar helfen.

6 Glossar

Berechnungsliste

Dies ist eine Liste mit Formeln, die jedes Attribut besitzt. In dieser Liste stehen die Formeln, die dieses Attribut beeinflussen, und zur Berechnung des Attributs ausgeführt werden müssen.

Charakter

Der Charakter ist eine fiktive Gestalt, die ein Spieler in einem Rollenspiel übernimmt. Der Spieler bestimmt die Handlungen der Person in der fiktiven Spielwelt. Ein Charakter hat bestimmte Eigenschaften die der Spieler zu berücksichtigen hat.

Geschichtsführung

Aufschreiben der Geschichte, bzw. Historie, was den Spielern in der fiktiven Welt passiert. Das kann zum Beispiel auch ein Tagebuch werden, wenn man nur aus der Sicht eines einzigen Spielers schreibt.

GURPS®

Ein Rollenspielsystem.

Harnmaster®

Ein Rollenspielsystem.

Mana

Mit Mana bezeichnet man üblicherweise die Energie die für Zaubersprüche benötigt wird.

Spielleiter

Der Spielleiter in einem Rollenspiel bestimmt was in der fiktiven Spielwelt passiert, und erklärt den Mitspielern, was ihren Charaktere geschieht. Gibt es während des Rollenspiels Diskussion, entscheidet letztendlich der Spielleiter. Er ist prinzipiell Gott in der Spielwelt.

Triggerliste

Dies ist eine Liste mit Formeln, die jedes Attribut besitzt. In dieser Liste stehen die Formeln, die dieses Attribut beeinflußt.

Zeitstempel

Angabe des Datums, wann was passiert ist. Hier wird normalerweise das Datum des Tages genommen, an dem Sich die Personen zum Rollenspielen getroffen haben.

7 Literatur

[1] Steve Jackson, GURPS® Basisbuch, Pegasus Press
ISBN 3-930635-44-1

[2] Steve Jackson, GURPS® Magie, Pegasus Press,
ISBN 3-93063-05-4

[3] AD&D Spielerset, AMIGO,
ISBN 3-933171-01-6

[4] N. Robin Crossby, Harnmaster, Columbia Games Inc.,
ISBN 0-920711-17-0

[5] Shadowrun second edition,FASA Corporation,
ISBN 1-55560-180-4

[6] Space Master - Player Book,Iron Crown Enterprise,
ISBN 1-55806-007-3

[7] M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10

[8] JAVA Homepage: http://www.java.sun.com

[9] David Flanagan, Java in a nutshell (3rd edition), O'Reilly,
ISBN 1-56592-487-8

[10] Guido Krüger, Go To Java 2, Addison-Wesley,
ISBN 3-8273-1370-8

[11] JFlex Homepage: http://www.jflex.de

[12] CUP Homepage: http://www.cs.princeton.edu/~appel/modern/java/CUP/

[13] JSP Homepage: http://www.java.sun.com/products/jsp

[14] Volker Turau, Java Server Pages, dpunkt verlag,
ISBN 3-932588-66-5

[15] Resin Homepage: http://www.caucho.com

8 Anhang

8.1 Beispiel RPDL Charakter

Der folgende Code enthält einen GURPS® Charakter in der Sprache RPDL. Das Ganze teilt sich in mehrere Dateien auf, die von dem Hauptcharakter eingebunden werden.

8.1.1 McRathgar.rpdl

```
/*
 * The actual character data. This is for the character "McRathgar"
 */

// first include all the necessary files.
INCLUDE "GURPS_base.rpdl";
INCLUDE "GURPS_equip.rpdl";
INCLUDE "GURPS_equip_weapons.rpdl";
INCLUDE "GURPS_equip_skills.rpdl";
INCLUDE "GURPS_magic.rpdl";

CREATE OBJECT McRathgar FROM Character
{
      SET Name = "McRathgar";
      SET ST = 12;
      SET DX = 13;
      SET IQ = 12;
      SET CO = 11;
      SET MaxCP = 100;
}

INCLUDE "GURPS_baseChar.rpdl";

// now, add the advantages/disadvantages to the charac ter
CREATE OBJECT AdvLiteracy FROM LIBRARY OBJECT AdvLiteracy;
LINK AdvLiteracy TO AdvantageList;

CREATE OBJECT AdvMageryFire FROM LIBRARY OBJECT AdvMageryFire;
LINK AdvMageryFire TO AdvantageList;

CREATE OBJECT AdvCombatreflexes FROM LIBRARY OBJECT AdvC ombatreflexes;
LINK AdvCombatreflexes TO AdvantageList;

// The Spells
CREATE OBJECT IgniteFire FROM LIBRARY OBJECT IgniteFire;
MODIFY OBJECT IgniteFire {
      SET CPCost=2;
}
LINK IgniteFire TO MagicList;

CREATE OBJECT CreateFire FROM LIBRARY OBJECT CreateFi re;
MODIFY OBJECT CreateFire {
      SET CPCost=2;
}
LINK CreateFire TO MagicList;

// create a sword
CREATE OBJECT TopBroadsword FROM LIBRARY OBJECT TopBroadsword;
LINK TopBroadsword TO ItemList;
```

```
CREATE OBJECT Crossbow FROM LIBRARY OBJECT Crossbow;
LINK Crossbow TO ItemList;

CREATE OBJECT SkillStealth FROM LIBRARY OBJECT SkillStealth;
LINK SkillStealth TO SkillList;
MODIFY OBJECT SkillStealth {SET CPCost+=0.5;}
CREATE OBJECT SkillBroadsword FROM LIBRARY OBJECT SkillBroadsword;
LINK SkillBroadsword TO SkillList;
MODIFY OBJECT SkillBroadsword {SET CPCost+=8;}
CREATE OBJECT SkillFastload FROM LIBRARY OBJECT SkillFastload;
LINK SkillFastload TO SkillList;
MODIFY OBJECT SkillFastload {SET CPCost+=0.5;}
CREATE OBJECT SkillFastdraw FROM LIBRARY OBJECT SkillFastdraw;
LINK SkillFastdraw TO SkillList;
MODIFY OBJECT SkillFastdraw {SET CPCost+=1;}


CREATE OBJECT Shield FROM Shield {
      SET Name = "kleines Schild";
}

LINK Shield TO ItemList;

CREATE OBJECT Rope FROM LIBRARY OBJECT Rope;
MODIFY OBJECT Rope {SET RopeLength=10;}
LINK Rope TO ItemList;

CREATE OBJECT ManaRing from Ring {
      SET Name="Ring Mana +3";
      SET $base.Fatigue += 3;
}


MODIFY OBJECT AdvMageryFire {
      SET Level = 3;
}

/*
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
MODIFY OBJECT McRathgar {SET ST =ST+1;}
MODIFY OBJECT McRathgar {SET ST =ST-1;}
*/
```

8.1.2 GURPS_base.rpdl

```
/*
 * Definitions for the Roleplaying game GURPS.
 * This is the main file you must include for GURPS
 */

CREATE ATTRIBUTE Name BASICTYPE STRING;
CREATE ATTRIBUTE CharacterPoints BASICTYPE NUMBER;
CREATE ATTRIBUTE MaxCP BASICTYPE NUMBER;
CREATE ATTRIBUTE CharCreating BASICTYPE BOOL;
CREATE ATTRIBUTE ST BASICTYPE INT;
CREATE ATTRIBUTE DX BASICTYPE INT;
CREATE ATTRIBUTE IQ BASICTYPE INT;
```

```
CREATE ATTRIBUTE CO BASICTYPE INT;
CREATE ATTRIBUTE BaseST BASICTYPE INT;
CREATE ATTRIBUTE BaseDX BASICTYPE INT;
CREATE ATTRIBUTE BaseIQ BASICTYPE INT;
CREATE ATTRIBUTE BaseCO BASICTYPE INT;
CREATE ATTRIBUTE Height BASICTYPE NUMBER;
CREATE ATTRIBUTE Weight BASICTYPE NUMBER;
CREATE ATTRIBUTE Money BASICTYPE INT;

CREATE ATTRIBUTE Fatigue BASICTYPE INT;
CREATE ATTRIBUTE ThrustDamage BASICTYPE DICE;
CREATE ATTRIBUTE SwingDamage BASICTYPE DICE;
CREATE ATTRIBUTE TakenHits BASICTYPE INT;
CREATE ATTRIBUTE Speed BASICTYPE NUMBER;
CREATE ATTRIBUTE Move BASICTYPE INT;

CREATE ATTRIBUTE PD BASICTYPE INT;
CREATE ATTRIBUTE DR BASICTYPE INT;

// attributes for the active defense
CREATE ATTRIBUTE Parry BASICTYPE INT;
CREATE ATTRIBUTE Dodge BASICTYPE INT;
CREATE ATTRIBUTE Block BASICTYPE INT;

// Belastung
CREATE ATTRIBUTE Load0 BASICTYPE INT;
CREATE ATTRIBUTE Load1 BASICTYPE INT;
CREATE ATTRIBUTE Load2 BASICTYPE INT;
CREATE ATTRIBUTE Load3 BASICTYPE INT;
CREATE ATTRIBUTE Load4 BASICTYPE INT;

// load all the tables
INCLUDE "GURPS_tables.rpdl";

// define the base-character object
CREATE TYPE Character
{
      REQUIRED ATTRIBUTE Name;
      // These are the number of CharacterPoints a Cha racter can spend
      ATTRIBUTE CharacterPoints = 0;
      ATTRIBUTE MaxCP = 100; // default is 100
//    ASSERT ERROR (CharacterPoints > MaxCP) "Character has spent too much CPs: " +
CharacterPoints;

      ATTRIBUTE CharCreating = true;       // flag, if we are creating a char acter

      // the four base attributes
      ATTRIBUTE ST = 10;     // Strength
      ATTRIBUTE DX = 10;     // Dexterity
      ATTRIBUTE IQ = 10;     // Intelligence
      ATTRIBUTE CO = 10;     // Constitution

      // Store here the values of the base attributes after creating the
      // basic character. Note: after the character has been created, the cost
      // for changing the attributes is doubled.
      //
      ATTRIBUTE BaseST;
      ATTRIBUTE BaseDX;
      ATTRIBUTE BaseIQ;
      ATTRIBUTE BaseCO;

      // Handle the cost for the attributes.
/*
      if (CharCreating) {
            SET CharacterPoints += #AttributeCost(Strength);
            SET CharacterPoints += #AttributeCost(Dexterity);
            SET CharacterPoints += #AttributeCost(Intelligence);
            SET CharacterPoints += #AttributeCost(Constitution);
      } else {
            SET       CharacterPoints       +=       #AttributeCost(Strength)*2       -
#AttributeCost(BaseST);
            SET      CharacterPoints       +=      #AttributeCost(Dexterity)*2       -
#AttributeCost(BaseDX);
            SET      CharacterPoints      +=      #AttributeCost(Intelligence)*2      -
#AttributeCost(BaseIQ);
```

```
              SET      CharacterPoints      +=      #AttributeCost(Constitution)*2      -
#AttributeCost(BaseCO);
      }
*/
      // When we finished creating the base character, i.e. The flag CharCreating
      // is changed into false, then save the current values for the base attributes
/*    ON CHANGE (CharCreating) {
              SET BaseST = ST;
              SET BaseDX = DX;
              SET BaseIQ = IQ;
              SET BaseCO = CO;
      }
*/
      // appearance:
      ATTRIBUTE Height; SET Height = #CharHeight(ST);
      ATTRIBUTE Weight; SET Weight = #CharWeight(ST);

      // The money, the character owns:
      ATTRIBUTE Money = 1000;       // at the beginning usually $1000

      // calculated values:
      ATTRIBUTE Fatigue; SET Fatigue = ST;
      ATTRIBUTE ThrustDamage; SET ThrustDamage = #CharDamageThrust(ST);
      ATTRIBUTE SwingDamage; SET SwingDamage = #CharDamageSwing(ST);
      ATTRIBUTE TakenHits = 0;
      ATTRIBUTE Speed; SET Speed = (DX + CO )/4;
      ATTRIBUTE Move; SET Move = Speed;    // = Speed rounded down

// passive defense
      ATTRIBUTE PD;  // Passive Defense (PV)
      ATTRIBUTE DR;  // Damage Resistance (SR)

// active defense
      ATTRIBUTE Dodge; SET Dodge=Move;
      ATTRIBUTE Parry;
      ATTRIBUTE Block;

// load:
      ATTRIBUTE Load0; SET Load0=ST*1;
      ATTRIBUTE Load1; SET Load1=ST*2;
      ATTRIBUTE Load2; SET Load2=ST*3;
      ATTRIBUTE Load3; SET Load3=ST*6;
      ATTRIBUTE Load4; SET Load4=ST*10;
}

// Set the Character type as the base type
BASE Character;

// create some global used basic types & attributes
CREATE ATTRIBUTE Description BASICTYPE String;
CREATE ATTRIBUTE Skill BASICTYPE INT COMMENT "Skill";
CREATE ATTRIBUTE Level BASICTYPE INT COMMENT "Level";

// all items should be derived from this type:
CREATE TYPE Object {
      REQUIRED ATTRIBUTE Name;
      ATTRIBUTE Description;
}

CREATE ATTRIBUTE UsedCP BASICTYPE NUMBER;

CREATE TYPE List {}

/* This is the global container for a list of items.
 * Here is the total number of used character points for this type of items
 */
CREATE TYPE CPList FROM List
{
      ATTRIBUTE UsedCP = 0;
      SET $base.CharacterPoints += UsedCP; //  add  the  used  characterpoints  to  the
global counter
}


CREATE ATTRIBUTE CPCost BASICTYPE NUMBER;
```

```
// every Advantage/Disadvantage sould be derived from this object , and added as item
// to the AdvantageList
CREATE TYPE CPObject FROM Object
{
      REQUIRED ATTRIBUTE CPCost;
      SET $parent.UsedCP += CPCost;
}

CREATE ATTRIBUTE Price BASICTYPE INT COMMENT "{en}Price of the item{de}Preis des
Gegenstandes";

CREATE Type Item FROM Object {
      ATTRIBUTE Price;
      ATTRIBUTE Weight;
      SET $parent.Weight += Weight;
}

/*-------------------------------------------------------------
 * Advantages/Disadvantages/Quirks
 *-------------------------------------------------------------
 * They are all stored in the AdvantageList.
 * Basically they are all handled identically.
 */
CREATE OBJECT AdvantageList FROM CPList {}
CREATE TYPE Advantage FROM CPObject {}
CREATE TYPE Disadvantage FROM CPObject {}
CREATE TYPE Quirk FROM CPObject {}

/*-------------------------------------------------------------
 * Items
 *-------------------------------------------------------------
 * Items can be put to the character. They are things like Weapons,
 * clothes, Armour, ...
 */
CREATE OBJECT ItemList FROM List {
      ATTRIBUTE Weight;
}
CREATE ATTRIBUTE TotalSwingDamage BASICTYPE DICE COMMENT "current total swing damage of
character";
CREATE TYPE SwingDamage {
      REQUIRED ATTRIBUTE SwingDamage;
      ATTRIBUTE TotalSwingDamage;
      SET TotalSwingDamage = $base.SwingDamage + SwingDamage;
}
CREATE ATTRIBUTE TotalThrustDamage BASICTYPE DICE COMMENT "current total thrust damage
of character";
CREATE TYPE ThrustDamage {
      REQUIRED ATTRIBUTE ThrustDamage;
      ATTRIBUTE TotalThrustDamage;
      SET TotalThrustDamage = $base.ThrustDamage + ThrustDamage;
}

CREATE Type Weapon FROM Item {}
CREATE Type HandWeapon FROM Weapon {}              // Nahkampfwaffen

// Attributes for ranged weapons
CREATE ATTRIBUTE Snapshot BASICTYPE INT COMMENT "Snapshot value";
CREATE ATTRIBUTE RangeAim BASICTYPE INT COMMENT "Rounds you can aim the target";
CREATE ATTRIBUTE HalfDmg BASICTYPE INT COMMENT "Half the damage";
CREATE ATTRIBUTE RangeMax BASICTYPE INT COMMENT "Maximum range";

CREATE Type RangedWeapon FROM Weapon {     // Fernkampfwaffen
      ATTRIBUTE Snapshot;
      ATTRIBUTE RangeAim;          // = ZG
      ATTRIBUTE HalfDmg;           // = 1/2 s
      ATTRIBUTE RangeMax;          // = Max
}
CREATE TYPE Shield FROM Weapon {}
CREATE TYPE Ring from Item {}


/*-------------------------------------------------------------
 * Skills
 *-------------------------------------------------------------
 * all skills are stored in the skill list.
```

```
 */
// create an object for each type of list with items which cost CPs
CREATE OBJECT SkillList FROM CPList {}
CREATE TYPE Skill FROM CPObject {
      ATTRIBUTE Skill;
      SET CPCost=0;
}
/*-------------------------------------------------------------
 * Magic
 *-------------------------------------------------------------
 * Magic spells are all stored in the MagicList.
 * For each college there exists a different type.
 */
CREATE ATTRIBUTE Magery BASICTYPE INT COMMENT "Level for magery";
CREATE OBJECT MagicList FROM CPList {
      ATTRIBUTE Magery;
}

// Attributes for spells
CREATE ATTRIBUTE SpellDuration BASICTYPE STRING;
CREATE ATTRIBUTE SpellCost BASICTYPE STRING;

// This is the base type for all spells
CREATE TYPE Spell FROM Skill {
      ATTRIBUTE SpellCost;
      ATTRIBUTE SpellDuration;
}

/*-------------------------------------------------------------
 * Basic Character design
 *-------------------------------------------------------------
 * all skills are stored in the skill list.
 */
CREATE TYPE Hand {}
CREATE OBJECT LeftHand FROM Hand {}
CREATE OBJECT RightHand FROM Hand {}
```

8.1.3 GURPS_tables.rpdl

```
/*
 * Here, all the basic tables required in GURPS are defined.
 */

// Character-points used for other attributes
TABLE INT AttributeCost(INT attr)
{
      1: -80;
      2: -70;
      3: -60;
      4: -50;
      5: -40;
      6: -30;
      7: -20;
      8: -15;
      9: -10;
      10: 0;
      11: 10;
      12: 20;
      13: 30;
      14: 45;
      15: 60;
      16: 80;
      17: 100;
      >=18: 100+(attr-17)*25;
}

TABLE NUMBER CharHeight (INT st)
{
      <=5: st*2.5+147.5;
      >5:  st*2.5+147.5;
}

TABLE NUMBER CharWeight (INT st)
{
      <=5: 65.0;
```

```
      6: 67.5;
      7: 67.5;
      8: 70.0;
      9: 72.5;
      10: 75.0;
      11: 77.5;
      12: 80.0;
      13: 82.5;
      >=14: 85+(st-14)*5.0;
}

TABLE DICE CharDamageThrust (INT st)
{
      <=4: 0W6;
      5: 1W6-5;
      6: 1W6-4;
      7,8: 1W6-3;
      9,10: 1W6-2;
      11,12: 1W6-1;
      13,14: 1W6;
      15,16: 1W6+1;
      17,18: 1W6+2;
      19,20: 2W6-1;
}

TABLE DICE CharDamageSwing (INT st)
{
      <=4: 0W6;
      5: 1W6-5;
      6: 1W6-4;
      7: 1W6-3;
      8: 1W6-2;
      9: 1W6-1;
      10: 1W6;
      11: 1W6+1;
      12: 1W6+2;
      13: 2W6-1;
      14: 2W6;
      15: 2W6+1;
      16: 2W6+2;
      17: 3W6-1;
      18: 3W6;
      19: 3W6+1;
      20: 3W6+2;
}
// ----------------------------------------
// Skill Levels

// Physical Easy
TABLE INT P_E (NUMBER cost) {
      <=0.5: $base.DX-1;
      <=1.0: $base.DX;
      <=2.0: $base.DX+1;
      <=4.0: $base.DX+2;
       >4.0: $base.DX+2 + cost/8;
}
// Physical Average
TABLE INT P_A (NUMBER cost) {
      <=0.5: $base.DX-2;
      <=1.0: $base.DX-1;
      <=2.0: $base.DX;
      <=4.0: $base.DX+1;
       >4.0: $base.DX+1 + cost/8;
}
// Physical Hard
TABLE INT P_H (NUMBER cost) {
      <=0.5: $base.DX-3;
      <=1.0: $base.DX-2;
      <=2.0: $base.DX-1;
      <=4.0: $base.DX;
       >4.0: $base.DX + cost/8;
}

// Mental Easy
TABLE INT M_E (NUMBER cost) {
```

```
      <=0.5: $base.IQ-1;
      <=1.0: $base.IQ;
       >1.0: $base.IQ + cost/2;
}
// Mental Average
TABLE INT M_A (NUMBER cost) {
      <=0.5: $base.IQ-2;
      <=1.0: $base.IQ-1;
       >1.0: $base.IQ-1 + cost/2;
}
// Mental Hard
TABLE INT M_H (NUMBER cost) {
      <=0.5: $base.IQ-3;
      <=1.0: $base.IQ-2;
       >1.0: $base.IQ-2 + cost/2;
}
// Mental Very Hard
TABLE INT M_VH (NUMBER cost) {
      <=0.5: $base.IQ-4;
      <=1.0: $base.IQ-3;
      <=2.0: $base.IQ-2;
       >2.0: $base.IQ-2 + cost/4;
}
```

8.1.4 GURPS_baseChar.rpdl

```
/*
 * This file performs the linkage of a basic character in GURPS.
 * This file should be included after the base character is set.
 */

LINK AdvantageList   TO $base;
LINK SkillList       TO $base;
LINK MagicList       TO $base;
LINK ItemList        TO $base;
LINK LeftHand        TO $base;
LINK RightHand       TO $base;
```

8.1.5 GURPS_equip.rpdl

```
/*
 * Here you can find misc equipment for GURPS.
 * They are all library objects.
 */

// Advantages / Disadvantages / Quirks

/*****************************************************
 ***** Advantages                                  ***
 *****************************************************/

CREATE LIBRARY OBJECT AdvLiteracy FROM Advantage
{
      SET CPCost = 10;
      SET Name = "{en}Literacy (Illiterate Society){de}Lesen und Schreiben";
}

TABLE INT MageryCPCost(INT level)
{
      1: 15;
      2: 25;
      3: 35;
}


CREATE LIBRARY OBJECT AdvMagery FROM Advantage
{
      SET Name = "{en}Magery{de}Magie";
      ATTRIBUTE Level = 1;
      ATTRIBUTE CPCost;
      SET CPCost = #MageryCPCost(Level);
      SET $MagicList.Magery = Level;
}

CREATE LIBRARY OBJECT AdvCombatreflexes FROM Advantage {
```

```
      SET Name = "{en}Combatreflexes{de}Kampfreflexe";
      SET CPCost=15;
//    SET $SkillFastdraw.Skill += 1;
}

CREATE ATTRIBUTE RopeLength BASICTYPE NUMBER;
CREATE LIBRARY OBJECT Rope FROM Item {
      ATTRIBUTE RopeLength;
      SET Name = "{en}Rope{de}Seil";
      SET Description="{en}Rope of the length {de}Seil der Länge " + RopeLength;
      SET Weight = RopeLength*0.075;
      SET Price = RopeLength/2;
}
```

8.1.6 GURPS_equip_skills.rpdl

```
/*
 * This file contains the skills used in GURPS. All the skills
 * are library objects.
 */

CREATE LIBRARY OBJECT SkillStealth FROM Skill {
      SET Name="{de}Tarnen{en}Stealth";
      SET Skill = #M_E(CPCost);
}

CREATE LIBRARY OBJECT SkillLockpick FROM Skill {
      SET Name="{en}Lock picking{de}Schlösser öffnen";
      SET Skill = #M_A(CPCost);
}


// Weapon skills

CREATE LIBRARY OBJECT SkillBroadsword FROM Skill {
      SET Name="{en}Broadsword{de}Breitschwert";
      SET Skill = #P_A(CPCost);
}

CREATE LIBRARY OBJECT SkillRapier FROM Skill {
      SET Name="{en}Rapier{de}Fechten";
      SET Skill = #P_A(CPCost);
}

CREATE LIBRARY OBJECT SkillBow FROM Skill {
      SET Name="{en}Bow{de}Bogen";
      SET Skill = #P_H(CPCost);
}

CREATE LIBRARY OBJECT SkillKnife FROM Skill {
      SET Name="{en}Knife{de}Messer";
      SET Skill = #P_E(CPCost);
}

CREATE LIBRARY OBJECT SkillFastload FROM Skill {
      SET Name="{en}Fastload{de}Schnelladen";
      SET Skill = #P_E(CPCost);
}

CREATE LIBRARY OBJECT SkillFastdraw FROM Skill {
      SET Name="{en}Fastdraw{de}Schnellziehen";
      SET Skill = #P_E(CPCost);
}

CREATE LIBRARY OBJECT SkillShield FROM Skill {
      SET Name="{en}Shield{de}Schild";
      SET Skill = #P_E(CPCost);
}
```

8.1.7 GURPS_equip_weapons.rpdl

```
/*
 * This file contains the weapons used in GURPS. All the weapons
 * are library objects.
 */
```

```
CREATE Type Sword FROM HandWeapon,ThrustDamage,SwingDamage {}

CREATE LIBRARY OBJECT Broadsword FROM Sword {
      REQUIRE($base.ST >= 10) "Min Strength is 10";
      SET Name="{en}Broadsword{de}Breitschwert";
      SET ThrustDamage = 1;
      SET SwingDamage = 1;
      SET Price = 500;
      SET Weight = 1.5;
}

CREATE LIBRARY OBJECT TopBroadsword FROM Sword {
      REQUIRE($base.ST >= 10) "Min Strength is 10";
      SET Name="{en}Top Broadsword{de}Spitzes Breitschwert";
      SET ThrustDamage = 2;
      SET SwingDamage = 1;
      SET Price = 600;
      SET Weight = 1.5;
}

CREATE LIBRARY OBJECT Crossbow FROM RangedWeapon,ThrustDamage {
      REQUIRE($base.ST >= 7) "Min Strength is 7";
      SET Name="{en}Crossbow{de}Armbrust";
      SET ThrustDamage = 4;
      SET Price = 150;
      SET Weight = 3;
      SET Snapshot = 12;
      SET RangeAim = 4;
      SET HalfDmg = $base.ST * 20;
      SET RangeMax = $base.ST * 25;
}
```

8.1.8 GURPS_magic.rpdl

```
/*
 * This file contains the magic spells used in GURPS. All the spells
 * are library objects.
 */

CREATE ATTRIBUTE FireMagery BASICTYPE INT;
CREATE TYPE MageryFireList {
      ATTRIBUTE FireMagery;
}
MODIFY OBJECT MagicList {
      ADD TYPE MageryFireList;
}

CREATE TYPE FireSpell FROM Spell {
      SET Skill += $MagicList.FireMagery;
}

TABLE INT MageryCollegeCPCost(INT level)
{
      1: 12;
      2: 17;
      3: 22;
}

CREATE LIBRARY OBJECT AdvMageryFire FROM Advantage
{
      ATTRIBUTE Level = 1;
      SET Name = "{en}Magery (Fire College Only) +{de}Magiebegabung (nur Feuer) +"
+Level;
      SET CPCost = #MageryCollegeCPCost(Level);
      SET $MagicList.FireMagery = Level;
}

/************************************************
 *** Spells
 ************************************************/

CREATE LIBRARY OBJECT IgniteFire FROM FireSpell {
      SET Skill += #M_H(CPCost);
      SET Name="{en}Ignite Fire{de}Feuer entzünden";
```

```
      SET Description="{de}1=Streichholz\n2=Fackel\n3=Schweißbrenner\n4=Magnesium";
      SET SpellCost="1-4";
      SET SpellDuration="1 s";
}

CREATE LIBRARY OBJECT CreateFire FROM FireSpell {
      SET Skill += #M_H(CPCost);
      SET Name="{en}Create Fire{de}Feuer erschaffen";
      SET SpellCost="1-4";
      SET SpellDuration="1 min";
      REQUIRE OBJECT ($IgniteFire) "Ignite Fire is required to get the Spell " + Name;
}
```