

Universität Stuttgart

Fakultät Informatik

Prüfer: Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer: Thomas Eisenbarth
Beginn am: 15.3.2000
Beendet am: 15.9.2000
CR-Klassifikation: D.3.2

Diplomarbeit Nr. 1848

Entwicklung einer Roleplaying Definition Language (RPDL)

Michael Mutschler

Institut für Informatik
Universität Stuttgart
Breitwiesenstraße 20-22
D-70565 Stuttgart

Zusammenfassung

Dieses Dokument beschreibt die Entwicklung einer Programmiersprache für Rollenspiele. Dabei sollen die Gemeinsamkeiten der verschiedenen Rollenspiele zusammengefaßt werden, und mittels entsprechender Kommandos eingegeben werden können. Letztendlich soll die Charaktererstellung erleichtert werden, und soweit wie möglich die verschiedenen Abhängigkeiten bei der Erstellung berücksichtigt werden.

Grundlage für die Sprache sind keine Computer-Rollenspiele, sondern die ursprüngliche Version mit Würfeln, Papier und Bleistift.

Diese Dokumentation ist in 3 Abschnitte gegliedert. Abschnitt 1 beschreibt die Aufgabenstellung, und die Hintergründe von Rollenspielen. Abschnitt 2 beinhaltet die eigentliche Spezifikation der Sprache, und in Abschnitt 3 wird die Implementierung in Java erläutert.

Die URL der Homepage, auf der der Quellcode, die Ausarbeitung, usw. zu finden ist, lautet:

<http://www.mutschler.de/rpdl>

1 Inhaltsverzeichnis

| | | |
|--------|---|----|
| 1 | Inhaltsverzeichnis | 1 |
| 2 | Aufgabenstellung..... | 3 |
| 2.1 | Was Rollenspiele sind..... | 3 |
| 2.1.1 | Wie funktionieren Rollenspiele?..... | 3 |
| 2.1.2 | Ablauf eines Rollenspieles | 4 |
| 2.2 | Aufgabe der Diplomarbeit..... | 5 |
| 2.3 | Anforderungen an die Programmiersprache | 5 |
| 3 | Spezifikation..... | 7 |
| 3.1 | Design der Sprache | 7 |
| 3.2 | Allgemeines..... | 9 |
| 3.3 | Geschichtsführung | 9 |
| 3.4 | Basistypen | 11 |
| 3.4.1 | Würfelwert..... | 12 |
| 3.4.2 | Gültige Operationen der Basistypen | 13 |
| 3.5 | Tabellen..... | 14 |
| 3.6 | Aufbau des Charakters..... | 16 |
| 3.7 | Formeln | 19 |
| 3.7.1 | Spezialfunktionen..... | 21 |
| 3.7.2 | count/countall..... | 21 |
| 3.7.3 | countparents/countallparents | 22 |
| 3.7.4 | istype | 22 |
| 3.8 | Attribute | 22 |
| 3.9 | Typen | 23 |
| 3.10 | Objekte | 24 |
| 3.11 | Durchführen der Berechnungen..... | 26 |
| 3.12 | Anweisungen | 28 |
| 3.12.1 | Attributdeklarationen..... | 29 |
| 3.12.2 | Zuweisungen | 30 |
| 3.12.3 | Zuweisungen mit „parentall“ | 30 |
| 3.12.4 | Zuweisungen mit Prioritäten..... | 32 |
| 3.12.5 | Bedingungen | 33 |
| 3.12.6 | Zusagen..... | 33 |
| 3.12.7 | Vorbedingungen..... | 34 |
| 3.12.8 | Events | 35 |
| 3.12.9 | Zusammenfassung der Prioritäten..... | 35 |
| 3.13 | Charaktergrapherstellung | 35 |
| 3.14 | Modifizieren des Charakters..... | 38 |
| 3.15 | Objektbibliothek | 39 |
| 4 | Implementierung | 40 |
| 4.1 | Allgemein..... | 40 |

| | | |
|-------|---|----|
| 4.2 | Einlesen | 40 |
| 4.2.1 | Statische Analyse beim Einlesen | 42 |
| 4.3 | Anweisungen | 42 |
| 4.4 | Algorithmus zur Berechnung der Attribute..... | 43 |
| 4.4.1 | Handhabung von Events (ON CHANGE)..... | 47 |
| 4.4.2 | Handhabung von Verzweigungen (IF) | 47 |
| 4.5 | Implementierung der Berechnungen..... | 49 |
| 4.5.1 | CalcHandler | 50 |
| 4.5.2 | Berechnung der Attributen | 51 |
| 4.6 | Einfügen von Objekten | 53 |
| 4.7 | Performance..... | 54 |
| 4.8 | Optimierungen | 56 |
| 4.8.1 | Verzögerung der Berechnungen | 56 |
| 4.8.2 | Caching der Eingabedaten | 57 |
| 4.8.3 | Text Ausgabe | 58 |
| 4.8.4 | HTML Ausgabe | 60 |
| 4.8.5 | HTML Ausgabe (Beispiel) | 60 |
| 5 | Glossar | 62 |
| 6 | Literatur..... | 64 |
| 7 | Anhang | 65 |
| 7.1 | Beispiel eines Charakterbogens..... | 65 |
| 7.2 | Beispiel RPDL Charakter..... | 66 |
| 7.2.1 | McRathgar.rpdl | 66 |
| 7.2.2 | GURPS_base.rpdl | 67 |
| 7.2.3 | GURPS_tables.rpdl | 71 |
| 7.2.4 | GURPS_baseChar.rpdl..... | 73 |
| 7.2.5 | GURPS_equip.rpdl..... | 73 |
| 7.2.6 | GURPS_equip_skills.rpdl | 74 |
| 7.2.7 | GURPS_equip_weapons.rpdl | 74 |
| 7.2.8 | GURPS_magic.rpdl..... | 75 |
| 7.3 | Erklärung..... | 76 |

2 Aufgabenstellung

Dieses Kapitel enthält die Aufgabenstellung und die eigentliche Motivation für die Idee von RPD (Roleplaying Definition Language).

2.1 Was Rollenspiele sind

Rollenspiele sind in den 70er Jahren in den USA entstanden. Damals wurde die Idee geboren, 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 und von Vampiren handeln.

Das erste Rollenspiel, *Dungeon & Dragons*, ist ein Fantasy-Rollenspiel, das im Mittelalter spielt, und viele Monster hat, wie Drachen, Trolle, etc. hat. Mittlerweile gibt es die erweiterte Variante, *Advanced Dungeon & Dragons* [3].

2.1.1 Wie funktionieren Rollenspiele?

Bei einem Rollenspiel gibt es üblicherweise 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 komplexen Hintergrund und ist in den Regelwerken genau beschrieben. Ein Buch, das so eine Welt gut beschreibt, ist „Der Herr der Ringe“ von J. R. R. Tolkien [4]. Die Geschichte handelt in einer mittelalterlichen Welt. Häufig werden die dort vorkommenden Fantasiegestalten, wie Elfen und Trolle, in die verschiedenen Rollenspielwelten übernommen. In Tolkiens Welt gibt es den Kontinent Mittelerde, bei dem im Nordosten die Elfen wohnen, im Norden in den Bergen die Zwerge und im Süden das Böse (Trolle, Orks, usw.). In dem Rollenspiel *Midgard*® [8] gibt es analog den Kontinent *Midgard*, auf dem es verschiedene Städte, Landschaften und Bewohner gibt, in denen die Abenteuer stattfinden.

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, erstellt sich jeder Spieler eine fiktive Person, den Charakter, mit dem er in Zukunft in dieser Welt agiert. Meist werden vom Spieler selbst die Charakterstärken bestimmt. Beliebte Figuren sind zum Beispiel Zauberer, Kämpfer und Diebe. Man kann aber auch ein Allround-Talent spielen, das 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 wie Stärke, Intelligenz, Konstitution und Geschicklichkeit, bei anderen gibt es dann zusätzlich noch Ausdauer, Geschwindigkeit, Geruchssinn, Hörsinn, Sicht, usw.

Nachdem jetzt bekannt ist, was der Charakter können soll, bestimmt man die Grundwerte. Das kann einerseits durch Würfeln passieren (wie in *Midgard*), oder man

verteilt die anfänglichen Charakterpunkte (CP). Beispielsweise können bei GURPS® [1] 100 CPs auf die Attribute und Fertigkeiten verteilt werden. Bei *Midgard* geschieht das durch zweimaliges würfeln mit einem 100-Seiten-Würfel, und das höhere Ergebnis zählt. Manche Spielleiter erlauben es, danach die gewürfelten Werte auf die Grundwerte selber zu verteilen. Damit kann man seiner gewünschten Spielerrolle besser gerecht werden. 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.

Der nächste Schritt ist die Auswahl der Fertigkeiten (Skills), die der Charakter hat. Einen Charakter spielt man normalerweise nicht als Säugling, sondern als eine Person mit einer Ausbildung. Die einzelnen Fertigkeiten und ihre Kosten, stehen wiederum im Regelwerk. Diese Fertigkeiten können im Lauf des Spieles gesteigert werden, wenn der Charakter an Erfahrung reicher wird, und dadurch Charakterpunkte sammelt. Je besser ein Charakter eine Fertigkeit beherrscht, 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 überlebensfähig sein und nicht in der ersten schwierigen Situation versagen.

Als letztes kommt noch die Grundausrüstung für den Charakter (z. B. Kleidung). Der Kämpfer wird schon ein paar Waffen und eine Rüstung haben, der Dieb eventuell ein paar Dietriche, usw.

2.1.2 Ablauf eines Rollenspieles

Nachdem die Spieler jeweils einen Charakter erstellt haben, kann es losgehen. Der Spielleiter muß zuerst 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, usw. 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, genauer nachzuforschen, zu der entsprechenden Adresse zu gehen und den Auftrag anzunehmen. 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 anschließend das Schloß knacken. Ein Kämpfer kann seine Waffe nehmen und aus der Tür Kleinholz machen; der Magier könnte einen Feuerspruch zaubern und die Tür verbrennen. Wer genau was macht, machen die Spieler unter sich aus. Sind sie sich einig geworden, sagen sie dem Spielleiter, wer was unternimmt. Zum Beispiel sagt der Spieler des Diebes zum Spielleiter, daß er zuerst nach Fallen suchen möchte. Daraufhin muß dieser Spieler würfeln, ob er erfolgreich ist oder nicht. Abhängig davon wie gut er Fallen finden kann, wird er sie entdecken, oder nicht. Normalerweise würfelt der Spielleiter verdeckt, und teilt dem Spieler das Ergebnis mit: Hat er schlecht gewürfelt, wird der Dieb keine Falle finden selbst wenn eine vorhanden ist. Hat er gut gewürfelt, wird er eine vorhandene Falle finden. Der Spieler kann sich also nicht darauf verlassen, daß keine Falle vorhanden ist, wenn der Spielleiter sagt, der Charakter findet keine.

Kommt es zu einem Kampf, weil zum Beispiel ein paar Diebe das Lager überfallen, so kommen die Kampf-Regeln ins Spiel. Diese sind stellenweise sehr komplex, so daß es schon einmal 15 Minuten dauern kann, bis eine Runde im Kampf beendet ist, und alle beteiligten Charaktere ihre einzelnen Aktionen ausgeführt 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 lange dauert, sondern flüssiger abläuft.

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

2.2 Aufgabe der Diplomarbeit

Die Erstellung des Charakters ist immer eine langwierige Aufgabe, und kann Stunden dauern. Insbesondere die Anwendung der vielen Regeln sowie das Nachschlagen in den Tabellen ist sehr zeitraubend. Oft kommen auch Formeln vor, die ausgerechnet werden müssen (z. B. $\text{Geschwindigkeit} = \text{Geschicklichkeit} + \text{Stärke} / 4$). Des Weiteren hängen die Fertigkeiten von den Grundwerten ab, und oft muß in Tabellen nachgeschaut werden, welcher Wert sich letztendlich ergibt. 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 diese Regeln eingeben werden können. Weiterhin sollen noch folgende Möglichkeiten bestehen:

- Laden von Bibliotheken, aus denen Objekte (Ausrüstungsgegenstände, Fertigkeiten, Zaubersprüche, etc.) ausgewählt werden können, die man dem neuen Charakter zuweist.
- Festhalten der Charakterentwicklung. Damit besteht die Möglichkeit, den Lebenslauf des Charakters nachzuvollziehen.

2.3 Anforderungen an die Programmiersprache

Aufgrund der Analyse von einigen Rollenspielregelwerken, sind die Anforderung an eine Programmiersprache folgende:

- Benutzung von Attributen. Attribute müssen Werte der folgenden Arten darstellen können: Zahlen, Zeichenketten und Würfel.
- Die Eingabe von Formeln für die Berechnung der Werte der Attribute. Außerdem muß die Möglichkeit bestehen, andere Attribute wie Variablen in Formeln zu benutzen.
- Die Möglichkeit der Eingabe von Tabellen und ihre Verwendung in Formeln.
- Hinzufügen von Objekten zu dem Charakter. 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.
- Das Rollenspiel *Rolemaster*® hat die Regel, daß pro Stufe, die der Charakter erreicht, eine Fähigkeit nur um maximal 2 Stufen verbessert werden kann. Dieses Verhalten soll ebenfalls realisiert werden können.
- 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.
- Für eine spätere Erweiterung, soll es die Möglichkeit eines „Wizards“ geben, der schrittweise die benötigten Werte abfragt.

3 Spezifikation

3.1 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. Wichtig ist, daß alles richtig berechnet wird, und alle Formeln richtig angewendet werden. Weiterhin ist es wünschenswert, wenn das Programm unter verschiedenen Betriebssystemen, wie Windows und diverse Unix Varianten (Linux, Solaris, HPUX, usw.), läuft. Dementsprechend wird die Implementierung auch in Java [10] stattfinden, auf welche diese Anforderungen am besten zutreffen.

Die Sprache ist eine Batchsprache: 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 definierten 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 am Anfang übersehen wurde und zu dem Charakter besser paßt. Auch dies soll prinzipiell möglich sein.

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

Die Darstellung des Charakters erfolgt über eine Baumstruktur. Wie detailliert der Baum letztendlich modelliert wird, hängt einerseits vom Regelwerk des jeweiligen Rollenspieles ab, andererseits davon, wie umfangreich die Gegenstände erstellt werden. Zum Beispiel kann ein Rucksack ein linkes und rechtes Fach enthalten, oder nur als großer Behälter angesehen werden. Siehe Abbildung 3.1.

Legende zu den Diagrammen:

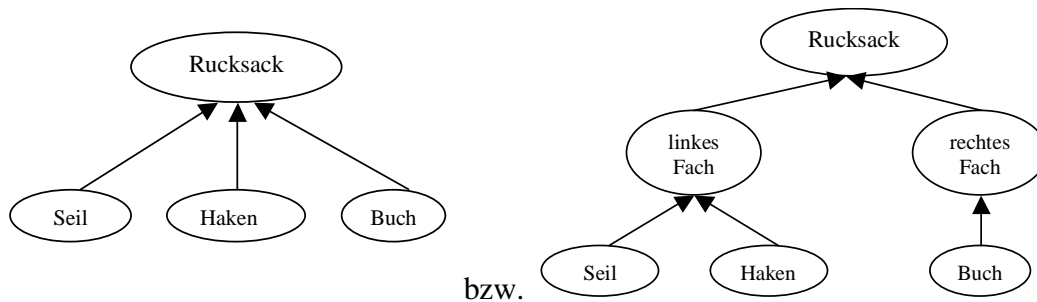
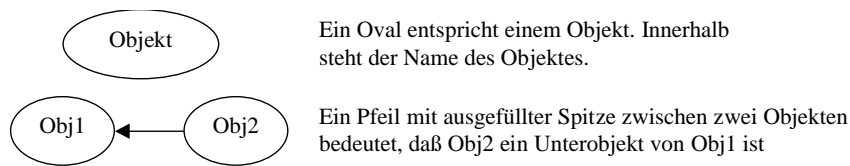


Abbildung 3.1: Baumstruktur des Charakters

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.

Eine weitere wichtiger Aspekt bei einem Charaktereditor ist die Verwendung von fertigen Objekten. So soll es möglich sein, daß es eine Bibliothek mit fertigen Gegenständen gibt, die nur noch dem Charakter zugewiesen werden müssen. Dies ist sinnvoll für die Standardausrüstung, wie Kleidung, Rüstung, Waffen, usw. Aber auch Dinge wie Fertigkeiten und Zaubersprüche sollen vorher eingegeben und bei verschiedenen Charakteren benutzt werden können.

Die Programmiersprache ist damit in drei Teile aufzuteilen:

- Beschreibung der Rollenspielregeln.
- Erstellen von Objektbibliotheken.
- Die eigentlichen (individuellen) Charakterdaten.

Die Regeln werden für das jeweilige Rollenspielsystem geladen. Die Rollenspielregeln sind für jeden Charakter innerhalb dieses Rollenspielsystems die gleichen. Sie müssen nicht unbedingt in einer einzigen Datei stehen, sondern können auch auf mehrere verteilt werden. So gibt es auch bei den einzelnen Rollenspielen nicht grundsätzlich ein Regelwerk, sondern auch Regelerweiterungen. Wie zum Beispiel das erweiterte Kampfsystem von *GURPS®*, das Treffer nicht nur allgemein auf den Körper zuläßt, sondern Unterschiede zwischen Kopf, Armen und Beinen macht.

Die Objektbibliothek basiert auf den Objekten, die in den Regelwerken beschrieben sind, und stellt eine Vereinfachung der Eingabe von gleichen Objekten dar. Jedes Rollenspiel hat eine Liste mit Standardausrüstungsgegenständen und ihren Eigenschaften. So hat zum Beispiel ein normales Schwert einen Preis, dem Schaden, den es verursacht, ein Gewicht, usw.

Der letzte Teil ist der individuelle Charakter. Dieser bestimmt 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.

3.2 Allgemeines

Das Einlesen der Kommandos geschieht über Quelldateien. Damit mehrere Dateien angegeben werden können, gibt es ein Include-Kommando, das eine weitere Datei an der aktuellen Position einliest. Eine Datei mit den eigentlichen Charakter Kommandos kann so am Anfang weitere Dateien einlesen, in der die eigentlichen Beschreibungen des Rollenspiels stehen. Es besteht dadurch die Möglichkeit, separate Dateien zu benutzen, in denen die unterschiedlichen Bibliotheksobjekte beschrieben sind, wie beispielsweise eine Datei mit Magiesprüchen, eine mit den verschiedenen Fertigkeiten, etc.

Syntax des Include-Kommandos in EBNF:

```
cmdInclude := 'INCLUDE' string ';' ;
```

Beispiel:

```
INCLUDE "GURPS_base.rpd1" ;
```

Der String, der in dem Kommando angegeben ist, beschreibt den Pfad zu der einzufügenden Datei. Der Pfad kann auch relativ sein, wobei er sich dann relativ zu dem Pfad der aktuellen Datei bezieht.

Kommentare können in den Dateien ebenfalls benutzt werden. Sie entsprechen der Java Syntax. Daher gibt es zwei Varianten sie einzugeben:

- a) Der Kommentar beginnt mit „/*“ und endet mit „*/“.
- b) Der Kommentar beginnt mit „/*“ und endet am Ende der aktuellen Zeile.

Geschachtelte Kommentare sind nicht möglich.

Die erzeugten Elemente in der Sprache werden über Identifier angesprochen. Die Identifier sind abhängig von der Groß- und Kleinschreibung, im Gegensatz zu den verwendeten Schlüsselwörtern. Diese können groß oder klein geschrieben werden. Ein Identifier fängt immer mit einem Buchstaben oder Unterstrich an, und besteht danach aus einer beliebigen Folge von Buchstaben, Zahlen und dem Unterstrich. Umlaute sind nicht zulässig.

Hier die Syntax des Identifier:

```
identifier := [a-zA-Z_][a-zA-Z0-9_]*
```

3.3 Geschichtsführung

Die gesamte Erstellung des Charakters besteht aus einer Reihe von Kommandos, die nacheinander ausgeführt werden.

Das hat den Vorteil, daß zu jedem Kommando ein Geschichtseintrag angegeben werden kann. Dieser Geschichtseintrag enthält einen Zeitstempel, optional einen Titel und die eigentliche Notiz.

Alle Geschichtseinträge zusammen ergeben die komplette Geschichte des Charakters. Dies kann aus der Sicht des Charakters geschehen, und ein Tagebuch ergeben, oder aus der Sicht der Rollenspielergruppe. Dort werden nach jedem Rollenspiel-Tag die Ereignisse und Veränderungen festgehalten werden, die der Charakter durchgemacht hat.

Aus implementierungstechnischen Gründen wird sich die Abarbeitung der Kommandos nicht an den Zeitstempeln orientieren, sondern anhand der Position in der Quelldatei. Deshalb muß bei der Erstellung des Charakters auf die Reihenfolge geachtet werden. Der Zeitstempel ist rein informativ und hat auf die eigentliche Erstellung des Charakters keine Auswirkungen.

Es bieten sich folgende Möglichkeiten der Realisierung an:

1. Der Geschichteseintrag ist in jedem Kommando enthalten.
2. Der Geschichteseintrag bildet ein eigenes Kommando.

Vor- und Nachteile der beiden Varianten:

Variante 1:

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

Dies könnte ungefähr so aussehen:

```
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 Geschichteseintragkommando im Quellcode, und alle Kommandos, die danach kommen, beziehen sich auf diesen Eintrag.

Es macht keinen Sinn, die Kommandos, wie zum Beispiel das Kaufen von Waren, detailliert zu kommentieren. In diesem Falle genügt die Notiz „Bilbo hat in Kaufhausen eine Winterausrüstung gekauft“, für alle folgenden Kommandos. Bei Variante 1 würde in jedem Kommando die gleiche Notiz stehen.

Ein Beispiel:

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

Der Geschichteseintrag soll nur in groben Zügen darüber informieren, was mit dem Charakter geschehen ist. Die Details über die Veränderung der Charakters sind normalerweise aus dem Quellcode ersichtlich. Deshalb ist Variante 2 besser geeignet, und wird auch in RPD

Syntax für einen Geschichteseintrag:

```
histEntry := '[' DATE string ';' ['TITLE' string ';' ]
'CONTENT' string '']
```

Beispiel:

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

Zuerst wird das Datum als String angegeben. Das Datum selbst hat keine spezifische Syntax, da es nur rein informell ist.

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

Zuletzt erfolgt die eigentliche Notiz, die mit dem Schlüsselwort „CONTENT“ eingeleitet wird.

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

Syntax für die erweiterte Eingabe des Geschichtseintrags:

```
histEntryLong := '[' 'DATE' string ';' ['TITLE' string ';']
               'CONTENT:START' ']'
               string
               '[' 'CONTENT:END' ']'
```

Beispiel:

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

Anmerkung: Der String in dem Eintrag behält seine Zeilenumbrüche bei! Der Text wird in diesem Fall nicht weiter geparkt, und Sonderzeichen werden direkt übernommen. Der Eintrag wird durch ein „[CONTENT:END]“ am Anfang einer neuen Zeile beendet.

3.4 Basistypen

Die Basistypen sind die verschiedenen Arten von Attributen, die in RPD verwendet werden können. In den Basistypen werden die eigentlichen Informationen gespeichert. Es gibt folgende 5 Arten:

NUMBER

NUMBER bedeutet eine Fließkommazahl. Mit NUMBER Attributen kann man die üblichen Rechenoperationen durchführen. Die Exponentialschreibweise ist nicht erlaubt.

INTEGER (oder auch INT)

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

DICE

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

BOOL

Ein boolesches Attribut. Ein BOOL 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 auch ein Anführungszeichen in dem String angegeben werden kann, muß vor dem Anführungszeichen ein Backslash „\“ stehen. Z. B.: „ein \"text\"“. Des Weiteren gibt es noch zusätzliche Sonderzeichen, für die der Backslash benutzt werden muß.

Ein zusätzliches Feature bei einem String besteht darin, 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 nach ISO-639 [11], wie er auch von Java benutzt wird, in geschweiften Klammern angegeben, danach folgt der eigentliche Text in der entsprechenden Sprache. Ist kein Ländercode angegeben, so wird die aktuelle Sprache angenommen. Der Ländercode ist nicht abhängig von der Groß- und Kleinschreibung.

Steht ein Zeichen nach dem Backslash, das in der folgenden Tabelle nicht aufgeführt ist, wird es normal ausgegeben, also der Backslash einfach entfernt.

| Zeichenkette | Ersetztes Zeichen |
|--------------|-------------------|
| \" | " |
| \\ | \ |
| \t | Tabulatorzeichen |
| \n | Neue Zeile |
| \{ | { |
| \} | } |

Beispiel:

```
"\"ein Text in Anführungszeichen\"\\nund Zeilenumbruch"
```

ergibt:

```
"ein Text in Anführungszeichen"
und Zeilenumbruch
```

Im Folgenden wird mit dieser EBNF-Syntax ein Basistyp definiert:

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

dazu kommen noch die Literale für die direkte Zuweisung:

```
const_integer := [0-9]*
const_number := const_integer '.' const_integer
const_string := ''' { <siehe oben> } '''
const_bool := ( 'true' | 'false' )
const := (const_integer | const_number | const_string |
const_dice | const_bool)
```

3.4.1 Würfelwert

Der Würfelwert ist ein besonderer Typ von Attributen, der festlegt wie ein Wert zufällig mit Hilfe von Würfeln zustande kommt. Dabei wird mit den entsprechenden Würfeln gewürfelt, und das Ergebnis entscheidet dann über Erfolg/Mißerfolg einer Aktion. Welche und wie viele Würfel benutzt werden steht in dem Würfelwert.

Ein Beispiel: Trifft der Charakter während eines Kampfes einen Gegner und will ihn angreifen, so würfelt man zuerst gegen seinen Fertigkeitwert der Fähigkeit (z. B. Schwertkampf). Diese Aktion ist in allen Rollenspielen anders, und ist in den Regelwerken nachzulesen. Nach erfolgreichem Angriff schaut man nach wieviel Schaden das Schwert verursacht, wie zum Beispiel „1W6+1“. Dann würfelt man mit einem 6-er Würfel und addiert zu dem Ergebnis 1 hinzu. Das ist der eigentliche Schaden, den der Gegner nimmt.

Es gibt bei Rollenspielen nicht nur den Standardwürfel mit sechs Seiten, sondern auch 4-er, 8-er, 10-er, 12-er, 20-er, 30-er und 100-er Würfel. (Wobei man bei dem 100-er-Würfel meist zwei 10-er Würfel nimmt, und das Ergebnis des einen mit zehn multipliziert, und zu dem anderen Ergebnis addiert). 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, also „2W6“.

Formal sieht ein Würfelwert folgendermaßen aus:

$$n_1 W_{X_1} + \dots + n_m W_{X_m} + n_0, n_i \in \text{INTEGER}, x \in \{4, 6, 8, 10, 12, 20, 30, 100\}$$

Der Würfelwert hat die Syntax:

```

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 6-er („normale“) Würfel, und zu dem Ergebnis wird zwei hinzuaddiert

1W4+3W8: ein 4er Würfel 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 akzeptiert. Wie letztendlich die Ausgabe erfolgt, ob mit *D* oder mit *W*, hängt von der Implementierung ab.

3.4.2 Gültige Operationen der Basistypen

Folgende Operationen sind zwischen den einzelnen Basistypen definiert und welcher Ergebnistyp sich dadurch ergibt (Signatur):

- NUMBER +|-|*|/ NUMBER → NUMBER:
Das Ergebnis ist das Resultat der mathematischen Verknüpfung der Werte.
- INTEGER +|-|*|/ NUMBER → NUMBER:
Das Ergebnis ist das Resultat der mathematischen Verknüpfung der Werte, wobei der INTEGER-Wert zuerst in einen NUMBER-Wert konvertiert wird.
- NUMBER +|-|*|/ INTEGER → NUMBER:
Das Ergebnis ist das Resultat der mathematischen Verknüpfung der Werte, wobei der INTEGER-Wert zuerst in einen NUMBER-Wert konvertiert wird.
- INTEGER +|-|*|/ INTEGER → INTEGER:
Das Ergebnis ist das Resultat der mathematischen 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 und zu diesem hinzuaddiert (subtrahiert). Das Ergebnis ist der neue Würfelwert.
- BOOL '|' & BOOL → BOOL:
Entspricht den booleschen Operatoren ODER bzw. UND .
- ! BOOL → BOOL:
Entspricht dem booleschen 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.

3.5 Tabellen

Eine große Rolle spielen Tabellen. So werden viele voneinander abhängigen Werte in einer Tabelle nachgeschaut. Beispielsweise hängt der Schaden, den ein Charakter verursacht, von seiner Stärke ab:

| Stärke (INTEGER) | Schaden (Würfelwert) |
|------------------|----------------------|
| 4 oder weniger | 0 |
| 5 | 1W6-5 |
| 6 | 1W6-4 |
| 7 | 1W6-3 |
| 8 | 1W6-3 |
| 9 | 1W6-2 |
| 10 | 1W6-2 |
| 11 | 1W6-1 |
| ... | ... |

Tabellen werden global in dem Interpreter gespeichert, d.h. sie sind von überall aus aufrufbar. Sie bilden einen eigenen Namensraum, und sind so unabhängig von anderen Elementen.

Eine Tabelle hat folgende Syntax:

```

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** leitet eine Tabellendefinition ein. Danach folgt der Typ des Rückgabewertes, dann der Name der Tabelle und zuletzt der Index, der für den Lookup in der Tabelle benutzt wird.

Tabellen können auch als Arrays angesehen werden, in denen für jeden Eingabewert ein Rückgabewert steht. Es gibt allerdings einige Konstrukte, die die Eingabe der Werte vereinfachen.

Die Bereiche, die zur Verfügung stehen, 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 $a \leq \text{wert} \leq b$.

Erweiterter Bereich (const ('<'|'<=') identifizier ('<'|'<=') 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 mittels des Default-Bereiches dafür einen Standard Wert angegeben werden.

Mehrere Bereiche (range ',' range):

Man kann auch verschiedene Bereiche für einen Wert angeben, indem die Bereiche durch Kommata getrennt werden.

Auf den Bereich folgt, durch einen Doppelpunkt (':') getrennt, der Wert der in diesem Bereich gültig ist. Der Wert, der hier angegeben ist, muß nicht unbedingt ein Literal sein, sondern kann auch eine Formel sein. Mehr zu Formeln siehe Kapitel 3.7.

In der Formel kann der übergebene Parameter wie ein Attribut benutzt werden, wie das folgende Beispiel zeigt.

Beispiel einer Schadenstabelle für die Stärke des Charakters:

```

TABLE DICE damage (INTEGER st)
{
    <=4: 0;
    5: 1W6-5;
    6: 1W6-4;
    7,8: 1W6-3;
    9,10: 1W6-2;
    11: 1W6-1;
    >11: 1W6+st; // Verwendung des Parameters
}

```

Bei der Auswertung einer Tabelle 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.

Eine Überprüfung, ob eine Tabelle vollständig ist, wird nicht gemacht. Auch überlappende Bereiche erzeugen keinen Fehler. Dadurch kann man die Default-Bereiche wirklich als Default benutzen, wie zum Beispiel folgendermaßen:

```
TABLE INTEGER test (INTEGER in) {  
  1: 0; // spezieller Wert für 0  
  2: 3; // spezieller Wert für 2  
  <=0,>0: in; // überall gültig: gibt einfach die Eingabe zurück  
}
```

3.6 Aufbau des Charakters

Um dem Struktur des Charakters gerecht zu werden, ist er baumartig aufgebaut: Beginnend mit einem Objekt, dem Basisobjekt, werden immer mehr Objekte angehängt, und so der Charakter nach und nach aufgebaut. Er hat zum Beispiel einen Rucksack, der wiederum ein Seil enthält, etc. Das sieht dann wie in Abbildung 3.2 aus.

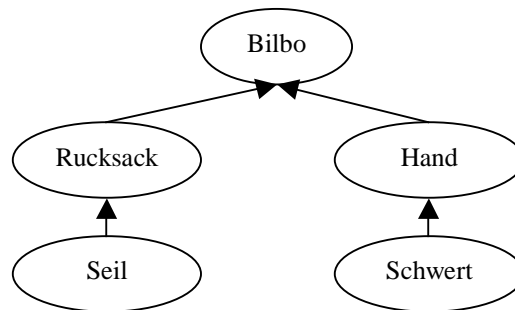


Abbildung 3.2: Beispiel eines Charakterbaumes

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

Üblicherweise sind die Objekte in Kategorien eingeteilt, wie zum Beispiel Waffen, und diese wiederum in einzelne Waffengattungen wie Fernkampfaffen und Nahkampfaffen. Häufig sind auch Zaubersprüche in verschiedene Kategorien unterteilt. Um dies abbilden zu können, ist eine Ableitungshierarchie mit (Mehrfach-) Vererbung am sinnvollsten.

Es gibt in RPD Typen, die abstrakten Klassen in anderen Programmiersprachen entsprechen, und die eigentlichen Objekte, die dann Instanzen dieser Typen sind. In RPD bilden Typen die verschiedenen Eigenschaften von Objekten. Ist ein Objekt von einem bestimmten Typ, so hat es die Eigenschaften des Typs.

Typen können selber von anderen Typen abgeleitet werden, und so für Erweiterungen der Eigenschaften sorgen. Abbildung 3.3 zeigt eine Vererbung von Typen. Siehe Kapitel 3.9 für eine genauere Beschreibung der Typen.

Legende:

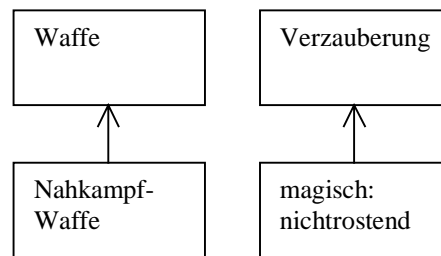
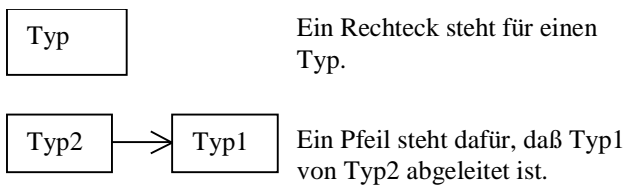


Abbildung 3.3: Beispiele für Typenableitungen

Aus Typen können die eigentlichen Objekte erzeugt werden. Dabei kann ein Objekt nicht nur aus einem einzigem Typ erzeugt werden, sondern auch aus mehreren (Mehrfachinstanziierung). Abbildung 3.4 zeigt ein Beispiel für eine Mehrfachinstanziierung.

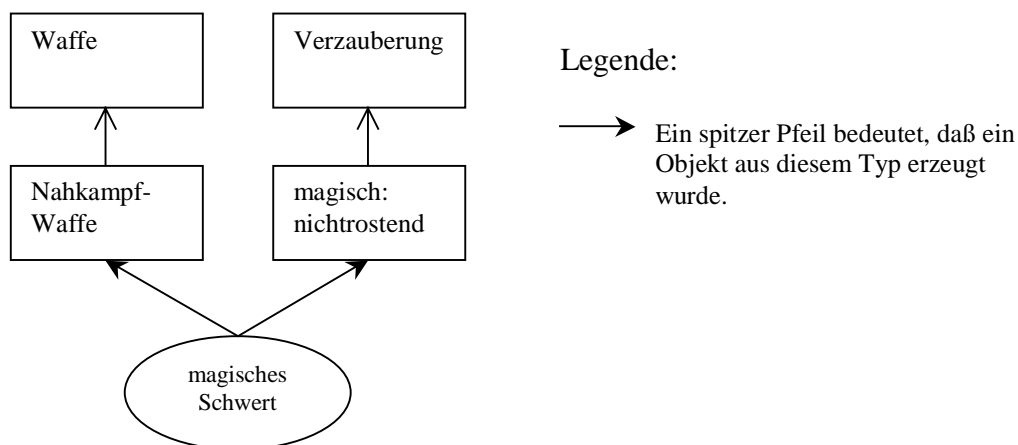


Abbildung 3.4: Erzeugen eines Objekts aus mehreren Typen

In Rollenspielen kommt es hin und wieder vor, daß sich die Eigenschaften von Objekten ändern. So kann ein Schwert nachträglich verzaubert werden, oder auch ein Zauber entfernt werden. Dies ist dadurch modelliert, indem man die Liste der Typen eines Objektes verändern kann. Es lassen sich nachträglich Typen hinzufügen, und entfernen (Siehe Kapitel 3.14).

Typen und Objekte werden über ihre Namen identifiziert. Damit es keine Konflikte gibt, haben Typen und Objekte jeweils einen anderen Namensraum. Es kann also ein Objekt und ein Typ mit dem gleichen Namen existieren.

Jetzt stellt sich die Frage, wie die Eigenschaften der Typen und damit auch den Objekten definiert werden. Dies geschieht über eine Liste mit Anweisungen, die jeder Typ, und jedes Objekt besitzt. Die wichtigsten Anweisungen sind dabei

Attributdeklarationen, und Zuweisungen. Jedes Objekt besitzt eine Liste mit Attributen, die durch Zuweisungen modifiziert werden können. Auch können andere Attribute von anderen Objekten verändert oder für Berechnungen benutzt werden.

Die Erzeugung des Charakters geschieht in mehreren Schritten. Zuerst müssen die verwendeten Typen mit ihren Anweisungen erzeugt werden. Die Anweisungen von Typen werden nicht ausgeführt, sie bilden nur eine Schablone für die eigentliche Objekterstellung. Als nächstes wird ein Objekt aus einem oder mehreren Typen erzeugt. Das Objekt ist damit erzeugt, befindet sich aber noch nicht im eigentlichen Charakterbaum. Dies geschieht im dritten Schritt, bei dem das Objekt mit einem anderen, das eine Verbindung zum Basisobjekt hat, verknüpft wird. Erst mit dem letzten Schritt werden die Anweisungen „aktiv“, also ausgeführt. Dazu werden alle Anweisungen von allen Typen in das Objekt kopiert und aktiviert. Siehe dazu auch Kapitel 4.3 der Implementierung. Abbildung 3.5 verdeutlicht den Ablauf.

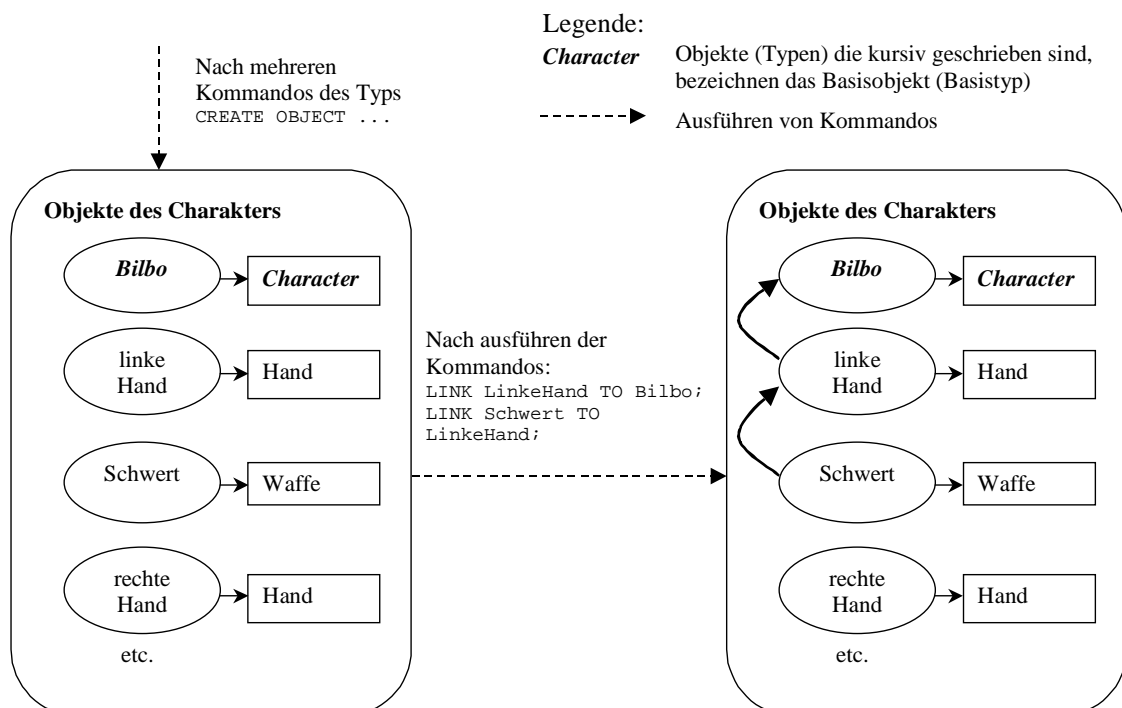


Abbildung 3.5: Verknüpfen von Objekten

Anweisungen können grundsätzlich in zwei Kategorien eingeteilt werden: In solche die nur einmal ausgeführt werden, und solche, die immer ausgeführt werden. In die erste Kategorie fallen Attributdeklarationen, während Zuweisungen in die zweite Kategorie fallen. Zuweisungen berechnen Werte, und weisen sie Attributen zu. Eine Zuweisung wird erneut ausgeführt, sobald sich eine Abhängigkeit (beispielsweise ein anderes Attribut) ändert. Sie ist also aktiv. Eine genaue Beschreibung der verschiedenen Anweisungen erfolgt in Kapitel 3.12.

Ein Problem bei der Baumstruktur eines Charakters besteht darin, daß manche Objekte sich an mehreren Vater-Objekten befinden können. So wird zum Beispiel ein Zweihandschwert in beiden Händen gehalten, und nicht nur in einer. Daher ist der Charakter nicht nur ein Baum, sondern ein gerichteter azyklischer Graph. Abbildung 3.6 zeigt die Modellierung eines Zweihandschwertes, das sich in beiden Händen befindet.

(Würde man noch eine Verbindung von Bilbo zum Schwert legen, könnte man von Bilbo's Harakiri sprechen.)

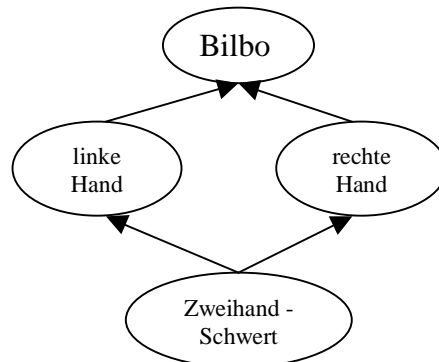


Abbildung 3.6: Beispiel für mehrere Vaterobjekte

3.7 Formeln

Um Werte berechnen zu können, und sie ggf. Attributen zuzuweisen, müssen Formeln eingegeben werden können. Neben den üblichen Operationen zwischen Basistypen können noch Attribute, Tabellen und Spezialfunktionen benutzt werden.

Eine Formel ist wie folgt (in EBNF) definiert:

```

Formula := wert | wert op wert
wert := const | tablecall | object | attribute |
specialfunction | (' wert ')
  
```

„op“ bedeutet dabei eine gültige Operation zwischen zwei Basistypen, wie in Kapitel 3.4 beschrieben. Bei mathematischen Operationen gilt die Regel Multiplikation (bzw. Division) vor Addition (bzw. Subtraktion).

Die Prioritäten der Operanden sind in der folgenden Tabelle aufgeführt.

| Operand | Beschreibung |
|---------------------|---|
| -, ! | Das Minus-Zeichen als Vorzeichen, logisches NICHT |
| *, / | Multiplikation, Division |
| +, - | Addition, Subtraktion |
| & | Logisches UND |
| | Logisches ODER |
| =, <=, <, >, >=, != | Vergleiche |

Der Wert kann entweder ein festes Literal, ein Tabellenaufruf, ein Objekt, ein Attribut oder eine Spezialfunktion sein. Im folgenden werden die einzelnen Möglichkeiten näher erläutert.

Ein Tabellenaufruf hat folgende Syntax:

```

tablecall := '#' identifier '(' wert ')'
  
```

Dabei bezeichnet der erste Identifier die Tabelle, und der Wert in Klammern ist der Wert, der nachgeschaut wird. Üblicherweise wird für den Wert ein Attribut genommen.

Ein Tabellenaufruf wird durch das vorangestellte Nummernzeichen „#“ identifiziert. Damit haben Tabellen einen eigenen Namensraum.

Beispiel:

```
#damage(4)
```

Der Zugriff auf ein Objekt (Objektreferenz) bzw. Attribut erfolgt folgendermaßen:

```
object := '$' identifier
attribute := [object '.' ] identifier
```

Objektreferenzen werden für den Zugriff auf andere Objekte benötigt; in den meisten Fällen um ein Attribut eines anderen Objekts anzusprechen. Sie werden auch für manche Spezialfunktionen benötigt. Eine Objektreferenz wird durch das Dollarzeichen „\$“ eingeleitet. Danach folgt der Name des eigentlichen Objekts. Über ein Punkt kann der Name eines Attributs angegeben und sein Wert benutzt werden.

Ist kein Objekt angegeben, so wird das aktuelle Objekt genommen, zu dem die Formel in Bezug steht. Formeln treten immer innerhalb von Anweisungen auf. Da Anweisungen nur aktiv werden, wenn sie einem Objekt gehören, das eine Verbindung zum Charaktergraphen hat, gibt es immer ein aktuelles Objekt.

Beispiele:

```
Gewicht
$Bilbo.IQ
```

Neben dem direktem Zugriff auf ein Objekt, können auch drei spezielle Objekte über reservierte Schlüsselobjekt angesprochen werden:

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

Bei „parent“ gibt es ein Problem, wenn, wie bei dem Zweihand-Schwert, mehrere Vaterobjekte existieren. In diesem Fall bezeichnet „parent“ ein beliebiges Vaterobjekt. Damit lassen sich die Berechnungen stellenweise vereinfachen wenn der Charakter nur als Baum erstellt wurde und nicht als Graph. Siehe dazu auch Kapitel 3.12.3.

Beispiele:

```
$base.IQ
$parent.Gewicht
```

Spezialfunktionen fangen mit einem Prozentzeichen „%“ an, und sind im nächsten Kapitel beschrieben.

Die folgende Tabelle zeigt eine Zusammenfassung für die Identifizierung der einzelnen Werte:

| Syntax | Wert |
|---------------|-----------------|
| "..." | STRING Literal |
| [0-9]*.[0-9]* | NUMBER Literal |
| [0-9]W D... | DICE Literal |
| [0-9]* | INTEGER Literal |
| true false | BOOL Literal |

| | |
|-------------|------------------------------|
| [a-zA-Z]... | Attribut Referenzierung |
| \$... | Objekt Referenzierung |
| #... | Tabellenaufruf |
| %... | Aufruf einer Spezialfunktion |

3.7.1 Spezialfunktionen

Manche Rollenspiele verlangen nach weiterer Funktionalität. Diese ist dann über Spezialfunktionen realisiert, wie zum Beispiel das Zählen von Objekten eines Typs.

Spezialfunktionen beginnen alle mit einem Prozent-Symbol „%“.

Im folgenden ist zu jeder Spezialfunktion die jeweilige Signatur angegeben, d. h. welche Basistypen die Funktion als Eingabe benötigt, und von welchem Basistyp das Ergebnis ist. Dabei benötigen manche Funktionen einen Typ Identifier. Dieser muß als Konstante angegeben werden, und besteht aus Buchstaben, Zahlen und dem Unterstrich. Er kann mit oder ohne Anführungszeichen angegeben werden.

3.7.2 count/countall

Die Funktion „count“ bzw. „countall“ liefert die Anzahl der Objekte eines Typs. „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! Wird für den Identifier ein leerer String angegeben, so werden alle Objekte gezählt.

Signatur: Identifier x OBJECT → INTEGER

Identifier → INTEGER

Syntax:

```
sp_count := 'count' '(' identifier [',' object] ')'  
sp_countall := 'countall' '(' identifier [',' object] ')'
```

Beispiele:

```
%count(Spell, $base)           // Anzahl der Zaubersprüche  
%countall(Seil, Rucksack)      // Anzahl der Seile im Rucksack
```

Diese Spezialfunktion liefert einen Wert vom Basistyp INTEGER zurück.

Beispiel:

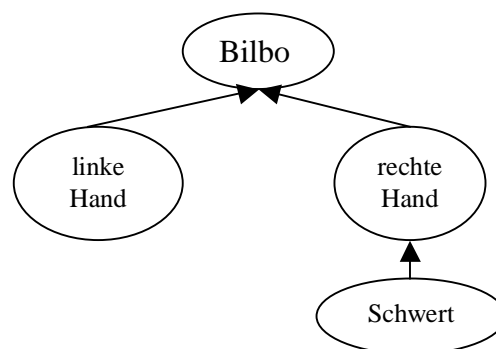


Abbildung 3.7: Beispielcharakter

Wenn alle Objekte den Typ „obj“ haben, dann ergibt sich folgendes:

```
%count(obj, base) => 2
%countall(obj, base) => 3
```

3.7.3 countparents/countallparents

Diese Funktionen sind analog zu den Funktionen „count/countall“, mit dem Unterschied, daß die Vaterobjekte gezählt werden.

Signatur: Identifier x OBJECT → INTEGER

Identifier → INTEGER

Syntax der countparents/countallparents-Funktion:

```
sp_cntparents := 'countparents' '(' identifier [',' object] ') '
sp_cntallparents := 'countallparents' '(' identifier [',' object] ') '
```

3.7.4 istype

Die Funktion „istype“ prüft, ob ein Objekt von einem bestimmten Typ ist, und liefert entsprechend true oder false zurück.

Signatur: OBJECT x Identifier → BOOL

Syntax der istype-Funktion:

```
sp_istype := 'istype' '(' object ',' identifier ') '
```

Beispiel:

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

3.8 Attribute

Attribute dienen der Speicherung von Werten. Damit ein Attribut in einem Typ bzw. Objekt in einer Attributsdeklarationsanweisung erzeugt werden kann, muß es zuvor global definiert werden. Dies erscheint auf den ersten Blick etwas umständlich, doch bietet das auch Vorteile.

Betrachten wir das Beispiel aus Abbildung 3.8:

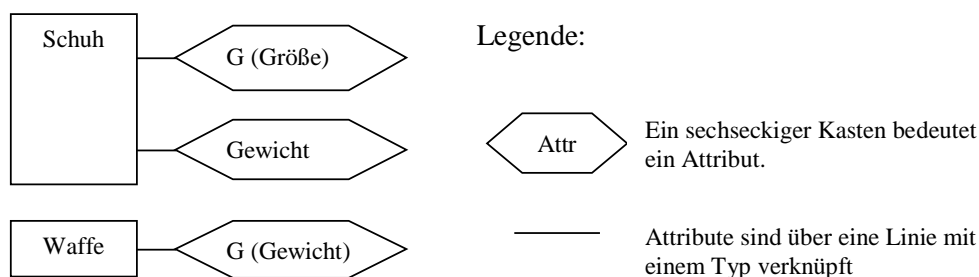


Abbildung 3.8: Attribute, mit Objekten verknüpft

In diesem Beispiel existiert das Attribut G einmal als Größe bei dem Schuh, und einmal als Gewicht bei der Waffe. Diese unterschiedliche Verwendung des gleichen Namens

eines Attributs in unterschiedlichem Kontext kann leicht zur Verwirrung des Programmierers führen.

Weitaus schwerwiegender ist in diesem Fall die Frage, was passiert, wenn nach obigem Beispiel ein Objekt aus diesen zwei Typen (Waffe und Schuh) erzeugt werden würde. Das Attribut *G* wäre zweimal unterschiedlich definiert (eventuell auch mit unterschiedlichen Basistypen), und somit mehrdeutig.

Um dieses Problem zu lösen, werden Attribute global definiert, und erhalten somit eine eigene semantische Bedeutung und einen eindeutigen Basistyp. Dadurch kann problemlos angenommen werden, daß zwei Attributsdeklarationen mit dem gleichen Namen auch das gleiche Attribut meinen.

Syntax um ein Attribut zu definieren:

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

Der BASETYPE gibt den Basistyp des Attributs an (wie „string“ oder „integer“). Der default Wert in Form eines Würfelwertes gibt an, wie der Wert des Attributs durch Würfeln erzeugt werden kann. Dies ist bei der Implementierung einer automatischen Charaktergenerierung von Bedeutung. Der Kommentar ist für eine Beschreibung des Attributs vorgesehen.

Beispiele zum Erzeugen eines Attributs:

```
CREATE ATTRIBUTE IQ BASETYPE integer DEFAULT 1W100
COMMENT Intelligenz;
CREATE ATTRIBUTE Gewicht BASETYPE number;
CREATE ATTRIBUTE ST BASETYPE integer
COMMENT "{de}Stärke{en}Strength";
```

Abbildung 3.9 verdeutlicht die Definition eines Attributs.

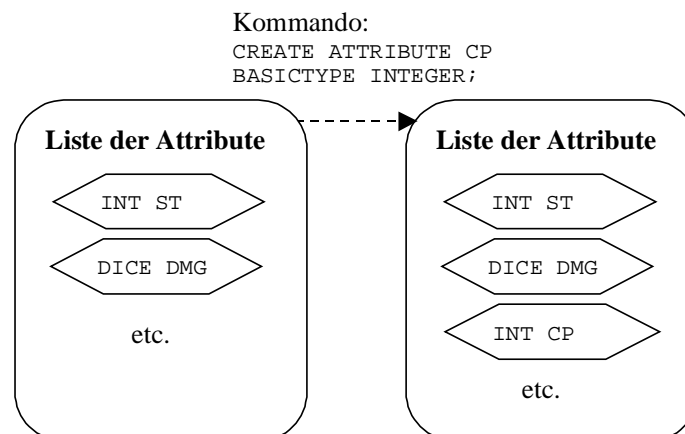


Abbildung 3.9: Erzeugen eines Attributs

3.9 Typen

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

Ein Typ wird folgendermaßen erzeugt:

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

Beispiel:

```
CREATE TYPE type FROM parenttype, parent2 {}
CREATE TYPE MissileWeapon FROM Weapon
{
    ATTRIBUTE range;
    ...
}
```

Die FROM Klausel gibt an, welches die Vartypen des neuen Typs sind.

Ein Typ kann folgende Anweisungen enthalten:

- Attributdeklarationen
- Zuweisungen
- Zusagen
- Vorbedingungen
- Events

Das Diagramm in Abbildung 3.10 gibt eine Übersicht über die Beziehungen zwischen Attributen und Typen:

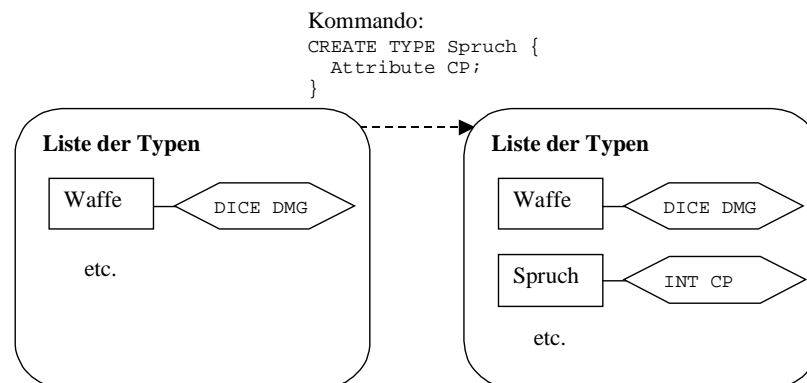


Abbildung 3.10: Erzeugen eines Typs mit einem Attribut

3.10 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 bei einem Objekt das Attribut „Magie“ erzeugt, in dem die Stärke der Magie festgehalten wird. Damit kann relativ einfach überprüft werden, ob ein Objekt magisch ist, indem man schaut, ob es vom Typ „magical“ ist.

Syntax um ein Objekt zu erzeugen:

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

Beispiele zur Erzeugung eines Objektes ist folgende:

```
CREATE OBJECT Schwert FROM type,type2 {}
CREATE OBJECT Seil from type
{
    ATTRIBUTE Length;
```

```

    SET Length = 10;
}

```

Erst wenn ein Objekt aktiv ist, also eine Verbindung zu dem Charaktergraphen erzeugt wird, werden die Anweisungen aus den Typen und dem Objekt aktiv. Dabei wird eine Liste mit Anweisungen angelegt. Hierbei gibt es zwei Probleme:

- Die Reihenfolge der Anweisungen
- Doppelte Anweisungen aufgrund von Mehrfachvererbung bzw. Mehrfachinstanziierung

Das Problem der Reihenfolge besteht darin, daß es bei folgenden Zuweisungen zu unterschiedlichen Ergebnissen kommen kann:

```

SET A=10;
SET A=A+1;

```

Deshalb muß die Reihenfolge der Anweisungen festgelegt werden:

- Kommen zwei Anweisungen aus dem gleichen Typ/Objekt, so entscheidet die Reihenfolge, in der die Anweisungen eingegeben wurden. Die erste Anweisung kommt vor der zweiten, usw.
- Kommen zwei Anweisungen nicht aus dem gleichen Typ/Objekt, entscheidet die Position des Typs/Objekts in dem die Anweisungen innerhalb des Vattertypen-Graphen definiert sind. Die Reihenfolge ist die, die in der FROM-Klausel angegeben ist. Dies wird rekursiv für alle Vattertypen gemacht, bis eine eindeutige Unterscheidung möglich ist.

Mehrfachvererbung bei Typen, und die Mehrfachinstanziierung bei Objekten, werfen letztendlich die gleichen Probleme auf. Faßt man die Instanziierung des Objektes so auf, daß vor der Instanziierung ein neuer Typ aus allen Vater-Typen des Objektes erstellt wird, so hat man nur einen einzigen Typ, von dem das Objekt abgeleitet wird. Abbildung 3.11 verdeutlicht dies.

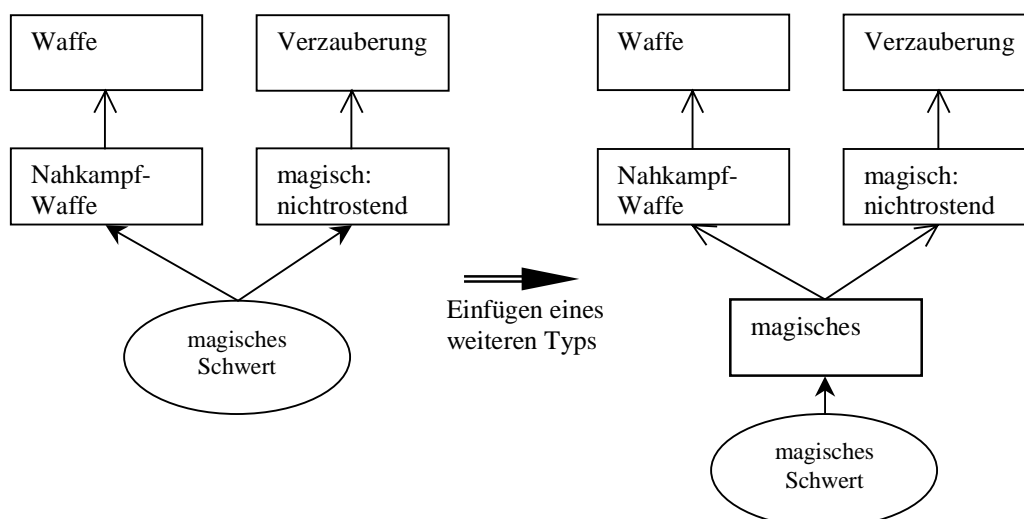


Abbildung 3.11: Einfügen eines Typs

Damit reduzieren sich diese zwei Probleme auf ein einziges, nämlich das der üblichen Mehrfachvererbung wie bei anderen Programmiersprachen (z. B. C++).

Dabei stehen die folgenden Fragen im Vordergrund:

- Was passiert bei einer Deklaration von zwei Attributen mit dem gleichen Namen?
- Was passiert mit Anweisungen, die durch mehrfache Ableitungen doppelt in dem Ableitungsgraphen auftauchen?

Das Problem mit den doppelten Attributen ist durch die globale Definition geklärt. Ist eine Anweisung, die ein Attribut für das Objekt deklariert, mehrfach vorhanden, so wird nur die erste Anweisung (und ihre Anfangszuweisung) benutzt; alle anderen werden ignoriert. Aufgrund der gleichen Bedeutung des Attributs kann dies gefahrlos gemacht werden.

Das zweite Problem der Mehrfachvererbung, die bei Ableitungen der Vatertypen aus gleichen Typen auftreten ist in Abbildung 3.12 zu sehen. *Typ2* und *Typ3* sind jeweils von *Typ1* abgeleitet, und das *Objekt* ist aus den Typen *Typ2* und *Typ3* erzeugt.

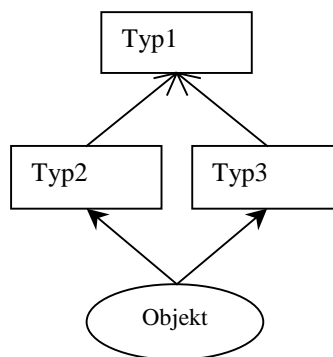


Abbildung 3.12: Typenableitung mit einem einzigem Vatertyp

In RPD sind die Anweisungen von *Typ1* nur einmal in dem *Objekt* vorhanden. Die Reihenfolge der Anweisungen von verschiedenen Typen ist in diesem Fall *Typ1*, *Typ2*, *Typ3*. Durch die rekursive Definition der Reihenfolge von unterschiedlichen Typen, kommt *Typ1* vor *Typ2*.

3.11 Durchführen der Berechnungen

Ein Charakter besteht aus verschiedenen Objekten, die aus mehreren Typen erstellt wurden, und die Objekte haben Attribute in denen Werte gespeichert werden. Der Charakter soll verändert werden können, und daher gibt es auch noch Kommandos, die ein Objekt verändern können. Damit werden Anweisungen zu dem Objekt hinzugefügt, die damit die Attribute beeinflussen können. Wie zum Beispiel den Wert eines Attributs um eins erhöhen.

Allerdings dürfen dabei Abhängigkeiten von verschiedenen Anweisungen nicht außer acht gelassen werden. So berechnet sich der Schaden, den ein Charakter verursachen kann, meist aus der Stärke. Wird der Charakter stärker, so muß sich implizit auch sein Schaden erhöhen. Eine Zuweisung ist also permanent gültig, und wenn Abhängigkeiten zwischen verschiedenen Formeln bestehen, so müssen diese nach dem Ausführen einer Anweisung aktualisiert werden.

Ein Attribut kann von unterschiedlichen Objekten aus manipuliert werden. Beispielsweise werden die Attribute des Basisobjektes häufig von Anweisungen aus verschiedenen Objekten verändert. Die Frage ist nun, in welcher Reihenfolge die

Berechnung stattfindet, bzw. welche Anweisung von welchem Objekt zuerst ausgeführt wird:

- Werden die Anweisungen der Objekte in der Reihenfolge wie die Objekte in den Baum hinzugefügt wurden, ausgewertet, oder
- Findet die Berechnung anhand der Position in dem Baum statt? Wenn ja, in welcher Reihenfolge?

Für a) spricht die einfachere Handhabung in der Berechnung. Man braucht nur die Anweisungen des neuen Objekts in den Graphen einzufügen. Problematisch ist dabei, wenn ein Objekt, entfernt, und sofort wieder eingefügt wird. Dabei kann sich die Reihenfolge der berechneten Anweisungen ändern. Außerdem können dadurch gleiche Graphen mit gleichen Objekten, unterschiedliche Ergebnisse liefern.

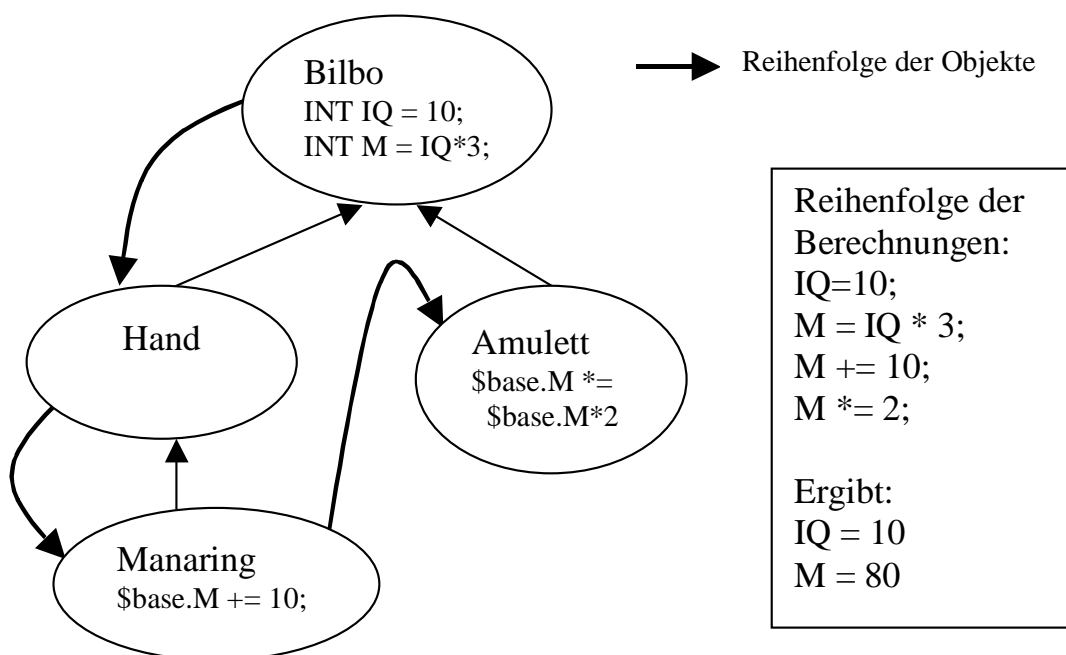


Abbildung 3.13: Tiefensuche

Daher ist Variante b) zu bevorzugen: Die Auswertung der Anweisung findet immer in einer definierten Reihenfolge statt. Für die Reihenfolge kommen verschiedene Algorithmen zur Auswahl. Dies kann zum Beispiel durch Tiefensuche (Abbildung 3.13), also rekursiv von links nach rechts, oder durch Breitensuche (Abbildung 3.14), von oben nach unten, geschehen. Bei der Breitensuche werden Objekte, die näher am Basisobjekt sind, zuerst ausgewertet. Auf diese Weise liefert jeder äquivalente Baum das gleiche Ergebnis, und die Entstehung des Baumes spielt keine Rolle bei den Berechnungen. Man kann also eine komplette Neuberechnung inklusive der Generierung der aktiven Anweisung mehrmals durchführen und kommt zu dem gleichen Ergebnis.

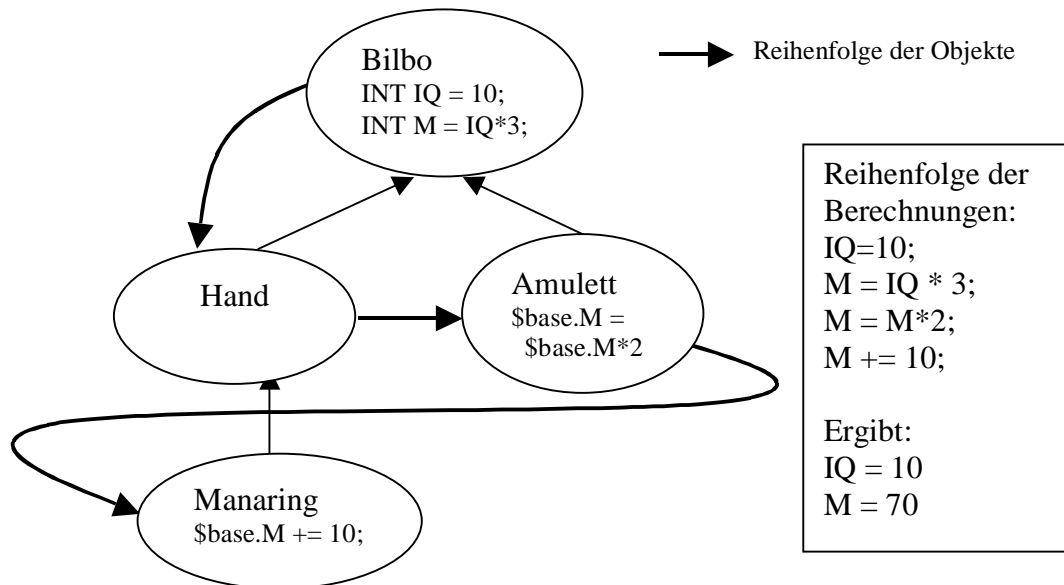


Abbildung 3.14: Breitenrecherche

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 mit RPD möglich ist. Gibt es dennoch Konflikte, werden sie auf die übliche Art und Weise gelöst: Man einigt sich mit dem Spielleiter.

Da es in RPD keinen Spielleiter gibt, bleibt nur die Möglichkeit, dem Benutzer die Wahl des Algorithmus zu überlassen. Als Standardalgorithmus ist die Breitenrecherche zu nehmen.

3.12 Anweisungen

Typen und Objekte bestehen aus einer Liste von Anweisungen, die sie beschreiben. Dies ist zum Beispiel die Deklaration von Attributen, oder Zuweisungen, die Auswirkungen dieses Objekts auf den Charakter beschreiben. Wenn beispielsweise 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 Anweisungen:

- Attributdeklarationen
- Zuweisungen
- Bedingungen
- Zusagen
- Vorbedingungen
- Events

Wichtig ist bei Anweisungen die Reihenfolge in der sie ausgeführt werden. Zum Beispiel wird eine Vorbedingung vor allen anderen Anweisungen ausgeführt, eine Attributdeklaration vor einer Zuweisung, etc. Daher haben die verschiedenen Anweisungen Prioritäten, nach denen sie vorab sortiert werden. Haben zwei

Anweisungen die gleiche Priorität, so bestimmt die Reihenfolge, in der sie in dem Objekt stehen, wann sie ausgeführt werden. Die genaue Reihenfolge wenn sie von unterschiedlichen Objekten kommen ist in Kapitel 3.10 beschrieben.

Weiterhin haben verschiedene Anweisungen eine unterschiedliche Gültigkeit. Eine Vorbedingung wird nur ein einziges mal ausgeführt, während eine Zuweisung immer gültig ist, und ausgeführt werden muß, wenn sich einer ihrer Parameter in der Formel ändert. Solch eine Gültigkeit wird permanent genannt.

3.12.1 Attributdeklarationen

Um ein Attribut in einem Objekt benutzen zu können, muß es erst für das Objekt deklariert werden. Das Attribut muß vorher allerdings in einem Kommando global definiert werden.

Priorität: hoch

Gültigkeit: permanent

Syntax einer Attributsdeklaration:

```
stmt_attr1 := 'ATTRIBUTE' identifier [ '=' formula ];
stmt_attr2 := 'REQUIRED' 'ATTRIBUTE' identifier;
```

Beispiel:

```
ATTRIBUTE name = "Mein Name";
REQUIRED ATTRIBUTE name;
```

Attribute sind grundsätzlich für alle Typen/Objekte sichtbar, d. h. es gibt keine Kapselung der Attribute wie *protected* oder *private* deklarierte Attribute bei Java oder C++.

Ein Attribut kann auf einen Wert initialisiert werden. Dabei kann eine beliebige Formel angegeben werden. Die Initialisierung verhält sich wie eine Zuweisung, und kann auch als solche geschrieben werden. Folgende Anweisungen sind äquivalent:

```
ATTRIBUTE A = 10;
ATTRIBUTE A; SET A=10;
```

Wird die zweite Variante mit dem Schlüsselwort **REQUIRED** benutzt, bedeutet dies, daß dieses Attribut bei der Erzeugung eines Objektes initialisiert werden muß. Dadurch kann man das Setzen eines Attributs erzwingen. Wird das Attribut nicht initialisiert, so wird eine Fehlermeldung ausgegeben. Ein Attribut mit den **REQUIRED** Flag kann nicht initialisiert werden.

Fehlt beim Anlegen eines Attributs die Initialisierungsanweisung, bekommt das Attribut folgenden Standardwert zugewiesen:

| Attributtyp | Standardwert |
|-------------|--------------------|
| STRING | "" (leerer String) |
| INTEGER | 0 |
| NUMBER | 0 |
| BOOL | true |
| DICE | 0 |

3.12.2 Zuweisungen

Eine Zuweisung ist das Ersetzen des Wertes eines Attributs mit dem Ergebnis einer Formel.

Priorität: normal

Gültigkeit: permanent

Syntax der SET-Anweisung:

```
stmt_set := 'SET' attribute ('='|'+=') formula ';' 
```

Beispiele:

```
SET Strength = 10;
SET IQ += 1;
```

Da häufig ein Wert aufaddiert wird, wie zum Beispiel bei

```
SET A = A + 1;
```

gibt es die Kurzschreibweise mit dem Symbol „+=“. Daher würde folgende Zuweisung das Gleiche ergeben:

```
SET A += 1;
```

Es gibt zwei Arten von Zuweisungen: Modifikationen, die einen Wert verändern, und Ersetzungen, die das Attribut auf einen festen Wert setzen. Eine Modifikation hat das zu setzende Attribut in seiner Formel (lokaler Zyklus) während die Ersetzung unabhängig von dem zu setzenden Attribut ist.

Beispiel:

```
SET A=10;           // Ersetzung
SET A=A+1;         // Modifikation
```

3.12.3 Zuweisungen mit „parentall“

Eine SET-Anweisung hat das Problem, wenn ein Vaterobjekt angesprochen werden soll, es aber mehrere Vaterobjekte gibt. Die Zuweisung an das Attribut „\$parent.Attr“ würde den Wert in einem beliebigem Vaterobjekt setzen. Daher gibt es die Möglichkeit mit dem speziellen Objekt „parentall“, alle Väterobjekte anzusprechen. „parentall“ ist dabei ein Schlüsselwort, das groß oder klein geschrieben werden kann.

Priorität: normal

Gültigkeit: permanent

Syntax:

```
stmt_set := 'SET' '$parentall' '.' identifier ('='|'+=')
formula ';' 
```

Beispiel:

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

Bei dieser Art von Anweisung wird in jedem Vaterobjekt das angegebene Attribut gesetzt. Existiert das Attribut in einem Vaterobjekt nicht, so wird eine Fehlermeldung ausgegeben. Mit Hilfe der Spezialfunktion *countparents* (Siehe Kapitel 3.7.3), kann damit eine Aufteilung eines Gewichts an die Vaterobjekte erreicht werden:

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

Objekte aus diesem Typ haben ein Eigen- und ein Gesamtgewicht. Fügt man ein Objekt an ein anderes hinzu, so wird das Gesamtgewicht aller übergeordneten Objekte verändert. In diesem Fall erfolgt eine gleichmäßige Gewichtsverteilung auf die Vaterobjekte.

Jetzt folgt ein Beispiel mit Objekten die aus dem obigen Typ erstellt wurden. Neben den Objekten sind die beiden Attribute „G“ und „GG“ aufgeführt, mit ihren jeweiligem Werten.

Charakter Bilbo ohne Schwert (Abbildung 3.15):

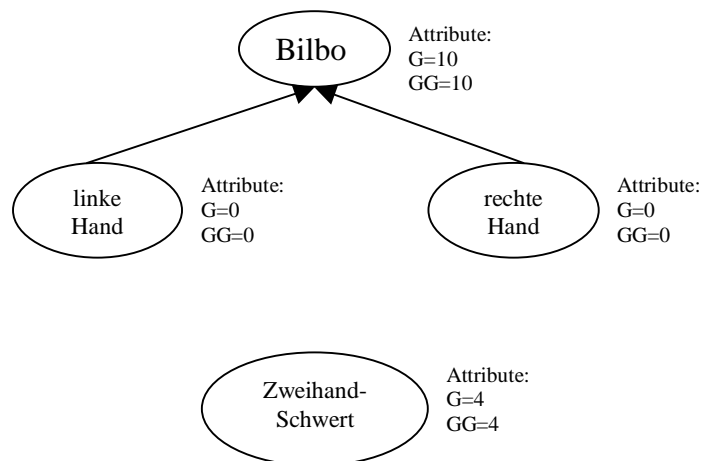


Abbildung 3.15: Charakter ohne Schwert

Charakter Bilbo mit Schwert in der linken Hand (Abbildung 3.16):

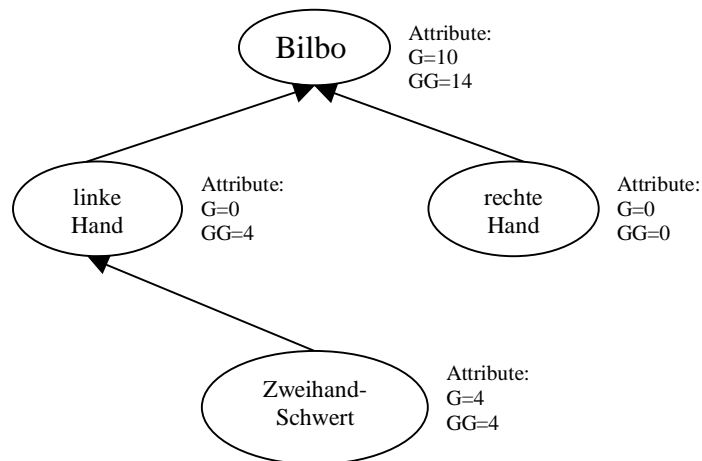


Abbildung 3.16: Charakter mit Schwert in einer Hand

Charakter Bilbo mit Schwert in beiden Händen (Abbildung 3.17):

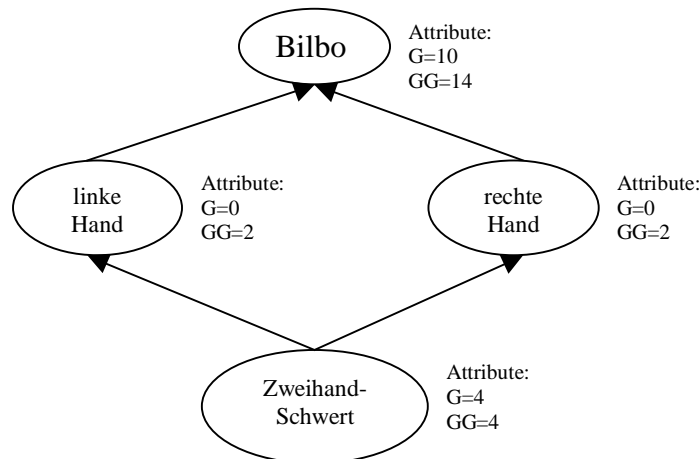


Abbildung 3.17: Charakter mit Schwert in zwei Händen

3.12.4 Zuweisungen mit Prioritäten

Ein Attribut wird üblicherweise aus 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 auch um das Problem Ersetzen/Modifizieren, wie zum Beispiel folgende zwei Anweisungen zeigen:

- (1) SET A=10; // Ersetzung
- (2) SET A=A+1; // Modifikation

Das Problem tritt auf, wenn Anweisungen von Typen kommen. Sie werden üblicherweise vor den Anweisungen, die in dem Objekt definiert wurden, ausgeführt. Soll aber eine Modifikations-Anweisung bei einer Typ-Deklaration benutzt werden, so stößt man auf das Problem, daß diese Anweisung bei einem späteren Ersetzen 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 += 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 dem CREATE OBJECT-Kommando überschrieben.

Eine Lösung des Problems wäre es, die Anweisung „SET Level=10“ durch die Anweisung „SET Level+=10“ zu ersetzen. Allerdings verliert man schnell den Überblick, wie das Attribut letztendlich berechnet wird. Außerdem muß in diesem Fall bei einer Änderung des Grundwertes (hier bei der Attributsdeklaration) der Level von allen abgeleiteten Objekt geändert werden. Letztendlich ist es auch ein semantischer Unterschied, ob der Level gesetzt oder verändert wird. Es soll zum 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+=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 drei verschiedene Prioritäten bei Zuweisungen.

Achtung: Die Priorität von SETFIRST ist zwischen der Priorität Hoch und Normal einzuordnen. Analog liegt die Priorität von SETLAST zwischen Normal und Niedrig.

Als Lösung des obigen Beispiels würde man die Anweisung

```
SETLAST Level += 1;
```

benutzen.

3.12.5 Bedingungen

Es gibt 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 wirken, wenn es eine passende Batterie enthält.

Priorität: normal

Gültigkeit: permanent

Syntax der Bedingung:

```
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...
}
```

Die Zuweisungen, die in den beiden Blöcken sind, können nur Zuweisungen sein. Es macht keinen Sinn, Attributsdeklarationen hier zuzulassen.

Um die Implementierung zu vereinfachen, sind keine geschachtelten IF-Anweisungen möglich. Siehe Kapitel 4.4.2.

3.12.6 Zusagen

Zusagen werden dazu benötigt um gewisse Aussagen abprüfen zu können. Zum Beispiel kann ein Rucksack nicht unbegrenzt Dinge aufnehmen. Deshalb kann mittels einer Zusage geprüft werden, ob das Gesamtgewicht der enthaltenen Gegenstände nicht zu groß ist.

Priorität: niedrig

Gültigkeit: permanent

Syntax:

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

Beispiel:

```
ASSERT WARN (a<=12) "Attribut a is größer als 12. (a=" + a +
")";
```

Die Formel muß einen BOOL-Wert ergeben; und wenn dieser `false` ergibt, wird ein Fehler (bzw. eine Warnung) ausgegeben. Im Fall einer Warnung wird das Laufzeitsystem normal weiterarbeiten; im Fall eines Fehlers den Fehler ausgeben, und die Bearbeitung des Kommandos abbrechen.

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

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 auch Werte von Attributen in der Fehlermeldung angeben zu können.

3.12.7 Vorbedingungen

Vorbedingungen sind bezüglich der Syntax ähnlich aufgebaut wie Zusagen. Der Unterschied besteht darin, daß sie vor dem Hinzufügen zu dem eigentlichen Basisobjekt geprüft werden. Schlägt eine Vorbedingung fehl, so kann das Objekt nicht gelinkt 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 mitauszugeben.

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

Priorität: hoch (höher als bei einer Attributdeklaration)

Gültigkeit: einmalig

Syntax:

```
stmt_require := REQUIRE '(' formula ')' [ formula ];
stmt_reqobj  := REQUIRE OBJECT '(' '$' object ')' [ formula ];
```

Die erste Variante ist die allgemeine Syntax; sie benötigt einen booleschen Ausdruck. Ist die Bedingung nicht erfüllt, so wird eine Fehlermeldung mittels der Formel 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“ kennen muß. Das sähe dann folgendermaßen aus:

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

Mit der ersten Variante kann diese Vorbedingung ebenfalls implementiert werden:

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

Der Grund für die Sonderbehandlung in diesem Fall liegt darin, daß später die Möglichkeit bestehen soll, alle fehlenden Objekte einfügen zu können. Wenn also der Charakter den Spruch „explosiver Feuerball“ lernen will, und er noch keinen „Feuerball“ zaubern kann, dann soll es möglich sein, alle Objekte, die für den neuen Spruch benötigt werden, auf einmal hinzufügen zu können. Dies wäre bei der

alternativen Darstellung (`REQUIRE (%countall(Feuerball, $base)>0)`) nicht möglich.

3.12.8 Events

Es gibt die Möglichkeit, Anweisungen auszuführen, wenn das Ergebnis einer Formel sich ändert. Das kann zum Beispiel dazu benutzt werden, um bei Erreichen eines neuen Levels einen Zähler zurückzusetzen. Der Zähler kann die Anzahl der erlernten Sprüche pro Stufe begrenzen. Dies ist in dem Rollenspiel *Rolemaster®* der Fall.

Eine weiteres Beispiel für einen sinnvollen Einsatz von Events ist, beim Umsetzen eines Flags einige Attribute zu speichern.

Als Anweisungen in dem Event-Block sind nur Zuweisungen zugelassen.

Priorität: niedrig

Gültigkeit: permanent

Syntax:

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

Beispiel:

```
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;  
  }  
}
```

3.12.9 Zusammenfassung der Prioritäten

Zum Schluß der Anweisungen noch eine Liste der Anweisungen nach Priorität sortiert (Anweisungen mit höherer Priorität kommen zuerst.):

1. Vorbedingungen
2. Attributdeklarationen
3. SETFIRST-Zuweisungen
4. SET-Zuweisungen, Bedingungen, Events
5. SETLAST-Zuweisungen
6. Zusagen

3.13 Charaktergrapherstellung

Bis jetzt gibt es nur eine Menge von Objekten, die den Charakter bilden sollen. Damit die Objekte miteinander interagieren können (und die Anweisungen ausgeführt werden), müssen sie zu einem Baum bzw. einem gerichteten azyklischem Graph zusammengefügt werden.

Dazu muß zuerst das Basisobjekt festgelegt werden. Alle Objekte, die mit dem Basisobjekt verbunden sind, bilden den eigentlichen Charakter. Das Basisobjekt wird implizit angelegt, indem ein Objekt von einem bestimmten Typ, dem Basistyp, erzeugt wird. Welcher Typ das ist, wird mit dem BASE-Kommando festgelegt.

Syntax des BASE-Kommandos:

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

Beispiel:

```
BASE Character;
```

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

Das Basisobjekt ist das einzige aktive Objekt, das kein Vaterobjekt hat. Bei der Erstellung des Basistyps und des Basisobjekts sollte darauf geachtet werden. Insbesondere bedeutet dies, daß in den Formeln des Basisobjekts keine Objektzugriffe auf `parent` (bzw. `parentall`) erfolgen dürfen.

Der nächste Schritt ist die Verknüpfung der erstellten Objekte mit dem Basisobjekt. Dies geschieht mittels des LINK-Kommandos:

Syntax des LINK-Kommandos:

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

Beispiel:

```
LINK objekt1 TO objekt2;
```

Dadurch wird ein Objekt an ein anderes gebunden. Es dürfen dabei keine Zyklen entstehen. Dies wird automatisch erkannt und eine entsprechende Fehlermeldung ausgegeben.

Für die erzeugten Objekte gilt grundsätzlich:

Entweder sie sind mit dem Basisobjekt verbunden, dann werden die Anweisungen der Objekte ausgeführt, oder sie befinden sich noch ungebunden in der Objektliste, dann werden sie nicht ausgeführt.

Um eine Verknüpfung zu lösen, existiert das UNLINK-Kommando:

Syntax des UNLINK-Kommandos:

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

Beispiel:

```
UNLINK objekt1 FROM objekt2;
```

Damit wird eine Verbindung zwischen diesen beiden Objekte aufgehoben.

Ein Objekt kann mit dem DELETE-Kommando komplett entfernt werden:

Syntax des DELETE-Kommandos:

```
cmd_delete := 'DELETE' 'OBJECT' identifier ';' ;
```

Beispiel:

```
DELETE OBJECT objekt;
```

Im Gegensatz zu UNLINK wird das Objekt sowie alle an es gebundenen Objekte gelöscht! Das Objekt ist danach nicht mehr in der Objektliste 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;
```

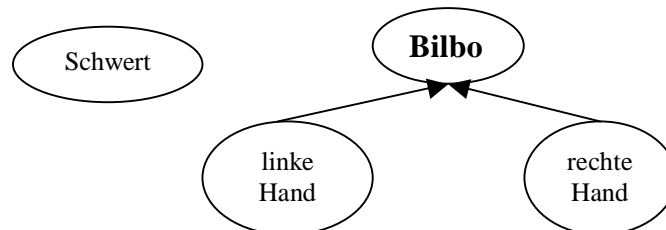


Abbildung 3.18: Charakter ohne verknüpftem Schwert

```
LINK Schwert TO rechteHand;
```

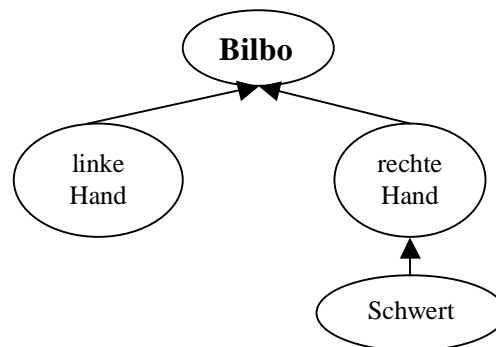


Abbildung 3.19: Charakter mit Schwert in rechter Hand

```
// Jetzt in die linke Hand geben:
UNLINK Schwert FROM rechteHand;
LINK Schwert TO linkeHand;
```

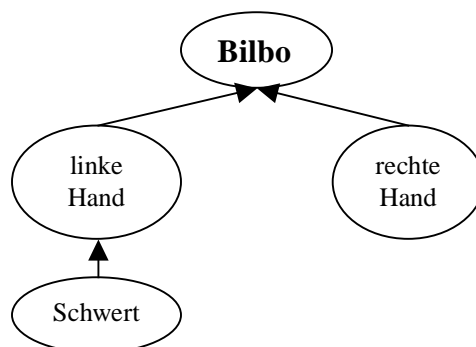


Abbildung 3.20: Charakter mit Schwert in linker Hand

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

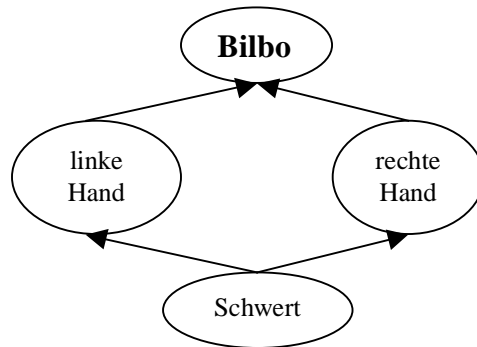


Abbildung 3.21: Charakter mit Schwert in beiden Händen

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

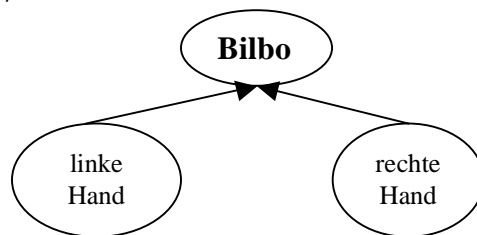


Abbildung 3.22: Charakter ohne Schwert

Dadurch, daß es eine einzige Menge mit Objekten gibt, sind alle Objekte eines Charakters eindeutig bestimmt. Es ist daher nicht möglich, daß es zwei Objekte mit der gleichen Bezeichnung gibt.

3.14 Modifizieren des Charakters

Eine wichtige Funktionalität ist die Modifikation von Objekten. Damit können Objekte nachträglich verändert, und so dem aktuellen Spielgeschehen angepaßt werden.

Syntax des MODIFY-Kommandos:

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

Beispiel:

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

Es wird immer ein Objekt modifiziert. Die neuen Anweisungen werden einfach an die bereits vorhandene Liste der Anweisungen des Objektes angefügt.

Zusätzlich kann mit Hilfe des MODIFY-Kommandos die Typliste des Objekts verändert werden, indem ein zusätzlichen Typ hinzufügt oder entfernt wird.

Dazu gibt es zwei weitere Anweisungen:

Syntax:

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

Beispiele:

```
ADD TYPE magisch;
REMOVE TYPE magisch;
```

Wird ein neuer Typ zu einem Objekt hinzugefügt, so bekommt es alle Anweisungen des Typs zugewiesen. Wird ein Typ von dem Objekt entfernt, so werden alle Anweisungen, die dieser Typ hinzugefügt hat entfernt. Dies geschieht allerdings nicht, wenn dieser Typ durch Mehrfachvererbung noch immer Teil des Objekts ist. Dabei wird auch Rücksicht auf mehrfach definierte Attribute genommen.

3.15 Objektbibliothek

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

Die entsprechenden Kommandos für die Objektbibliothek sind folgende:

| Charakter Kommando | Bibliothek Kommando |
|---------------------------|----------------------------|
| CREATE OBJECT | CREATE LIBRARY OBJECT |
| DESTROY OBJECT | DESTROY LIBRARY OBJECT |

Um ein Objekt aus der Objektbibliothek zu benutzen, gibt es folgendes Kommando:

Syntax:

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

Beispiel:

```
CREATE OBJECT Schwert FROM LIBRARY OBJECT Schwert;
```

Damit wird eine Kopie des Objektes aus der Objektbibliothek erzeugt, und der Objektliste hinzugefügt. Damit ein Objekt auch mehrfach benutzt werden kann, muß ein Name für das neue Objekt angegeben werden. So können beispielsweise ohne Probleme mehrere Seile erzeugt werden, die von dem gleichen Bibliotheks-Objekt stammen.

4 Implementierung

4.1 Allgemein

RPDL in Java besteht aus den Klassen in der Package *de.mutschler.rpd*. 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 zu Java 1.1 kompatibel zu bleiben. Ein Vorteil von Java 1.2 ist das Collection API, mit dem man die verschiedenen Container für die Objekte (wie Listen, Sets, Maps) elegant implementieren kann.

Es gibt eine Basisklasse (*RpdBase*), die die komplette RPD Umgebung handhabt. Sollen mehrere Charaktere erstellt werden, können mehrere Instanzen von *RpdBase* gebildet werden. Sie enthält die verschiedenen Container für die einzelnen Elemente (Tabellen, Typen, Objekte, Objektbibliothek, etc.).

Generell wurden folgende Programmierrichtlinien bei der Implementierung berücksichtigt:

- Der Zugriff auf Attribute erfolgt immer über Wrapper-Methoden (z. B. *getName()*, *setName()*).
- Die Methode *toString()*, die jedes Java-Objekt enthält, wird nur für debug-Zwecke benutzt.
- Fehler werden als *RpdException* geworfen.
- Zur Vereinfachung des Debugging ist jede Klasse von *de.mutschler.util.LogObj* abgeleitet, die Methoden für die Ausgabe (in ein File, oder auf die Konsole) zur Verfügung stellt.

4.2 Einlesen

Das Einlesen der Files geschieht über einen Parser, der mit Hilfe von Lex/Yacc-Tools erzeugt wird. Dazu wird die Java-Implementierung dieser Programme benutzt: Für den Lexer JFlex [14], und für Yacc das Programm CUP [15]. Lex und Yacc sind Standard-Hilfsmittel zum Erzeugen eines Compilers. Genauere Informationen über die Arbeitsweise dieser Programme sind in der Literatur zu finden [16], [17].

Der Parser analysiert die Quelldateien, und schickt die Kommandos mittels der Methode *addCommand()* an das *RpdBase*-Objekt. Als zentrale Klasse zum Einlesen dient *de.mutschler.rpd.fileimporter.FileImporter*. Sie bekommt ein *RpdBase*-Objekt, und einen Dateinamen, und importiert diese Datei mittels des von JFlex und CUP erzeugten Lexer und Parser.

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

Einen Vorteil bietet die einfache Schnittstelle zum `RpdBase`-Objekt: Es läßt sich problemlos ein `INCLUDE`-Kommando einbinden: Dazu wird nur eine neue Instanz der Klasse `FileImporter` mit dem Namen der einzubindenden Datei erzeugt, und das entsprechende File eingelesen. Das Ganze kann rekursiv benutzt werden, um so alle weiteren Dateien einzulesen. Die eingelesenen RPD-Kommandos werden dadurch an das gleiche `RpdBase`-Objekt in der beabsichtigten Reihenfolge gesendet.

Das folgende Diagramm verdeutlicht den Ablauf.

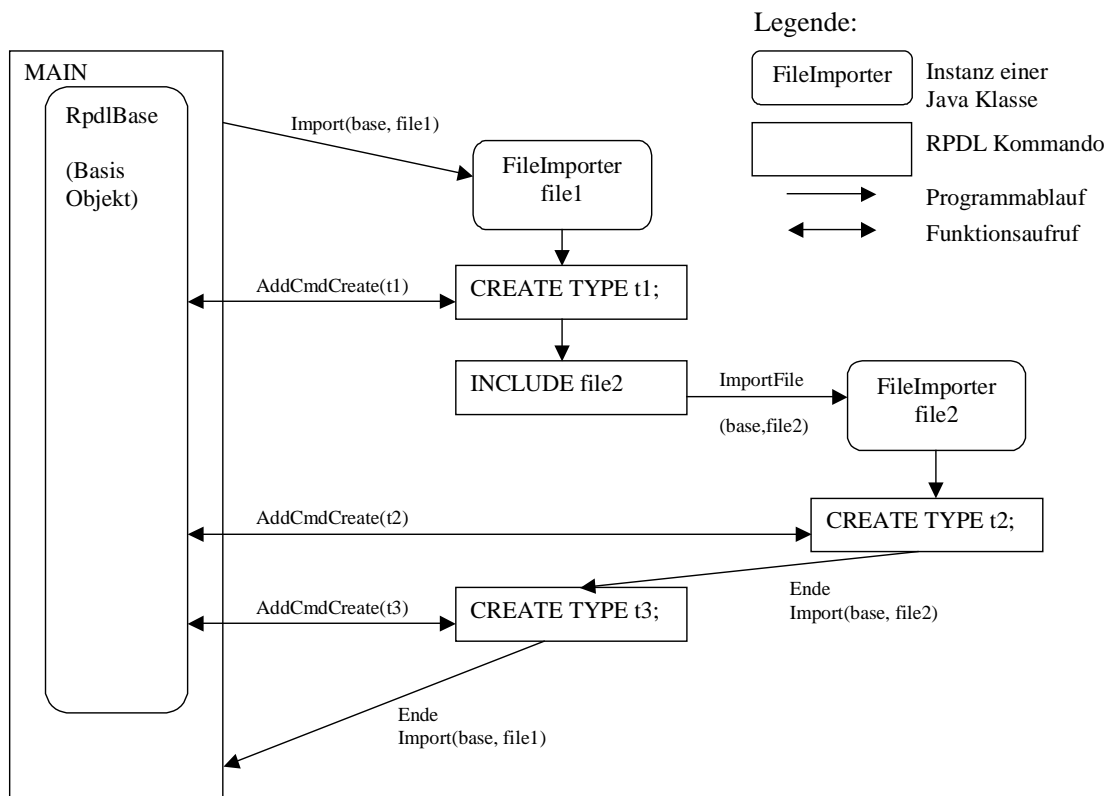


Abbildung 4.1: Ablauf des Einlesens der Kommandos

In der Implementierung sieht das so aus: Der `FileImporter` generiert Objekte des Typs `RpdCommand` aus der Package `de.mutschler.rpd.command`. Dies ist die Basisklasse, von der alle Kommandos abgeleitet sind. Wie zum Beispiel `CmdCreateAttr`, `CmdLink`, ... Die Klasse `RpdCommand` verfügt über eine Methode: `add(RpdBase)`. Diese kann kommando-spezifisch implementiert werden und wird von der `RpdBase` während der `addCommand()` Methode aufgerufen. Dabei werden kommandospezifische Objekte wie Typen, Tabellen, etc. generiert, und in das RPD-System eingebunden.

Des weiteren wird am Ende von `addCommand()` der `CalcHandler` aufgerufen, um neue Berechnungen durchzuführen. Dies passiert nur, wenn eine Veränderung des Charaktergraphen stattgefunden hat. Siehe Abbildung 4.2

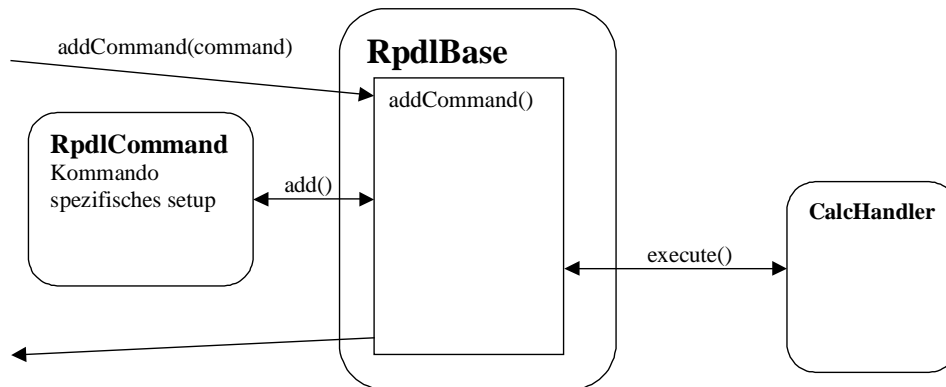


Abbildung 4.2: Ablauf des Hinzufügens von Kommandos

4.2.1 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 RPD

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

4.3 Anweisungen

Eine wichtige Rolle bei RPD

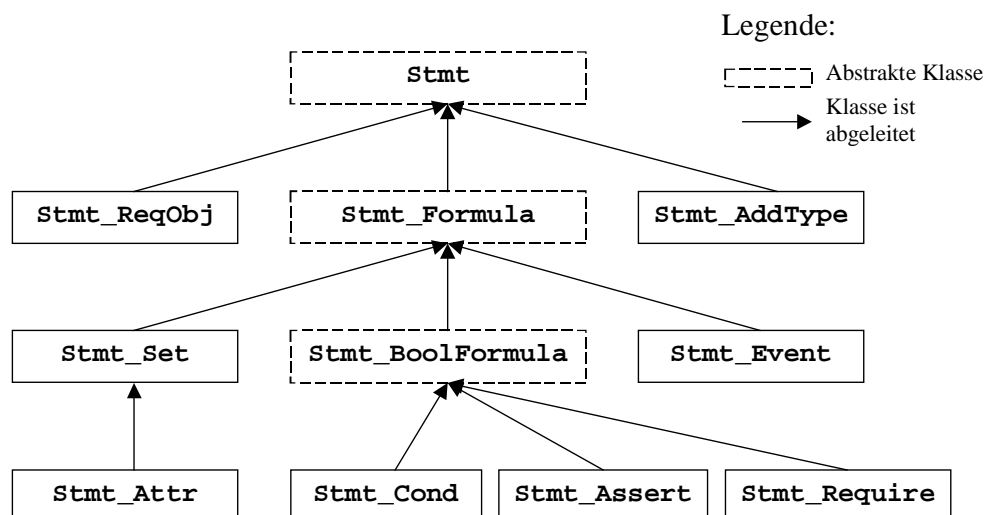


Abbildung 4.3: Klassenhierarchie der Anweisungen

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

- **ID-Manager** (nur in Stmt): Jede Anweisung bekommt eine eindeutige, aufsteigende ID. Dadurch können die Anweisungen innerhalb der Objekte

sortiert werden, wenn sie gleiche Priorität haben. Außerdem vereinfacht dies die Implementierung von `equals()`, dem Test auf Gleichheit zweier Anweisungen: Es brauchen nur noch die IDs der Anweisungen verglichen zu werden.

- Die Methode **checkStatement()**: Sie wird zum Zeitpunkt von `addCommand()` aufgerufen. Genauer gesagt, von `RpdCommand.add()` der jeweiligen Kommandoimplementierung (z. B. von `CmdCreateObject().add()`). In `checkStatement()` findet die statische Analyse der Anweisungen statt. Sie bekommt ein Entity (Die Basisklasse von `RpdObject` und `RpdType`) übergeben und kann sich so auf das zugehörige Objekt beziehen.
- Die Methode **setDerivedEntity()**: Damit wird das `RpdObject` bzw. `RpdType` gesetzt, in dem die Anweisung definiert ist.
- Die Methode **onBaseLink()**: Diese wird aufgerufen, wenn das Statement aktiv wird, also zu einem Objekt gehört, das neu in den Charaktergraphen aufgenommen (oder entfernt) wird. Üblicherweise wird diese Methode durch den Link-Befehl getriggert. Hier werden die Events für die Berechnung aufgesetzt. (Eintrag in die Berechnungs- und Triggerlisten von Attributen. etc.)
- Die Methode **execute()**: Sie führt die Anweisung aus. Hier werden keine Abhängigkeiten 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. der Eintrag der Zielattribute bei Zuweisungen in den `CalcHandler`.
- Die Methoden **getDependencies()** und **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 Anweisungen bei Berechnungen kann den einzelnen Arten von Anweisungen individuelle Prioritäten gegeben werden. Diese wird von der Methode **getPriority()** geliefert. Z. B. hat `Stmt_Assert` die niedrigste Priorität, da sie **nach** dem Ausführen von SET-Anweisungen ausgeführt werden muß.

4.4 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 aus Abbildung 4.4 dienen:

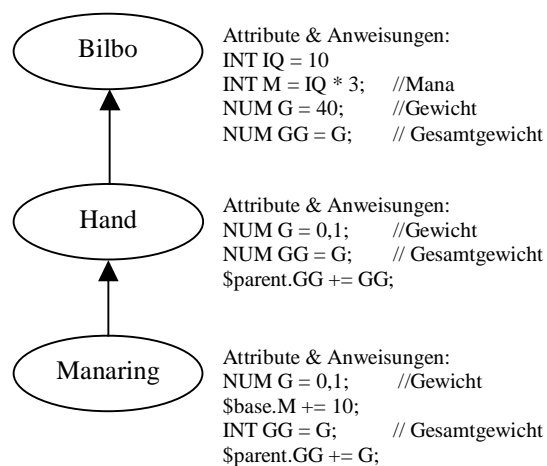


Abbildung 4.4: Beispielcharakter für Berechnung

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 Attribut 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: IQ=10; M=30

Das Problem sind nun Anweisungen die Attribute hinterher ändern. Da Zuweisungen permanent gültig sind, müssen ihre Abhängigkeiten berücksichtigt werden. Der Algorithmus schlägt fehlt nach dieser Zuweisung fehl:

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

So erhalten wir in der Berechnung nun:

(3) IQ = 12

Das Ergebnis ist dann: 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 [9], 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 für die Berechnung der abhängigen Attribute.

Es ist also notwendig, daß eine Formel neu berechnet wird, sobald sich eines 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 und M sehen so aus:

$IQ_T = \{2\}$

$M_T = \{\}$

Der Algorithmus zum Ausführen einer neuen Anweisung lautet nun:

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

Da 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 += 1;}
```

Die Anweisung „ $IQ+=1$ “ ist äquivalent zu der Anweisung „ $IQ=IQ+1$ “. Im folgenden wird mit letzteren Schreibweise fortgefahren, da sie das Problem besser verdeutlicht.

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:

$IQ_T = \{2, 4\}$

$M_T = \{\}$

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

Abhilfe schafft hier eine Liste mit Anweisungen zur Berechnung der Attribute, 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, die Berechnungsliste wird mit dem neuen Algorithmus erzeugt:

1. Füge die Formel zur Berechnungsliste des Zielattributs hinzu.
2. Für alle Attribute in der Formel: Wenn Zielattributs \notin Formelattribute, dann füge die Formel zur Triggerliste des Zielattributs hinzu.
3. Berechne alle Formeln der Berechnungsliste.
4. Berechne alle Attribute neu, die in der Triggerliste für das Zielattribut stehen.

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 die erste Menge die Berechnungsliste, und die zweite Menge die Triggerliste

| Attribut | 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 mehrere Formeln hinweg werden noch nicht erkannt. Dieses Problem läßt sich folgendermaßen lösen:

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

Damit alle Abhängigkeiten richtig propagiert werden, und auch Zyklen erkannt werden, sind folgende Schritte für eine neue Zuweisung durchzuführen:

1. Füge die Formel der Berechnungsliste des Zielattributs hinzu.
2. Erzeuge eine Liste mit abhängigen Attributen der Formel (neue Abhängigkeiten).
3. Entferne das eigene Attribut aus der Liste der neuen Abhängigkeiten. Damit werden lokale Zyklen erlaubt.
4. Prüfe, ob sich das (aktuelle) Zielattribut 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 diese Attribute Schritt 4 und 5 aus.

Der Algorithmus läuft rekursiv ab. Er terminiert, da alle Zyklen erkannt werden, und in diesen Abhängigkeiten keine Zyklen existieren.

Eine weiterer Vorteil bei dem Aufbau der Abhängigkeitsliste besteht darin, daß später das Sortieren der Berechnungen schnell und effizient durchgeführt kann: Ein Attribut muß vor dem anderen berechnet werden, wenn es in der Abhängigkeitsliste des anderen enthalten ist.

Damit kann eine weitere Zyklererkennung zur Laufzeit eingebaut werden (die allerdings niemals zuschlagen sollte!). Es gibt einen CalcHandler, der alle Berechnungen von Attributen zwischenspeichert. Werden die Berechnungen 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. Siehe dazu auch die Implementierung des CalcHandlers in Kapitel 4.5.1.

4.4.1 Handhabung von Events (ON CHANGE)

Events lassen sich sehr einfach implementieren, wenn noch folgende Erweiterung gemacht wird:

- Grundsätzlich läßt sich die Berechnung optimieren, wenn bei der Berechnung des Attributs der Wert vorher und nachher verglichen wird, und wenn sich das Attribut nicht verändert hat, werden die Attribute aus der Triggerliste nicht neu berechnet.

Für die ON CHANGE Anweisung bedeutet die: Die Formel innerhalb der Anweisung bekommt den Wert an ein Pseudo-Attribut zugewiesen, und dieses Pseudo-Attribut bekommt jede Anweisung innerhalb des ON CHANGE Blocks als einzige Abhängigkeit.

4.4.2 Handhabung von Verzweigungen (IF)

Das Problem bei Verzweigungen ist, daß unterschiedliche Anweisungen benutzt werden, je nach Wert der Bedingung.

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

1. Möglichkeit:

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

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 ergibt sich aber kein konsistentes Ergebnis.

2. Möglichkeit:

Die IF-Anweisung merkt sich, welche Attribute in den Anweisungen der beiden Zweigen benutzt werden, und trägt sich bei diesen Attributen ebenfalls in die Triggerliste ein. Die Anweisungen in den Zweigen selber dürfen sich dabei nicht bei den Attributen eintragen, 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 Anweisungen in den Zweigen. Wenn sie sich alle 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)
}
[...]
```

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

3. Möglichkeit

Die Anweisungen in beiden Zweigen tragen sich in die Triggerlisten ihrer Attribute, und gleichzeitig in die Triggerlisten der Attribute der Bedingung ein. Damit werden die Anweisungen ausgeführt wenn sich die Bedingung **oder** ein Attribut der Anweisung ändert. Diese Variante liefert ein konsistentes Ergebnis, hat aber den Nachteil, daß die Bedingung wiederholt ausgeführt wird.

4. Möglichkeit

Jede Anweisung bekommt ein Flag, ob sie ausgeführt werden soll, oder nicht. Ist dieses gesetzt, so wird die Formel berechnet, ansonsten nicht. Bei dieser Variante werden die Anweisungen **beider** Zweige in den Charaktergraph eingetragen. Die IF-Anweisung **muß** vor den Anweisungen in den Zweigen berechnet werden, um in den Zweiganweisungen das Flag richtig zu setzen.

Diese Variante ist ebenfalls konsistent in der Berechnung, denn die Anweisungen werden nur einmal in den Charaktergraph 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 Nachteil ist, daß auf einmal Formeln mit angeschaut werden, die im Moment nicht benötigt werden, und eventuell weitere

Berechnungen miteinfließen 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, wie einem Aktivierungsflag 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 inaktiv gesetzt werden und in dem true-Block müssen alle Anweisungen aktiviert werden. Dann muß überprüft werden, ob weitere IF-Anweisungen aktiv geworden sind; in diesem Fall, 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.

4.5 Implementierung der Berechnungen

Wird ein Objekt mit dem Basisobjekt verknüpft (LINK-Kommando) werden die Anweisungen des Objektes „aktiv“. Dieses „Aktiv schalten“ geschieht in der Methode checkBase() der Klasse RpdObject. Hier wird geprüft, ob das Objekt ein Vaterobjekt mit Verbindung zum Basisobjekt besitzt. Falls ja, wird das Objekt entsprechend initialisiert:

Aus allen Typen, aus denen das Objekt besteht, werden die zugehörigen Anweisungen initialisiert. Dazu wird eine Liste mit StmtHandles angelegt, die eine Verknüpfung des aktuellen Objektes mit einer Anweisung darstellen. Grundsätzlich werden alle Berechnungen über StmtHandles erledigt.

Wie schon in Kapitel 4.4 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 Berechnungsliste hinzu, und berechnet dann das Attribut anhand der Berechnungsliste komplett neu.

4.5.1 CalcHandler

Der CalcHandler übernimmt die Ausführung der Berechnungen. Er verwaltet eine Liste mit CalcObject Objekten. Ein CalcObject stellt eine Berechnung dar, die durchgeführt werden soll. So gibt es zum Beispiel die Klasse CalcAttr, die für die Neuberechnung eines Attributs sorgt. Ein CalcObject implementiert das Java Comparable 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 durchgeführt:

Jedes Attribut hat eine Liste von Attributen, von denen es abhängt. Die Erstellung dieser Liste ist in dem nächsten Kapitel beschrieben. Ist ein Attribut in der Abhängigkeitsliste eines anderen Attributs, so muß dieses vorher dem anderen Attribut berechnet werden.

Des weiteren enthält der CalcHandler eine Liste, in die CalcObjecte kommen, die schon ausgeführt wurden, die performedList. Wird ein neues 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 früher 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 das Schaubild in Abbildung 4.5:

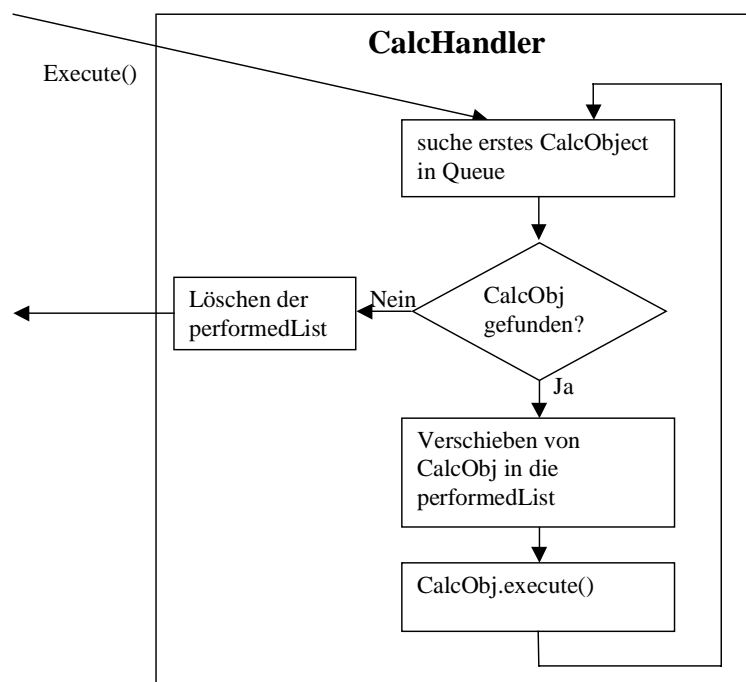


Abbildung 4.5: Arbeitsweise des CalcHandler

4.5.2 Berechnung der Attributen

Jedes Attribut verwaltet drei Listen, die für die Berechnung relevant sind: Die Berechnungsliste, die Triggerliste und die Abhängigkeitsmenge. Die Berechnungsliste ist dabei eine sortierte Liste, die StmtHandles in einer bestimmten Reihenfolge enthält. Die Sortierung erfolgt nach folgenden Kriterien, und in der angegebenen Reihenfolge:

1. Priorität der Anweisungen (z. B. SET vor ASSERT)
2. Position des Objektes im Baum (Ebenensuche, Tiefensuche)
3. Position des Typs, von denen das Objekt abgeleitet wurde
4. Reihenfolge beim Einlesen (ID der Anweisungen)

Der Programmcode in StmtHandle sieht folgendermaßen aus:

```

/**
 * Compare the StmtHandle with another. If the current Stmt should be
 * executed before the given one, a negative number is returned.
 */
public int compareTo(Object o) {
    StmtHandle sh2 = ((StmtHandle)o);
    Stmt st1 = getStatement();
    Stmt st2 = sh2.getStatement();
    // Step 1: check the priority of the statement.
    if (st1.getPriority() < st2.getPriority()) return -1;
    if (st1.getPriority() > st2.getPriority()) return 1;
    // priority is equal. Now compare the objects.
    int res=0;
    RpdLObject ro1 = getObject();
    RpdLObject ro2 = sh2.getObject();
    // Step 2: check the position of the object in the tree.
    if (ro1 != ro2) { // this can speedup things
        ObjectOrder oo = ro1.getMain().getObjectOrder();
        res = oo.compare(ro1, ro2);
    }
    if (res != 0) return res;
    // hmm. same object, same priority.
    // Now, if the statement is derived from the same
    // entity (Object or type) then take the order from
    // the type-list of this object. Else look at the order
    // of the parenttypes of the current object.
    Entity e1 = st1.getDerivedEntity();
    Entity e2 = st2.getDerivedEntity();
    // Step 3: check the derived entity,
    // the order of the types in the object.
    if (e1 != e2) { // this can speedup things
        try {
            res = ro1.compareEntities(e1, e2);
        } catch (RpdLException re) {
            // errors should not happen here.
            // They are internal errors then.
            // But display them anyway
            ro1.getMain().addError(re);
            throw new RuntimeException("internal error: "
                + re.getMessage());
        }
    }
    if (res != 0) return res;
    // Step 4: sort in the order they were created.
    int id1 = st1.getID();
    int id2 = st2.getID();
    return (id1<id2)?-1:(id1==id2?0:1);
}

```

Wird eine Anweisung der Berechnungsliste eines Attributs hinzugefügt, so erfolgt automatisch ein Eintrag in den CalcHandler für eine Neuberechnung dieses Attributs.

Des weiteren erfolgt die statische Überprüfung auf Zyklen während der ersten Berechnung, und die Prüfung von Attributen mit dem REQUIRED-Flag.

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

Beispiel:

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

Damit sind die Abhängigkeiten der Attribute folgende: $B_d = \{A\}$, $C_d = \{A, B\}$

Nach dem Kommando

3. $B = B + D$

sieht das ganze so aus (lokale Zyklen kommen nicht in die Liste):

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

Die Implementierung für die Erstellung der Mengen ist folgende:

1. Wird eine Anweisung hinzugefügt, so wird aus der Anweisung die Liste der Abhängigkeiten (getDependencies()) erzeugt. Dies ist nur die lokale Menge der abhängigen Attribute. Diese Menge ist nicht rekursiv aufgebaut!
2. Von dieser Menge wird das aktuelle Attribut entfernt (lokale Zyklen sind erlaubt!).
3. Die Menge wird dem aktuellen Attribut mittels addDependencies() hinzugefügt:
4. addDependencies(): Es wird geprüft, ob das aktuelle Attribut sich in der Menge der neuen Attribute befindet.
5. Die Menge der neuen Attribute wird der Menge des Attributs hinzugefügt.
6. Die Menge der neuen Attribute wird an alle Attribute, die sich aus den Ziel Attributen der Anweisungen der Triggerliste ergeben hinzugefügt: Dazu wird addDependencies() von jedem Attribut aufgerufen, und bei Schritt 4 weitergemacht.

In dem Beispiel oben bei Anweisung (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. $B \notin 3_d$: kein Zyklus!
5. $B_d = B_d \cup 3_d = \{A, D\}$
6. Menge der zu propagierenden Attribute: Statement (2) $C = 3 * B$: Weitere Attribute sind $\{C\}$, also $C.addDependencies(3_d)$:
7. $C \notin 3_d$: Kein Zyklus!
8. $C_d = C_d \cup 3_d = \{A, B, D\}$
9. Triggerliste von C ist leer: fertig.

4.6 Einfügen von Objekten

Wie schon in Kapitel 3.11 beschrieben gibt es verschiedene Möglichkeiten den Baum abzuarbeiten. 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 einem Vergleich nur noch die Position in der Liste verglichen. Die Liste wird neu aufgebaut, sobald ein neues Objekt hinzukommt.

Die zugehörigen Klassen befinden sich in dem Package `de.mutschler.rpd.tree`. Abbildung 4.6 zeigt die Klassenhierarchie dieser Package:

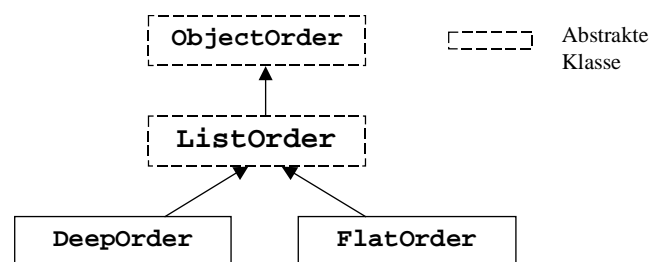


Abbildung 4.6: Klassenhierarchie von `ObjectOrder`

Wobei `ListOrder` die Verwaltung der Liste usw. erledigt, während `DeepOrder` und `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();) {
            Rpd1Object ro = (Rpd1Object)it.next();
            // add it recursively (!)
            addObject(list,ro);
        }
    }
}

```

4.7 Performance

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

Als Eingabe dient die Datei McRathgar.rpd1, die in Anhang 7.2.1 zu finden ist.

Getestet wurde das wiederholte Einlesen der Dateien in einer Endlosschleife. Die Ausgabe der benötigten Zeit ist in Millisekunden angegeben. 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 sehr stark (~10 Prozent langsamer)

Abbildung 4.7 zeigt das Laufzeitverhalten mit der Java Virtual Machine von JDK 1.3. Die Ausgabe ist folgende:

```

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

```

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, inklusiv derer, die doppelt berechnet worden wären.
3. Die Gesamtzeit, die für die Berechnung benötigt wurde.

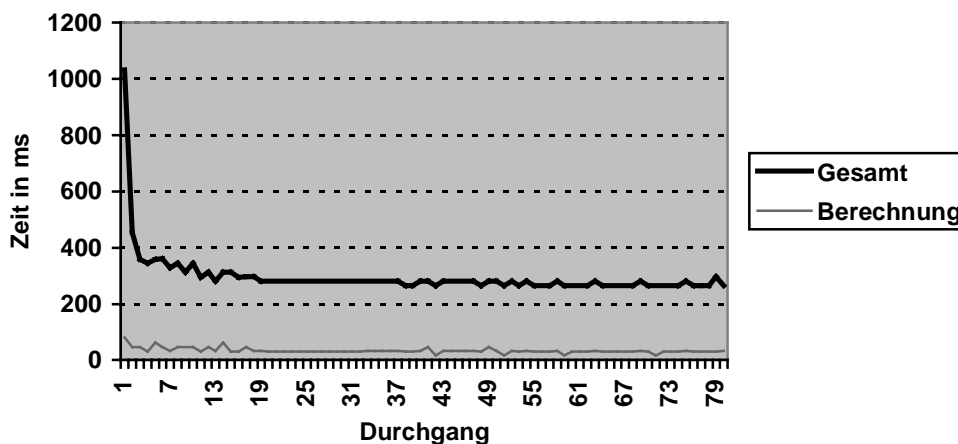


Abbildung 4.7: Laufzeitdiagramm JDK 1.3

Wie man sieht, pendelt sich die Ausführungsgeschwindigkeit bei 281 ms ein. Man kann sehr gut die lokalen Optimierungen von Hotspot verfolgen. Hotspot erstellt Statistiken während der Laufzeit, und optimiert die häufig ausgeführten Stellen im Java Bytecode. Daher wird das Programm mit jedem Durchgang schneller, bis ein 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 von Abbildung 4.8 sind die Laufzeiten der verschiedenen Virtual Machines (VM) des JDK 1.3 im Vergleich aufgeführt:

1. Die Classic VM („java -classic ...“)
2. Die Standard VM (hotspot) ohne Just In Time Compiler („java -xint ...“)
3. Die Server VM von Hotspot 2.0 („java -server ...“)
4. Die Standard VM (hotspot) mit Just In Time Compiler („java -hotspot ...“)

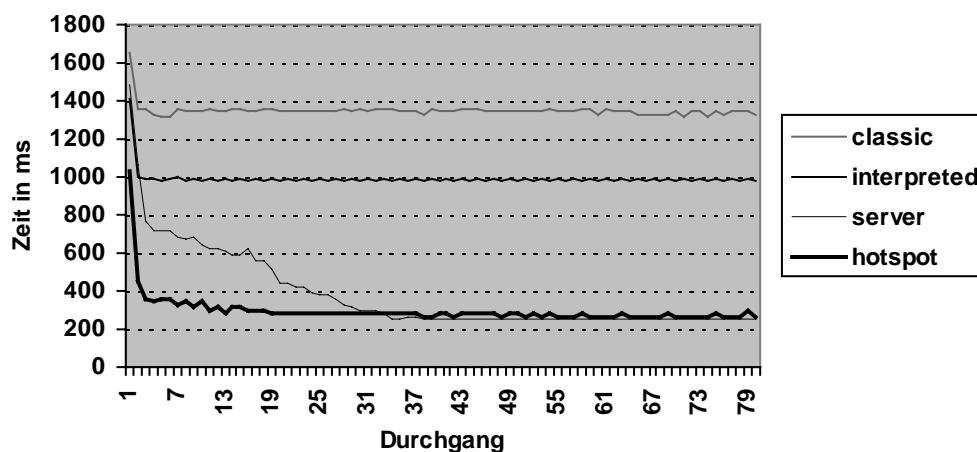


Abbildung 4.8: Laufzeitdiagramm der verschiedenen Virtual Machines

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 4.9):

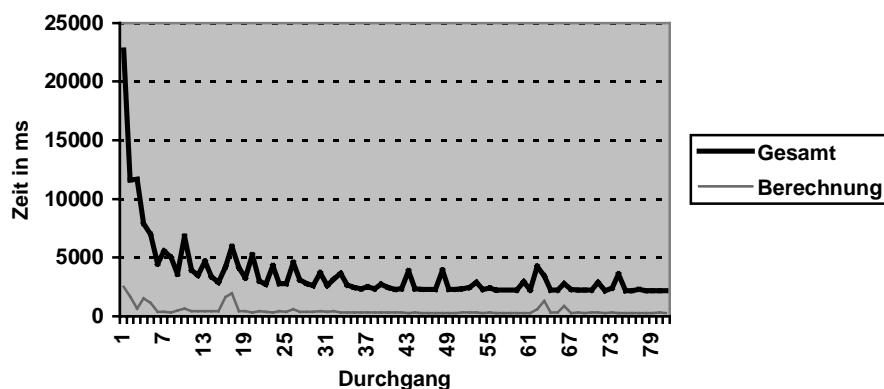


Abbildung 4.9: Laufzeitdiagramm HP 712

Wie man sieht, ist der Rechner ~ 10-mal langsamer als ein Pentium mit 450 Mhz. Der Grund für diesen Test ist folgender: Dies ist der Server (www.mutschler.de) im Internet ist, auf dem die Diplomarbeit veröffentlicht wird.

4.8 Optimierungen

4.8.1 Verzögerung der Berechnungen

Eine Optimierung die, eingebaut ist, ist die Möglichkeit, Berechnungen während des Einlesens nicht sofort auszuführen, sondern erst am Ende. Gleichzeitig werden dadurch keine Berechnungen doppelt ausgeführt. Zum Beispiel würden folgende Kommandos dadurch das Attribut A nur einmal 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.rpd) folgendes:

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

Abbildung 4.10 zeigt das Ergebnis in einem Diagramm.

Wie man sieht, werden insgesamt weniger Attribute neu berechnet (47 weniger, das entspricht ~30 Prozent). Die Ausführungszeit ist dagegen gleich geblieben. Das liegt daran, daß die Größe der Berechnungslisten eine Rolle spielt, und bei dem Beispielcharakter im wesentlichen neue Berechnungslisten erstellt werden, weniger Modifikationen an Attributen, die weitere Neuberechnungen von Attributen (über die Triggerlisten) nach sich führen würden.

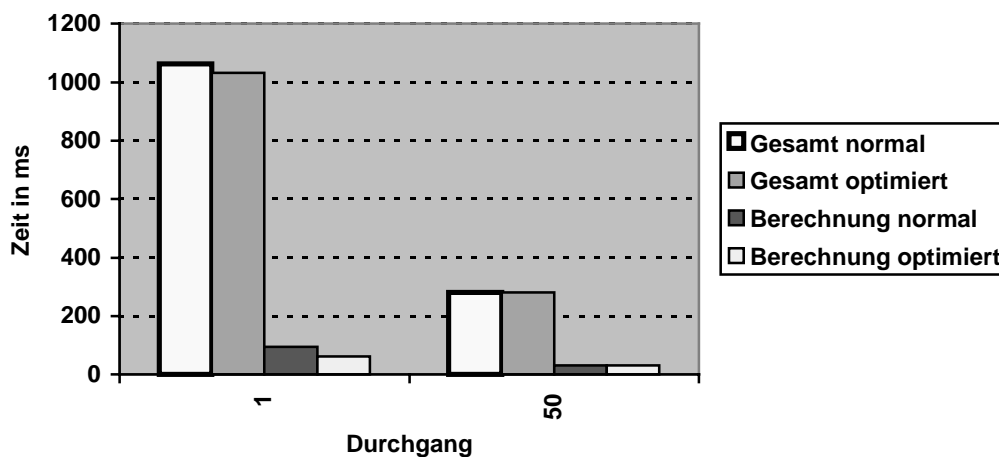


Abbildung 4.10: Laufzeitvergleich mit Berechnungsoptimierung (1)

Fügt man 10-mal folgende Anweisungen an den Beispielcharakter hinzu, so sieht man, daß die Optimierung doch Vorteile bringt:

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

Ohne Optimierung:

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

Mit Optimierung:

Time to execute: #01: 1078ms calcs: 105/272/78ms
 Time to execute: #50: 297ms calcs: 105/272/32ms

Abbildung 4.11 zeigt das Ergebnis im Vergleich.

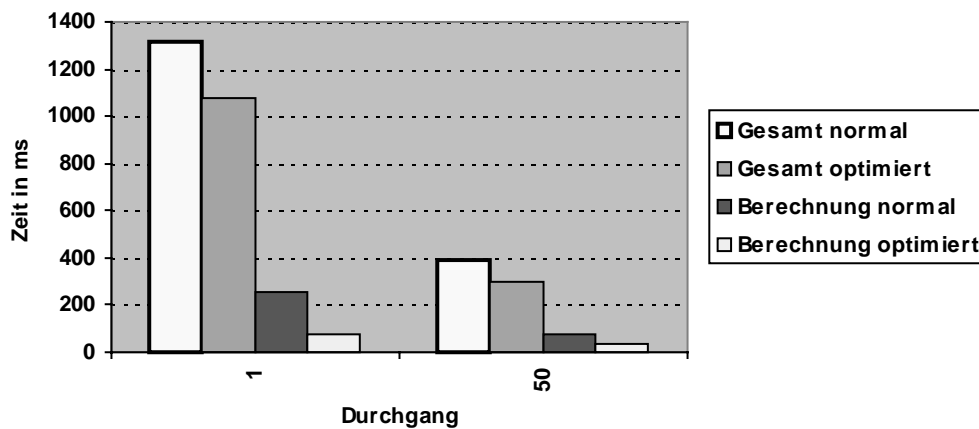


Abbildung 4.11: Laufzeitvergleich mit Berechnungsoptimierung (2)

4.8.2 Caching der Eingabedaten

Aus den Laufzeiten ist deutlich erkennbar, daß die meiste Zeit bei dem Einlesen der Daten verbraucht wird, und nicht bei der Berechnung selbst. Daher macht es Sinn, die Eingabe zu optimieren. Da die Eingabe über Kommandos erfolgt, und diese über eine einzige Methode an das eigentliche RPDL-System übergeben werden (RpdParser.addCommand()) kann an diesem Punkt angesetzt werden, um die von dem Parser gelieferten Kommandos zwischenspeichern. Es bieten sich dafür die Serialisierungsmöglichkeiten von Java an, mit denen man Objekte laden und speichern kann. Siehe Java Dokumentation der Klassen „java.io.Serializable“ und „java.io.ObjectInputStream“.

Das Erzeugen der Cache-Datei(en) dauert es etwas länger:

Time to execute: #01: 1500ms calcs: 105/252/63ms

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

Time to execute: #01: 1219ms calcs: 105/252/62ms

Time to execute: #50: 500ms calcs: 105/252/32ms

Abbildung 4.12 zeigt das Ergebnis im Vergleich.

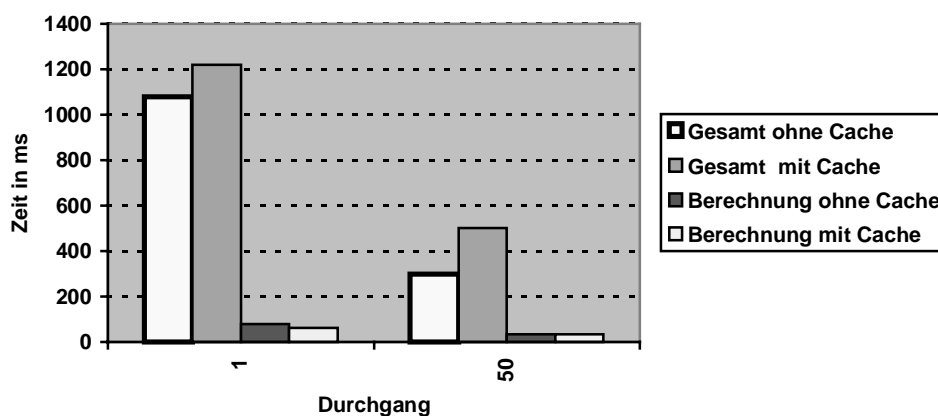


Abbildung 4.12: Laufzeitvergleich mit Caching

Das Einlesen ist dadurch nur halb so schnell wie mit Lex/Yacc (500ms \leftrightarrow 281ms)! Diese Optimierung ist also nicht sinnvoll. Deshalb wurde 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 den Lex/Yacc Code optimieren, um dadurch eine bessere Performance zu erreichen. Dies ist allerdings nicht Ziel dieser Diplomarbeit.

Eine Möglichkeit zum Caching sollte jedoch in Erwägung gezogen werden:

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

4.8.3 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 4.4, "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 ungefähr 22 kB 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 Anweisungen:
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 Reihenfolge dargestellt werden.

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

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

Nun folgt die Liste der Attribute sowie ihre Berechnungs- und Triggerlisten. Wie man bei dem Attribut GG sieht, können hier auch Anweisungen von anderen Objekten stehen. Die zweite Anweisung der Berechnungsliste zeigt: Diese Anweisung stammt 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.

4.8.4 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, der verbreitetste Standard, in Frage. Allerdings ist die Layout-Erstellung bei Servlets nicht so einfach, und deshalb gibt es Java Server Pages (JSP) [19]. Diese basieren auf Servlets und erlauben es, den Java-Code in die Seite einzubinden.

Als JSP-Engine existiert zum Beispiel Resin [21], bei der schon ein HTTP Server integriert ist.

Im nächsten Kapitel ist eine Ausgabe des Beispielcharakters McRathgar.rpd über eine JSP Seite zu sehen.

4.8.5 HTML Ausgabe (Beispiel)

Zum Vergleich siehe auch den original Charakterbogen in Anhang 7.1.

GURPS Charakterbogen von McRathgar

| | | | | | |
|-----------------------------|---------------------|------------------------------|---------|----------------------------|------------------------------|
| ST: | 12 | Erschöpfung: | | Vorteile, Nachteile | |
| GE: | 13 | | 12 | CP | |
| IQ: | 12 | Grundschaden: | | 10 | Lesen und Schreiben |
| KO: | 11 | Schw: | 1W6 + 2 | 22 | Magiebegabung (nur Feuer) +3 |
| | | Stoß: | 1W6 - 1 | 15 | Kampfrelexe |
| Bewegung: | Grundgeschw: | Bewegung: | | -10 | Rechtschaffenheit |
| | 6 | 6 | | -10 | Ehrenkodex |
| Belastung: | | Passive Verteidigung: | | -5 | Pyromanie |
| Keine (0): | 12 | Gesamt: | 0 | -5 | Leichter Schlaf |
| Gering(1): | 24 | Schadensresistenz: | | -5 | Pflichtgefühl |
| Mittel (2): | 36 | Gesamt: | 0 | -1 | Vorsichtig |
| Stark(3): | 72 | | | -1 | Mag nichts Hochprozentiges |
| Extrem(4): | 120 | | | -1 | Mag keinen Fisch essen |
| Aktive Verteidigung: | | | | Fertigkeiten: | |
| Ausweichen: | Pariieren: | Abblocken: | | CP | FW |
| 6 | 0 | 0 | | 2 | Tarnen 13 |
| =BW | =Waffe/2 | =Schild/2 | | 8 | Breitschwert 15 |
| | | | | 0.5 | Schnellladen 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 | | | | | | | | |
| 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 | | | | | | | | | |
| | | CP | | | | | | | | | | |
| Attribute: | | 80 | | | | | | | | | | |
| Vorteile: | | 9 | | | | | | | | | | |
| Fertigkeiten: | | 11.5 | | | | | | | | | | |
| Magie: | | 4 | | | | | | | | | | |
| 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 Schmied beendet. Auf dem Fest nach 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 zu finden ist, einen Dudelsack zu holen. Dummerweise kommt kein Einheimischer ohne Schutz in die Burg, da es dort spukt. Deshalb sollen die Fremden McRathgar helfen.

5 Glossar

Attribut

In einem Attribut werden verschiedene Werte gespeichert. Attribute werden in Objekten angelegt, um beispielsweise sein Gewicht, seine Kosten, etc. zu speichern.

Basisobjekt

Mit Basisobjekt wird das Objekt bezeichnet, mit dem der Charakterbaum anfängt. Siehe Kapitel 3.13.

Basistyp

Ein Objekt, das aus dem Basistyp gebildet wird, ist das Basisobjekt. Es kann nur einen einzigen Basistyp geben, der von dem BASE-Kommando festgelegt wird. Siehe Kapitel 3.13.

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.

Charaktergraph

Ein Charakter in RPDL besteht aus einzelnen Objekten. Diese Objekte sind wiederum in einem Graphen verbunden, und bilden so den eigentlichen Charakter. Der Graph ist ein gerichteter azyklischer Graph. Siehe Kapitel 3.6 für eine Übersicht, und Kapitel 3.13 für die eigentliche Erstellung des Graphen.

Charakterpunkte

Um einen Charakter zu generieren, gibt es bei vielen Rollenspielen die Möglichkeit eine gewisse Anzahl von Punkten auf die einzelnen Fertigkeiten zu verteilen. Diese Punkte heißen Charakterpunkte.

EBNF

Formale Schreibweise für eine Syntax.

Geschichtsführung

Die Geschichtsführung dient dem aufschreiben der Geschichte, bzw. Historie, was den Charakteren in der fiktiven Welt passiert. Dies kann zum Beispiel auch die Form eines Tagebuchs annehmen, wenn man aus der Perspektive eines einzigen Charakters berichtet wird.

GURPS®

Ein Rollenspielsystem. [1]

Harnmaster®

Ein Rollenspielsystem. [5]

Lexer

Ein Lexer erstellt ein Programm, das Zeichen einliest, und sie in Symbole gliedert. Diese werden dann an der Parser (Yacc) zur weiteren Analyse geliefert.

Mana

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

Midgard®

Ein Rollenspielsystem. [8]

Objekt

Ein Objekt ist ein Element aus dem der Charakter in RPDL erstellt wird. Zusammen mit den Verknüpfungen der Objekte zu dem Charaktergraphen bilden sie den eigentlichen Charakter. Siehe Kapitel 3.10.

Package

Ein Package ist eine logische Zusammenfassung von Klassen in Java. Die Packages bilden eine Baumstruktur. Z. B. „java.util“, „java.util.security“.

Spielleiter

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

Triggerliste

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

Typ

Objekte bestehen aus einem oder mehreren Typen. Ein Typ entspricht meist einer Eigenschaft eines Objektes. Siehe Kapitel 3.9 für eine genauere Beschreibung von Typen.

Vaterobjekt

Objekte in dem Charaktergraphen haben (mit Ausnahme des Basisobjekts) immer ein Vaterobjekt. Das Vaterobjekt ist das dem aktuellem Objekt übergeordnete Objekt.

Yacc

Yacc (Yet Another Compiler Compiler), ist ein Programm, das aus einer EBNF-Syntax einen Parser erstellt, der die Grammatik erkennt und verarbeitet. Üblicherweise wird ein Lexer für die Eingabe benötigt.

6 Literatur

- [1] Steve Jackson, GURPS® Basisbuch, Pegasus Press, 1994
- [2] Steve Jackson, GURPS® Magie, Pegasus Press, 1995
- [3] David Cook & Steve Winter & Jon Pickens, AD&D Spielerset Charakterband, AMIGO Spiel + Freizeit GmbH (Original von TSR, 1987), 1995
- [4] J. R. R. Tolkien, Der Herr der Ringe, Klett Cotta Verlag, 1972
- [5] N. Robin Crosby, Harnmaster, Columbia Games Inc., 1991
- [6] Shadowrun second edition, FASA Corporation, 1992
- [7] Space Master - Player Book, Iron Crown Enterprise, 1992
- [8] Midgard: Das Abenteuer beginnt Box, Pegasus Press, <http://www.pegasus.de>
- [9] M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10, 1985
- [10] JAVA Homepage: <http://www.java.sun.com>
- [11] ISO Language Code ISO 639:
<http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>
- [12] David Flanagan, Java in a nutshell (3rd edition), O'Reilly, 1999
- [13] Guido Krüger, Go To Java 2, Addison-Wesley, 1999
- [14] JFlex Homepage: <http://www.jflex.de>
- [15] CUP Homepage: <http://www.cs.princeton.edu/~appel/modern/java/CUP>
- [16] A. V. Aho & R. Sethi & J. D. Ullman, Compilerbau 1 + 2, Addison-Wesley, 1988
- [17] Skript "Konzepte von Programmiersprachen" WS 1998/1999, Prof. Dr. E. Plödereder, Universität Stuttgart, 1998
- [18] Robert Sedgewick, Algorithmen in C++, Addison Wesley, 1992
- [19] JSP Homepage: <http://www.java.sun.com/products/jsp>
- [20] Volker Turau, Java Server Pages, dpunkt Verlag, 2000
- [21] Resin Homepage: <http://www.caucho.com>

7 Anhang

7.1 Beispiel eines Charakterbogens

Ein GURPS® Charakterbogen sieht zum Beispiel so aus (Abbildung 7.1):

GURPS CHARAKTERBOGEN

Name: Der Rückstom Spieler

Aussehen: 1,65 m, 55 kg, helle Haut, dunkle Haar & Augen

Charaktergeschichte: Straßenkind, Dieb

Colormaning: Rollentyp:

Werte CP: Gesamt-CP: 100

Attribute

CP -5 **ST** 8 Erschöpfung

CP 60 **GE** 15 Grundschaden Schw: IW-2

CP 20 **IQ** 12 Stoß: IW-3

CP 20 **KO** 12 Treffer

BEWEGUNG Grundgeschwindigkeit 6,75 BW 6

Belastung: Passive Vorteil:

Aktive Verteidigung:

Schadensresistenz:

Vorteile, Nachteile und Marotten:

Waffen und Besitztümer

| Gegenstand | Typ | Höhe | FW | S | kg | Schaden |
|----------------|-----|------|----|----|------|---------|
| Kl. Messer | sch | IW-5 | I7 | 30 | 0,25 | |
| Dolch | st | IQ-4 | I7 | 20 | 0,25 | |
| Diebeswerkzeug | | | | 30 | - | |
| Kleidung | | | | 10 | 0,5 | |
| Schuhe | | | | 40 | 1 | |
| Lederjacke | | | | 50 | 2 | |
| Ring | | | | 20 | - | |
| Silberdrain | | | | 10 | - | |

Gesamt: 2105 3,879kg

Waffenreichweite

| Waffe | SS | ZG | 1/2 S | Max |
|--------|----|----|-------|-----|
| Messer | II | 0 | 3 | 8 |
| Dolch | II | 0 | 3 | 8 |

Attribute: CP: 85

Nachteile: CP: -40

Marotten: CP: -5

Fertigkeiten: CP: -25

Gesamt-CP: 100

Abbildung 7.1: GURPS® Beispielcharakterbogen

7.2 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.

7.2.1 McRathgar.rpd

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

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

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.rpd";

// now, add the advantages/disadvantages to the character
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 AdvCombatreflexes;
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 CreateFire;
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;}
*/

```

7.2.2 GURPS_base.rpd

```

/*
 * 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.rpd1";

// define the base-character object
CREATE TYPE Character
{
    REQUIRED ATTRIBUTE Name;
    // These are the number of CharacterPoints a Character 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 character

    // 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 could 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 {}

```

7.2.3 GURPS_tables.rpdI

```

/*
 * 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;
}
```

7.2.4 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;
```

7.2.5 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;
}
```

7.2.6 GURPS equip_skills.rpd

```

/*
 * 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}Schnellladen";
    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);
}

```

7.2.7 GURPS equip_weapons.rpd

```

/*
 * 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;
}

```

7.2.8 GURPS_magic.rpd

```

/*
 * 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;
}

```

7.3 Erklärung

Ich versichere, daß ich diese Arbeit selbstständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe.

(Michael Mutschler)