Fortgeschrittenenpraktikum

Grafisches Java-Oz Interface

vorgelegt von:
Martin Homik

Betreuer:
Christian Schulte

Lehrstuhl:
Prof. Dr. G. Smolka
Universität des Saarlandes

6. März 2002

Inhaltsverzeichnis

Kapitel 1

Allgemeines

1.1 Aufgabenstellung

Wie können zwei Prozesse untereinander Daten austauschen, wobei einer der Prozesse Oz-Programme ausführt und der andere Java-Programme? Dabei sind folgende Punkte wichtig:

- Der Entwurf einer Schnittstelle, die eine lose Kopplung zwischen Java-Prozeß und Oz-Prozeß erlaubt.
- Die Schnittstelle soll einfach sein, sowohl in Design, Implementierung, als auch in der Benutzung.
- Die Schnittstelle soll leistungsfähig sein, sie soll eine gute Abbildung von Oz Datenstrukturen auf Java Datenstrukturen erlauben. Das Austauschen von reinen Zeichenketten ist zu primitiv, gewünscht ist die Kommunikation von zum Beispiel Objekten, Arrays, etc.
- Aufbauend auf dieser Schnittstelle soll eine objektorientierte Bibliothek fürOz entworfen und implementiert werden, die die graphische Funktionalität von Java (AWT oder Swing Toolkit) in Oz verfügbar macht. Dabei ist angestrebt, die Schnittstelle so transparent wie möglich zu machen, so daß ein Oz-Benutzer nicht notwendigerweise wissen muß, daß Java irgendwie involviert ist.
- Entwurf und Implemetierung einer graphischen Applikation, die zeigt, daß die Schnittstelle hinreichend mächtig und effizient ist.
- Entwurf und Implementierung einer auf Java-Applets basierenden Bibliothek, die es erlaubt Oz-Applets in einem Web-browser auszuführen.

1.2 Implementierungsstand vom 6. März 2002

Dieses Projekt wurde ursprünglich im März 1998 im Rahmen eines Fortgeschrittenenpraktikums gestellt. Das Praktikum war offiziell im Oktober des selben Jahres beendet. Das Interface lief wie gewünscht. In der Zwischenzeit wurde die Spezifikation von Objekten und Klassen geändert, so daß die ursprüngliche Version neu entwickelt werden mußte, bedingt durch die Tatsache, daß das Kreieren von transparenten Klassen zur Laufzeit der Hauptbestandteil des Projektes ist. Seitdem hat es Erweiterungen gegeben, und eine realistische Abschätzung der Nutzbarkeit ist nun möglich.

Der Zustand der Implementierung des Projektes unterscheidet sich deutlich von der Aufgabenstellung. Einige Punkte sind erfüllt, andere sind noch für die Zukunft geplant. So will ich an dieser Stelle lediglich die Besonderheiten dieses Projektes, die die jetzige Projektimplementierung bereitstellt, aufzählen.

Grundlage für das Interface ist eine funktionierende Kommunikation, die hier durch ein dynamisches Client-Server Modell und ein angepaßtes, einfaches Protokoll verwirklich wurde. Diese Kommunikationsstruktur kann zu jedem beliebigen Zeitpunkt gestartet und beendet werden.

Das Projekt ist hauptsächlich auf die Nutzung von Java AWT und Java Swing ausgerichtet. So ist der wichtigste Punkt der Zugriff auf Klassen und Objekte in Java. Diese Interkommunikation zwischen Oz und Java ist völlig transparent. Oz baut zur Laufzeit hierarchisch eigene Klassen, die den AWT und Swing Klassen entsprechen. Somit kann der Anwender grafische Java Elemente in Oz programmieren, ohne zu wissen, daß ein Java Prozeß im Hintergrund läuft.

Das Generieren von Klassen zur Laufzeit ist ein wichtiger Punkt. In früheren Stadien des Projektes, wurden alle Klassen generiert sobald Oz gestartet wurde. Nun ist es so, daß alle Klassen direkt zu Beginn bekannt sind, diese aber durch die Lazy-Funktionalität erst bei Instanziierung eines Objektes generiert werden. Der Vorteil liegt auf der Hand: weniger Speicherverbrauch und weniger Zeitverbrauch beim Start von Oz.

Die Transparenz geht soweit, daß das System selbst erkennt, ob auf eine statische oder auf eine dynamische Methode zugegriffen wird. Desweiteren findet es selbst heraus, welche überladene Methode der Anwender ausführen möchte.

Das wiederum heißt, daß der Anwender keine Besonderheiten befolgen muß. Er kann sich bei seiner Programmierung an die bestehende Online Dokumentation anlehnen.

Die Anwendung einer Methode stellt einen weiteren bemerkenswerten Gesichtspunkt dar. Der Anwender kann selbst bestimmen, ob er ein Ergebnis zurückerwartet. Somit können deklarierte Variablen, die nur ein Ergebnis abfangen, es aber nicht bearbeiten, gespart werden.

Natürlich kann man auch auf Felder einer Klasse zugreifen. Hierzu wurden zusätzliche Methoden in die OzKlasse eingefügt. Das Interface kennt sämtliche primitiven Java-Typen, die natürlich auch noch als Methodenparameter auftreten.

Die schönste Eigenschaft dieses Projektes im Hinblick auf AWT und Swing ist die Möglichkeit der Interaktivität, also der Kommunikation durch Ereignisse mit dem Endanwender. Hier stellt das System ein komplettes AWT Event-Handling zur Verfügung. Dabei erkennt das System selbständig, welche Ereignisse der Anwender aufgezeichnet haben möchte. Die restlichen Ereignisse werden unterschlagen. In Zukunft sollen noch Event-Listener von Swing bereitgestellt werden.

Im Zusammenhang mit Event-Handling muß man sich speziell mit PaintEvents auseinandersetzen, so daß in Oz geschriebene paint oder update Methoden ausgeführt werden können. In diesem Bereich treten allerdings erhebliche Mängel auf, die auf die Architektur von Java zurückzuführen sind.

Neben dem Event-Handling spielt das Exception-Handling eine weitere wichtige Rolle in der Interaktivität. Hier reagiert der Anwender auf Systemfehler, die in Java aufgetreten sind und weiter nach Oz geschickt werden.

Basierend auf der allgemeinen Implementierung der Schnittstelle ist es dem Anwender möglich, selbst Klassen in Java zu schreiben und diese in Oz bereitzustellen. Somit ist das Interface beliebig erweiterbar.

1.3 Verschiedene Interface Ansätze

Als ich dir Aufgabe bekam, ein Java-Oz Interface basierend auf Sockets zu schreiben, suchte ich zunächst nach weiteren Ansätzen und ich bewertet diese. Insbesondere achtete ich darauf, ob sich andere Lösungen in meine Aufgabe integrieren lassen, oder ob Ideen übernommen werden können. So will ich nun einige Ansätze, die zwar keinen Einsatz fanden, die aber trotzdem interessant sind, vorstellen:

RMI und CORBA RMI steht für *Remote Message Invocation* und liegt dem Java Paket bei. Mit Hilfe von RMI kann man Objekte auf einem anderen Host kreieren und auf sie zugreifen. Das ganze passiert transparent. Sobald man eine Referenz zurückbekommt, kann man mit ihr direkt auf Objekte zugreifen. Man programmiert also eine Art Agentensystem. RMI ist zu 100% Java, d.h. auf beiden Hostseiten muß Java kommunizieren. Dies hat zur Folge, daß Oz keinen direkten Zugriff auf RMI hat, und daß dieser Ansatz für meine Aufgabenstellung ungeeignet ist.

CORBA steht für *Common Object Request Broker Adapter* und verfolgt ein ähnliches Ziel wie RMI mit dem Unterschied, daß CORBA universeller und komplexer ist. Mit Hilfe von CORBA kann man auf Objekte zugreifen, die in unterschiedlichen, objektorientierten Sprachen implementiert wurden. Somit spart man viel Code, der nicht wiederholt in einer weiteren Sprache auftaucht. Ein weiterer Vorteil ist, daß man selbst bestimmt, in welchem Umfang auf eine Sprache zugegriffen wird. Im besten Fall auf alle Klassen oder Objekte. Die Schnittstelle hierfür wird in einer eigenen Sprache, nämlich IDL (*Interface Definition Language*) geschrieben. Die Nachteile dieses Ansatzes liegen darin, daß

neben Oz und Java nun ein drittes System verlangt wird. Dies steigert die Komplexität und den Speicheraufwand. Desweiteren ist kein CORBA-Oz-Compiler vorhanden, welcher notwendig ist, und somit entworfen werden müßte.

Java Serialization Java Serialization wird intensiv in Verbindung mit RMI verwendet. Es dient dazu, ein Objekt auf einem Host auf ein Objekt auf einem anderen Host abzubilden. Hier werden die Werte, also Felder, direkt übertragen. Der Sinn von Serialization ist die Ergänzung zu RMI. RMI bietet der Gegenseite eine Referenz an, während Serialization einen Schritt weiter geht und das komplette Objekt überträgt. Dies kann zum Beispiel dann sinnvoll sein, wenn oft auf das Objekt zugegriffen oder wenn das Objekt auf einen schnelleren Rechner geladen wird. Allerdings stellen sich folgende Probleme: Allein durch die Notwendigkeit von RMI fällt die Verbindung zu Oz weg, da RMI nur von Java verstanden wird. Außerdem ist es speicherintensiv und nutzlos, jedes Objekt sowohl in Java als auch in Oz zu verwalten.

Jacl Interface Das Jacl Paket besteht aus drei Komponenten: Jacl, TclBlend und Tcl-Studio. Es dient dazu, die Sprachen Java und Tcl miteinander zu vereinigen. Sieht man sich dieses Paket nun genau an, dann stellt man fest, daß Jacl zu 100% in Java geschrieben, und daß es ein reiner Interpreter ist, der Tcl-Skripte als File oder per URL einliest und diese interpretiert. Somit braucht man also noch nicht einmal das Tcl/Tk Paket. Jacl versucht einfach nur beide Sprachen cross-platform zu halten, da Tcl ursprünglich in C geschrieben wurde. Interessant an diesem Paket ist die Verwaltung von Objekten, die in Skripten generiert werden. Aber auch dieser Ansatz ist weitgehend uninteressant, da hier Tcl/Tk von Java „geschluckt" wird. Gesucht ist aber eine Lösung, in der beide Sprachen souverän nebeneinander und miteinander arbeiten.

Jipl Interface Das Jipl Interface ist ein Interface zwischen den Sprachen Java und Prolog. Die Implementation beruht auf dem Java Native Interface. Man kann bequem von Java aus Prolog Programme starten und von Prolog aus auf Java Objekte zugreifen. Wie das genau gemacht wurde verstehe ich nicht, da der Source unvollständig ist.

Java Native Interface Das Java Native Interface ist zwar kompliziert zu handhaben und ungeeignet für verteiltes Programmieren von Objekten, dennoch hat es einen Nutzwert, insbesondere wenn man Plattformunabhängigkeit vernachlässigen muß oder aber nah an der Hardware programmieren will. Das Native Interface ist in erster Linie eine Schnittstelle nach C/C++. Man erhält die Möglichkeit auf einen in einer „native" Sprache geschriebenen und compilierten Code zugreifen zu können. Das funktioniert auch in die Gegenrichtung. Man kann z.B. von C/C++ eine JavaVM starten, Objekte erzeugen und auf ihre Methoden zugreifen.

Das führt zu folgender Überlegung. Die Sprache Oz besitzt ein C-Interface. Über diese Schnittstelle läßt sich bei Bedarf eine JavaVM starten und mit ihr arbeiten. Das würde ein anderes Design der Aufgabe erfordern. Die Socketprogrammierung würde entfallen. Stattdessen würde die in C/C++ gestartete JavaVM als eine Art Server arbeiten. Anweisungen in Oz müßten in entsprechen-

de C-Anweisungen umgewandelt, die über JNI nach Java weitergeleitet werden. Dort werden sie ausgeführt. Der umgekehrte Weg ist ebenfalls möglich.

Es stellt sich dennoch die Frage, inwiefern ein generischer Lösungsanzatz für mein Projekt noch möglich ist, denn der Anwender soll ja über Klassen auf Java zugreifen können. Ich vermute vielmehr, daß diese Klassen per Hand eingetippt werden müßten.

Kapitel 2

Anwendung des Interface

Dieses Kapitel richtet sich an den Programmierer, der AWT- und Swingklassen in Oz nutzen möchte. Er sollte schon mit den Grundsätzen von AWT und Swing vertraut sein, da hier nicht die einzelnen Klassen erklärt werden. Vielmehr wird dargestellt, wie diese Klassen in Oz benutzt werden. Anhand eines kleinen Beispiels werden alle Vorteile, die das Interface zur Verfügung stellt, besprochen.

Wer sich genauer über die Klassen informieren möchte, der sollte in die mitgelieferte Java Online-Dokumentation reinschauen. Wer sich danach richtet, kann nichts falsch machen.

Eine kurze Bemerkung zu Swing. Das Swing Package ist separat installiert. Ein Zugriff auf Swing-Klassen funktioniert daher nicht für Java 1.2.

2.1 Beispiel: HelloWorld

Wie das so in jeder Sprache ist, ist das allererste Beispiel *HelloWorld*. Normalerweise wird das Beispiel so einfach wie möglich gehalten. Dieses erste Oz-Beispiel jedoch ist komplexer, aber trotzdem einfach zu verstehen. Es zeigt folgende Besonderheiten auf, die in den folgenden Abschnitten genauer erläutert werden:

- Objektinstanziierung
- Methodenaufruf
- Exception Handling
- Event Handling

Im Verzeichnis *demos* befinden sich weitere Beispielprogramme, die die Funktionalität dieser Schnittstelle darstellen:

Algorithmus 1 Beispiel: HelloWorld.oz

```
local
    Frame Label Pane OzBorder BorderFact
    OzGridLayout Button1 Button2 Listener CloseEv
    class Sowieso from JOzAdapters.actionListener
        attr state: false
        meth actionPerformed(ActionEvent)
            E P D S
        in
            case @state
            of false then
                try
                    {Label setText(5)}
                    state←true
                catch javaException(exc: E path: P detail: D stack: S) then
                    {Browse javaException(exc: E path: P detail: D stack: S)}
                    {Label setText("Clicked")}
                    state←true
                end
            else
                {Label setText("Click again!")}
                state←false
            end
        end
    end
    class CloseEvent in JOzAdapters.actionListener
        meth actionPerformed(ActionEvent)
            {Java.stop}
        end
    end
in
    Frame = {New Swing.jFrame init("Hello World!")}
    Label = {New Swing.jLabel init("HelloWorld")}
    Pane = {New Swing.jPanel init()}
    Button1 = {New Swing.jButton init("Click me!")}
    Button2 = {New Swing.jButton init("Click me!")}
    Listener = {New Sowieso init()}
    CloseEv = {New CloseEvent init()}
    {Button1 addActionListener(Listener)}
    {Button2 addActionListener(CloseEv)}
    BorderFact = {New Swing.borderFactory init}
    {BorderFact createEmptyBorder(30 30 10 170 result: OzBorder)}
    OzGridLayout = {New Awt.gridLayout init(0 1)}
    {Pane setBorder(OzBorder)}
    {Pane setLayout(OzGridLayout)}
    {Pane add(Label)}
    {Pane add(Button1)}
    {Pane add(Button2)}
    {Frame setContentPane(Pane)}
    {Frame pack()}
    {Frame setVisible(true)}
end
```

helloWorld.oz Dies ist das in Algorithmus 1 abgebildete Programm. Es zeigt die Funktionalität sowohl von AWT als auch von Swing.

helloWorld2.oz Gleiches Beispiel wie oben, aber ohne Event Handling.

helloWorldawt.oz Das gleiche Beispiel, jedoch nur in AWT.

helloWorldawt2.oz Das gleiche Beispiel, jedoch nur in AWT und ohne EventHandling.

accessField.oz Kleines Beispiel zu einem Feldzugriff.

master.oz Ein umfangreicheres Beispiel, welches das Spiel Mastermind implementiert. Hier werden ganz deutlich die Grenzen in bezug auf paint() gezeigt.

master2.oz Gleiches Spiel, wobei das Spiel komplett in Java programmiert ist, und in Oz über eine eigene Schnittstelle bereitgestellt wurde.

2.2 Bekannte Typen

Das Interface kennt alle primitiven Typen von Java. Ausgeschlossen sind zur Zeit Arrays. Zu den primitiven Typen zählen:

Java Typen	Oz Typen
java.lang.Boolean (boolean)	bool
java.lang.Character (char)	string
java.lang.Byte (byte)	JavaClass.byteInt
java.lang.Short (short)	JavaClass.shortInt
java.lang.Integer (int)	integer
java.lang.Long (long)	longInt
java.lang.Float (float)	float
java.lang.Double (double)	JavaClass.doubleFloat
java.lang.Void (void)	unit

Dadurch daß Oz innerhalb der Typen Integer und Float nicht mehr weiter unterscheidet, ist es notwendig weitere Typen, wie aus obiger Tabelle zu ersehen, zur Verfügung zu stellen. Die Notwendigkeit resultiert aus der eindeutigen Signatur einer Java Methode. Wird also ein Parameter des Typs Short als Argument verlangt, dann kann man nicht den Typ Integer schicken. Dies würde zu einer *NoSuchMethodException* führen. Ebenfalls gilt, daß ein zu großer Integerwert, der nach Java geschickt wird, dessen Klasse als Integer deklariert ist, obwohl seine Größe auf eine Klasse des Typs Long schließen läßt, eine *NumberFormatException* auslöst. Somit gilt, daß man sich an die Vorgaben von Java halten muß.

Alle diese zusätzlichen Typen sind in dem Modul *JavaClass* zu finden. Sie haben folgende Methoden:

- init(+Value)
  Konstruktor mit Initialisierungswert. Sollte init nicht benutzt werden, so ist der Standardwert entweder bei 0 oder bei 0.0.

- get(?Value)
  Gibt den Wert zurück.

- set(+Value)
  Setzt den neuen Wert. Als Value kann auch ein String übergeben werden. Dieser wird in den entsprechenden Typen umgewandelt.

- toString(?Value)
  Gibt den Wert als String zurück.

Zwei Dinge fallen besonders auf. Zum einen wird in Java der Typ String nicht als primitiv gesehen. Das Interface aber behandelt Strings primitiv, da es sonst zuviel Overflow gäbe. Strings kann man auch in Oz bequem bearbeiten.

Zum anderen darf man die Begriffe *unit*, *void* und *null* nicht durcheinander bringen. Will man ein unit, dies ist ein Name der für alles Beliebige steht, nach Java übertragen, so wird intern der Wert null der Klasse void geschickt. Für die Rückrichtung gilt dasselbe.

2.3 Objektinstanziierung

Das Instanziieren eines Objektes ist denkbar einfach. Wie nicht anders zu erwarten, wird dieser Vorgang mit *New* eingeleitet, gefolgt von der Klasse, die man instanziiern möchte. Die Klassen befinden sich in Modulen. Zur Zeit stehen folgende Module zur Verfügung:

- AWT
- Swing
- JOzAdapters

Anders als in Oz üblich besitzen diese Klassen einen „Konstruktor". Dieser ist immer die Methode *init*. Mit *init* wird Java der Wunsch nach einem Objekt mitgeteilt. Diesem Konstruktor können, wenn danach verlangt wird, Argumente übergeben werden. Welche Argumente es sind, und in welcher Reihenfolge diese angegeben werden, muß man der Java Online-Dokumentation entnehmen. Es ist wichtig, daß die Reihenfolge eingehalten wird. Das Ergebnis einer Instanziierung ist natürlich ein Objekt.

Da *init* der einzige festgelegte Methodenname ist, kann es vorkommen, daß dadurch eine schon in Java vorhandene Methode mit dem Namen *init* überschrieben wird. Um dennoch auf diese zugreifen zu können, muß man sich mit einem *Static Method Call* weiterhelfen.

Anders als bei Java, instanziiert das Interface auch abstrakte Klassen, wobei diese nur als Proxyklassen in Oz auftreten. Es findet keine Aufforderung an Java zum Instanziieren dieser Klassen statt. Die Klasse wird nach der Übertragung der Klasseninformationen auch in Oz als abstrakt gekennzeichnet. Sinnvoll ist dieser Schritt insbesondere in dem Fall, wenn eine abstrakte Klasse auf statische Methoden oder statische Felder zugreifen möchte. Hierzu braucht man in Oz eine Proxyklasse, über die man überhaupt einen Zugriff bekommt.

Leider ergibt sich durch die Reflection API ein schwerwiegendes Problem. Diese erlaubt nicht die Konstruktion einer eigenen Klasse, indem eine Standardklasse, wie z.B. JComponent, erweitert wird. Entsprechend fällt auch die Instanziierung weg. Eine Subklasse von JComponent wird in Java als Zeichenfläche empfohlen. Diese Möglichkeit weist das Interface nicht auf. Dies liegt daran, daß abstrakte Klassen nicht instanziiert werden dürfen, wohl aber deren Subklassen, sofern keine abstrakten Methoden mehr übrig sind. Sobald der Konstruktor init() aufgerufen wird, wird zwar ein Oz Objekt generiert, aber kein erweitertes Java Objekt.

2.4 Methodenaufruf

Genauso einfach wie die Objektinstanziierung ist die Methodenausführung. Die Syntax ist nach wie vor genau dieselbe. Man muß ein Objekt und eine Methode mit oder ohne Argumente angeben. Wie schon bei der Instanziierung beschrieben, müssen die Argumente in der richtigen Reihenfolge, gemäß der Java Klassendokumentation, angegeben werden. Auch hier gilt, daß keine Features erlaubt sind, bis auf eine Ausnahme. Diese ist für das Rückgabeergebnis reserviert. Mit dem Feature *result* hat der Anwender die Möglichkeit ein Ergebnis einer Methodenausführung aufzunehmen. Ist er jedoch an dem Ergebnis nicht interessiert, so kann man das Feature *result* einfach weglassen. Ein gutes Beispiel hierfür ist folgende Zeile:

Beispiel:

{BorderFact createEmptyBorder(30 30 10 170 result: OzBorder)}

Hier wird eine Methode *createEmptyBorder* mit den Parametern 30, 30, 10 und 170 aufgerufen. Da wir das Ergebnis für später brauchen, wird es in der Variable OzBorder gespeichert. Dies ist nur möglich, weil das Feature *result* angegeben ist. Ohne *result* gibt es keinen Rückgabewert. In diesem Zusammenhang sollte man den folgenden beliebten Fehler, der sich aufgrund der Syntax von Prozeduren, Funktionen oder Java Methodenaufrufen einschleicht, vermeiden.

Beispiel:

OzBorder={BorderFact createEmptyBorder(30 30 10 170)}

Dieselbe Zeile macht ein weiteres Feature deutlich. Es gibt für den Anwender keinen Unterschied, ob die ausgeführte Methode statisch oder dynamisch ist. Ein Problem ergibt sich aber, wenn eine Klasse, wie z.B. BorderFactory in Swing, nur aus statischen Methoden besteht, und somit keinen Konstruktor besitzt. In Java findet der Aufruf einer solchen Methode über die Klasse statt, also zum Beispiel:

Beispiel:

```
JavaBorder=BorderFactory.createEmptyBorder(30, 30, 10, 170);
```

Oz dagegen macht hier keinen Unterschied, da alle Java-Klassen auf eine Oz-Klasse abgebildet werden. Will man nun solch eine statische Methode aufrufen, muß man erst die Klasse instanziieren. Intern wird der Konstruktoraufruf abgefangen und nicht nach Java geschickt. Gleiches gilt für abstrakte Klassen, die statische Methoden beinhalten.

Selbstverständlich kann man auf diese Weise auch auf statische Methoden einer Klasse, die auch nicht-statische Methoden beinhaltet, zugreifen.

Zuletzt noch eine Bemerkung zu überladenen Methoden. Auch in diesem Fall muß sich der Anwender keine Gedanken machen. Das System erkennt selbst an den Typen der Argumente, welche überladene Methode aufgerufen wird. Aus diesem Grund ist es unter anderem auch wichtig, die Reihenfolge der Argumente zu erhalten. Anhand dieser erkennt Java die richtige Methode.

2.5 Zugriff auf Felder

Neben dem Methodenzugriff gibt es auch den Zugriff auf Felder. Dieser ist nur lesend möglich, weil ein guter Programmierstil Methoden zum Ändern von Feldern bereitstellt. Der Zugriff auf Felder geschieht ebenfalls über dafür vorhergesehene Methoden. Jedes Feld hat eine eigene Zugriffsmethode, deren Bezeichnung sich aus dem Namen des Feldes und einem vorangestellten *field_* zusammensetzt. Außerdem sollte wie immer das Feature *result* angegeben werden, damit man das Ergebnis entgegen nehmen kann.

Beispiel:

```
Color={New Awt.color init(0 0 0)}
{Color field_orange(result: Orange)}
```

Wie für den statischen Methodenzugriff, gelten dieselben Bedingungen für statische Felder in normalen als auch in abstrakten Klassen.

Für die Zukunft ist der Zugriff auf statische Felder innerhalb von Interfaces geplant, denn diese sind im Grunde abstrakte Klassen.

2.6 Exception Handling

Bei einer Instanziierung oder bei einer Methodenausführung können auch Fehler auftreten. So kann es passieren, daß der Anwender falsche Argumente angibt, oder daß er eine Methode im falschen Zusammenhang benutzt. Alle diese Fehler werfen eine Exception, die es zu fangen gilt, will man sicher programmieren.

Es gibt drei Arten von Exceptions, die den Anwender auf einen Fehler aufmerksam machen:

- **jozExceptions**

  Syntax: jozException(where: Where what: What det: Detail)

  Diese Ausnahme wird nur geworfen, wenn es Fehler innerhalb des Systems gibt, d.h. Fehler, die der Anwender nicht verursacht. Dies können zum Beispiel falsche Protokolltokens sein. Hierbei besagt das Feature *where*, wo der Fehler aufgetreten ist. *what* gibt Information darüber, in welcher Methode einer Klasse die Ausnahme geworfen wurde, während *det* eine kleine Erläuterung gibt.

- **userJOzExceptions**

  Syntax: userJOzException(where: Where det: Detail)

  Diese Ausnahme wird geworfen, wenn Oz selbst merkt, daß der Anwender einen Fehler gemacht hat. Dies ist zum Beispiel der Fall, wenn der Anwender einen Parameter angibt, der noch gar nicht unterstützt wird. Auch hier beschreibt *where* den Ort der Ausnahme und *det* die Beschreibung zum Fehler. Hier wäre es wünschenswert eine Art Stackaufruf zu ermitteln, um festzustellen, bei welcher Aktion die Ausnahme geworfen wurde.

- **javaExceptions**

  Syntax: javaException(exc: Exc path: Path detail: Detail stack: Stack)

  Diese Ausnahmen sind die wichtigsten, denn sie werden von Java zur Laufzeit geworfen. In *exc* steht der Name (Typ:Atom) der Exception, so wie er in Java bekannt ist. *path* ist das Präfix der Exception, dargestellt als Liste von Atomen, also z.B. [java awt]. *detail* enthält wie bisher eine genauere Beschreibung der Ausnahme. Schließlich steht in *stack* der Aufrufpfad bis zum Zeitpunkt, an dem die Exception geworfen wurde. Dieser Pfad bezieht sich auf die Java Seite. Wünschenswert wäre ein Aufrufstack für Oz.

Wie einfach *javaException* sich ins System integrieren läßt, zeigt das Beispiel. Dort wird versucht, nach einem Ereignis die Methode *setText* in der Klasse *JLabel* mit einem Integerwert als Argument zu applizieren. Das Argument hat somit einen falschen Typ, so daß eine Exception geworfen und somit der Aufruf korrigiert wird.

2.7 Event Handling

Das Anzeigen von grafischen Elementen ist ja schon ganz schön, aber es fehlt die Interaktivität. Diese wird durch das Event Handling bereitgestellt. Um Ereignisse abzufangen, muß man einen sogenannten *Listener*, der nach für ihn bestimmten Ereignisse horcht, schreiben und ihn bei der Komponente, die das Ereignis hervorruft, anmelden.

Das System kennt folgende AWT Listener, die im Modul JOzAdapters zu finden sind:

Awt Listener	Abbildung in Oz
java.awt.event.ActionListener	JOzAdapter.actionListener
java.awt.event.AdjustmentListener	JOzAdapter.adjustmentListener
java.awt.event.ComponentListener	JOzAdapter.componentListener
java.awt.event.ContainerListener	JOzAdapter.containerListener
java.awt.event.FocusListener	JOzAdapter.focusListener
java.awt.event.ItemListener	JOzAdapter.itemListener
java.awt.event.KeyListener	JOzAdapter.keyListener
java.awt.event.MouseListener	JOzAdapter.mouseListener
java.awt.event.MouseMotionListener	JOzAdapter.mouseMotionListener
java.awt.event.TextListener	JOzAdapter.textListener
java.awt.event.WindowListener	JOzAdapter.windowListener

Jeder dieser Listener horcht nach bestimmten Ereignissen. Ist der Anwender an einem bestimmten Ereignis interessiert, muß er eine Klasse, die die oben aufgeführten Listener erweitert, schreiben. Diese Klasse muß vor allem die gewünschten Methoden, die die Listener als Java-Interface bereitstellen, implementieren. Im Klartext heißt dies, daß, wollte der Anwender zum Beispiel einen MouseListener implementieren, der nur die Ereignisse *mouseEntered* und *mouseExited* verfolgen soll, er dann genau diese zwei Methoden, nämlich *mouseEntered* und *mouseExited*, angeben muß. Die restlichen drei Ereignisse, d.h. Methoden, sind nicht relevant und müssen nicht implementiert werden.

Zur Zeit fehlen noch die spezifischen Swing Listener.

2.8 paint() und update()

Es gibt ein Ereignis in Java, welches durch keinen Listener abgefangen wird, nämlich *PaintEvent*. Dieses Ereignis tritt immer dann auf, wenn die Awt Logik feststellt, daß eine Komponente neu gezeichnet, oder zumindest teilweise neu gezeichnet werden muß. Das Interface fängt auf der Java Seite dieses Ereignis ab, wertet aus, ob es sich um einen Aufruf von *paint()* oder *update()* handelt, und ruft die entsperechende Methode der betroffenen Komponente in Oz auf. Gleichzeitig wird das Ereignis in Java weitergegeben, damit standardisierte Kontainer und Komponenten aufgefrischt werden. Der Anwender kann also wie von Java gewohnt die Vorgaben übernehmen. Eine typische Anwendung wird in Algorithmus 2 gezeigt.

Allerdings existiert ein gravierender Unterschied. Wird zum Beispiel ein Frame wieder sichtbar, weil ein darüberliegendes Fenster es vorher verdeckte, so wird nicht an alle Komponenten ein *PaintEvent* geschickt, sondern lediglich an die Topkomponente. Diese berechnet selbst, welche Subkomponenten erneuert werden müssen. Da das Interface keine Möglichkeit hat, dieses selbst zu berechnen, bleibt nichts anderes übrig, als in *paint()* oder *update()* der Topkomponente in Oz alle anderen Subkomponenten seperat aufzurufen. Dies hat zur Folge, daß intensives Zeichnen, zum Beispiel

Algorithmus 2 Beispiel: Nutzung von paint()

```
class MeinBild from Awt.canvas
        ⋮
    meth paint(Graphics)
        Black
    in
        Black = {New Awt.color init(0 0 0(}
        {Graphics setBackground(Black)}
    end
        ⋮
end
```

das Aktualisieren eines Spielbretts, äußerst langsam vonstatten geht. Zum einen liegt das daran, daß auch unnötige Regionen neu gezeichnet werden, aber auch an der Interpretation der einzelnen Befehle zur Laufzeit. Man darf nicht vergessen, daß die Proxyklassen nicht im Bytecode vorliegen. Ich komme zu dem Ergebnis, daß man Zeichnungen tunlichst vermeiden und als Alternative eine Erweiterung schreiben sollte. Für Standardkomponenten von Awt und Swing reicht es jedoch vollkommen aus.

Ein weiteres Problem in bezug auf *paint* und *update* liefert Swing. Swing benutzt nicht, wie in der Dokumentation angegeben, *java.awt.Graphics* zum Zeichnen sondern *com.sun.java.swing.SwingGraphics*, welches die erste Klasse erweitert. Leider ist SwingGraphics weder public, private noch protected. Diese Klasse ist irgendwo intern im Paket definiert und wird auch nur dort instanziiert. Außerhalb dieses Pakets und außerhalb der zugehörigen Komponente ist kein Zugriff darauf möglich, so daß ein Zeichnen auf einem *JPanel* nicht gewährt wird und mit einer *IllegalAccessException* abbricht. Auch hier der Verweis auf eine eigene Erweiterung, um diesem Problem auszuweichen.

2.9 Weitere Probleme und Lösungsvorschläge

Ein anderes Problem liegt darin, daß Java von sich aus bei Bedarf, ähnlich wie bei paint und update, bestimmte Methoden aufruft. So passiert dies auch bei den Methoden *getPreferredSize()*, *getMinSize()* und *getMaxSize()*. Da diese Methoden auf der Java Seite leer sind, und da sie keine Proxies sind, die nach Oz verweisen, kann dem System keine Größe der Komponente übermittelt werden. Eine kleine Abhilfe schafft hier das Kommando *setSize(Dimension)*. Leider sieht das Ergebnis nicht besoonders schön aus. Oft kommt es dazu, daß das Kommando relativ spät ausgeführt wird, was dazu führt, daß die Komponente während des Betriebes ihr Aussehen ändert. In Swing sind neue Methoden in *JComponent* vorhanden, mit denen man auch die minSize, maxSize und preferredSize setzen kann. Auf diese Weise kann man das Problem lösen.

In der Regel möchte man nicht nur einfach Klassen instanziieren, sondern auch Standardklassen erweitern, um an die eigenen Bedürfnisse anzupassen. Das Interface er-

laubt lediglich eine auf Oz eingeschränkte Erweiterung. Das heißt, man kann Klassen beliebig erweitern, aber, und das ist der Punkt, der Zugriff auf mit *protected* markierte Methoden oder Felder ist nicht möglich. Dies liegt daran, daß die Reflection API nur mit ClassByteCode arbeitet, also mit Klassen, die in Java geschrieben und kompiliert wurden. Da wir unsere Klassen in Oz schreiben und nur Standardklassen von Java instanziieren lassen und keine Klasse kompilieren, kann Java auch nichts von der Erweiterung einer Standardklasse wissen, und so wird der Zugriff verweigert. *protected* heißt schließlich: Als *protected* deklarierte Attribute sind zugreifbar für Unterklassen, werden an diese vererbt und sie sind zugreifbar für Code im gleichen Paket.

Das Fazit dieses Kapitels muß jedoch sein, daß wenn man selbst Zeichnen möchte, man die Komponente in Java schreibt und über ein Modul Oz zur Verfügung stellt.

2.10 Erweitern um eigene Klassen

Die Konstruktion der Schnittstelle erlaubt die eigenständige Erweiterung der Schnittstelle. Hierzu müssen bereits kompilierte Java Klassen vorliegen und über den CLASSPATH bekannt sein. Ist dies gegeben, so reicht folgende Struktur eines Oz Moduls vollkommen aus, um eigene Klassen in das Oz System einzubinden. Algorithmus 3 zeigt ein Beispiel für das Package java.lang.

Algorithmus 3 Beispiel: Oz Package

```
declare
   Lang
in
local
   fun {MakeMaker Prefix}
      fun lazy {$ Suffix}
         {JavaClass.newClass {VirtualString.toString Prefix#Suffix}}
      end
   end
   SM = {MakeMaker 'java.lang.'}
in
   Lang = {Record.map
             lang(object: 'Object'
                  string: 'String') SM}
end
```

Dieses Modul muß schließlich mit `\insert` in den Sourcecode eingebunden werden, so daß man mit *Lang.string* die Klasse String instanziieren kann.[1]

Im Zusammenhang mit den in diesem Kapitel genannten Swing Problemen und der langsamen Bilddarstellung von Zeichnungen, ist das Erweitern um eigene Klassen eine sehr effiziente Alternative. Der Anwender programmiert sein aufwendiges GUI in Java

[1] String wird in Wirklichkeit nicht als Klasse zur Verfügung gestellt.

und bindet die Klasse in Oz ein. Alle Methoden, die als public deklariert sind, sind für Oz zugänglich, und so kann Oz die notwendigen Berechnungen der Applikation führen und Jva die nebensächlichen darstellungswünsche mitteilen.

Hat man eine ganze Sammlung von Klassen, die man bereitstellen will, so ist es sinnvoll ein Package zu erstellen. Dieses kann man zu einem JAR komprimieren. Ob als normales Package oder als JAR ist egal, wichtig ist, daß diese Klassen im Pfad liegen.

Kapitel 3

Implementierung

Für den interessierten Programmierer ist die Implementierung der Schnittstelle wesentlich wichtiger als die Nutzung. So soll in diesem Kapitel das Design genau betrachtet werden. Besonderes Augenmerk soll auf die Implementierung der Besonderheiten, die sich dem Anwender darbieten, geworfen werden.

Im ersten Abschnitt wird das Protokoll vorgestellt, welches festlegt, wie die Kommunikation ablaufen soll.

Nachdem bekannt ist, wie eine Unterhaltung stattfindet, werden die an der Unterhaltung beteiligten *Sockets* erklärt. Deren Verhalten wird durch das *Client-Server Modell* bestimmt. Dieser Aspekt wird im zweiten Abschnitt genauer, sowohl für Java als auch für Oz, dargestellt.

Bisher wurde vorausgesetzt, daß ein Java-Prozeß im Hintergrund arbeitet. Leider ist noch nicht geklärt, wie dieser Prozeß gestartet wurde. Dieser Problematik wird sich der dritte Abschnitt widmen.

Im vierten Abschnitt steht die eigentliche Transparenz im Vordergrund. Die Transparenz zeigt sich dem Anwender dadurch, daß er normal in Oz programmiert, d.h. ihm stehen sämtliche grafischen Klassen zur Verfügung. In Wirklichkeit werden die Instanzen jedoch in Java generiert. Daraus folgt, daß die Ausführung der Methoden ebenfalls in Java stattfindet. Interessant ist hier, wie dem Anwender die Transparenz zur Verfügung gestellt wird. Dieser Aspekt ist einer der Schwerpunkte in dieser Schnittstelle.

Der zweite Schwerpunkt behandelt die auf der Java-Seite von Oz eingehenden Wünsche, d.h das Protokoll wird entgegengenommen und interpretiert. Abschnitt fünf zeigt die Arbeitsweise des Java-Parsers, der die gesamte Objektverwaltung leitet und auf die unterschiedlichen Anfragen entsprechend reagiert und antwortet.

Wenn die obigen Punkte richtig funktionieren, dann kann man zumindest schon etwas auf dem Bilschirm anzeigen. Dies wird auf die Dauer unbefriedigend sein, da der Anwender etwas mehr Interaktion wünscht. Dies ist auch etwas Essentielles und kein

Programmierer möchte auf diesen Luxus verzichten. Daraus ergibt sich die Notwendigkeit, Ereignisse bearbeiten zu können. Da Java schon ein eigenes *Event-Handling* besitzt, ist es sinnvoll, dieses für sich arbeiten zu lassen. Wie das gemacht wird, damit beschäftigt sich der Abschnitt sechs.

Eine wichtige Besonderheit dieser Schnittstelle ist die verzögerte Klassenanforderung. Damit werden Ressourcen wie Zeit und Speicher geschont. Das einfache System wird in Abschnitt sieben dargestellt.

Damit sind alle wichtigen Dinge geklärt und an dieser Stelle sollte man einen guten Überblick über die Schnittstelle haben. In dem letzten Abschnitt wird daher resümiert. Dabei will ich insbesondere auf Verbesserungsvorschläge, auf zukünftige Besonderheiten und auf die eigenen Erfahrungen mit dem Projekt eingehen.

3.1 Protokoll

Zu Beginn dieses Abschnitts möchte ich zunächst den Begriff *Protokoll* erläutern und ihn anhand der Beschreibung in meine Arbeit einordnen. Ein Protokoll ist wie im normalen Sprachgebrauch[1] eine Vereinbarung über einen geordneten Ablauf einer Kommunikation, wobei die Vereinbarung in der Informatik einen diktatorischen Charakter besitzt: Wer sich nicht an sie hält, wird von der Kommunikation ausgeschlossen. Protokolle sind von Anwendung zu Anwendung verschieden. Dadurch bleibt die Flexibilität erhalten. Protokolle können auch geschichtet werden. Entsprechend der aufeinander aufbauenden Schichten bei der Übertragung von Informationen wird für jede Ebene ein eigenes Protokoll vereinbart, dessen Realisierung auf das Protokoll auf der nächst tieferen Ebene baut. Ein wichtiges Beispiel hierfür ist das **Open Systems Interconnection Reference Model** (Day und Zimmermann, 1983), abgekürzt **ISO OSI** bzw. manchmal auch **OSI model**.

Beim Entwurfen von Protokollen benutzt man in der Regel *endliche Automaten* oder *Petri-Netze*. Hierbei nimmt man an, daß sich jeder Kommunikationspartner und das übertragene Medium in Zuständen befindet, die man zu einem Gesamtzustand zusammenfaßt. In der Praxis verwendete Protokolle sind meist sehr aufwendig, so daß Skizzen für endliche Automaten umfangreich werden.

In meiner Implementierung ist die Kommunikation und dementsprechend auch das Protokoll sehr einfach. Sämtliche Kommunikationsformen bestehen aus zwei Teilen; dem *Initiator* und dem *Responder*. Der Initiator soll immer eine Anfrage, die an den Responder gerichtet ist, starten. Der Responder nimmt diese Anfrage entgegen, wertet sie aus, und schickt eine entsprechende Antwort an den Initiator zurück. Konkret wird diese Kommunikation durch ein *Client-Server Modell* implementiert. Das Protokoll ist im Prinzip die Sprache, in der sich der Client mit dem Server unterhält. Dabei ist genau festgelegt, wie die Unterhaltung verständlich abläuft. Das Protokoll ist zur Zeit noch so einfach, daß es keine Schichten enthält. Dies kann sich in Zukunft ändern, wenn das Projekt größere Ausmaße annimmt, eventuell das Design ändert und das

[1] Beim Besuch der Queen läuft alles nach Protokoll. Wer sich nicht daran hält, wird gerügt.

Protokoll verkompliziert. Man darf aber nicht vergessen, daß dieses selbst auf anderen Protokollen aufbaut, zum Beispiel auf dem *Socket-Protokoll*, welches wiederum auf weiteren Protokollen aufbaut. Der Ausschluß aus der Kommunikation bei Nichteinhaltung der Regeln trifft hier nicht vollständig zu, da der Programmierer beim Auftreten eines Fehler durch Auffangen einer Exception darauf reagieren kann. Sicherlich gilt hier dennoch, daß falsche Angaben beispielsweise in der Parameterübergabe nicht den erhofften Effekt erzielen.

Im folgenden Abschnitt will ich mein Protokoll mittels einer Syntax formalisieren. Darauf aufbauend werden die unterschiedliche Fälle, die dieses Protokoll vorsieht, anhand der Syntax, einer Beschreibung und einem Beispiel vorgestellt. In späteren Kapiteln wird immer wieder auf diese Fälle zurückgegriffen, so daß sie später nicht mehr im Detail erläutert werden.

3.1.1 Syntax

Eine konstruierte Sprache hat Regeln, die bestimmen, wie zur Verfügung stehende Zeichen und Wörter aneinandergereiht werden dürfen. Regeln kann man formal aufschreiben. Diese formale Beschreibung des Aufbaus von Sätzen oder Wörtern, die zu der Sprache gehören, bezeichnet man als Syntax. Mit Hilfe der Syntax kann man demnach bestimmen, ob bestimmte Zeichenreihen oder Sätze korrekt sind. Dies gilt auch für Protokolle, die auf solchen Vereinbarungen basieren. Die Korrektheit von Protokollen kann man auf mehrere äquivalente Weisen bestimmen:

1. Man kann einen Automaten angeben, der genau die syntaktischen korrekten Zeichenreihen akzeptiert.
2. Durch Syntaxdiagramme kann man die Syntax einer Sprache graphisch beschreiben. Diese Methode wendet man häufig bei der Definition von Programmiersprachen an.
3. Man kann eine Grammatik definieren, deren erzeugte Sprache genau der Menge aller syntaktisch korrekten Zeichenfolgen entspricht.

Es ist von Vorteil, den dritten Punkt für die Syntaxbeschreibung des Java-Oz Protokolls zu wählen. Speziell wende ich die EBNF (Extended Backus Naur Form) Notation an. Variablen erkennt man durch die Klammerung mit $\langle$ und $\rangle$. Nicht geklammerte Ausdrücke nennt man Tokens, d.h sie sind feststehend und nicht ersetzbar. Weitere Terminale werden durch die Wertebereiche von $\langle$*Integer*$\rangle$, $\langle$*String*$\rangle$ etc. bestimmt. Manche Variablen, zum Beispiel die Portnummer des Servers oder die Referenz eines Objektes stammen genau aus diesen Bereichen. Tabelle 3.1 listet alle Variablen auf, die auf solche Bereiche verweisen. Man hätte die Bereiche auch gleich in die Regeln einbauen können, dadurch würde aber die Lesbarkeit der Produktionen gemindert werden.

In Tabelle 3.2 sind alle bekannten Wertebereiche, die das Protokoll kennt aufgezählt. Eine Produktion ersetzt eine Variable durch eine Regel. Hierbei heißt ’::=’ “kann sein”

⟨*Port*⟩	::=	⟨*Integer*⟩
⟨*Class*⟩	::=	⟨*String*⟩
⟨*Referenz*⟩	::=	⟨*Integer*⟩
⟨*Number*⟩	::=	⟨*Integer*⟩
⟨*MethName*⟩	::=	⟨*String*⟩
⟨*Detail*⟩	::=	⟨*String*⟩
⟨*Stack*⟩	::=	⟨*String*⟩

Tabelle 3.1: Java-Oz Protokoll: Abbildung von Variablen auf Standardtypen

und das Symbol '|' wird als "oder" gelesen. Zusätzlich steht die Klammerung { und } für keine oder mehrere Wiederholungen. Das Terminal ':' soll als ein Beispielsaperator eingesetzt werden, der die Wörter trennt. Daran erkennt das Interface, bis zu welchem Punkt eine Aussage geht. Zur Zeit wird in der Implementierung das Zeichen '<<' benutzt.

⟨*Value*⟩	::=	null
	|	⟨*Boolean*⟩
	|	⟨*Integer*⟩
	|	⟨*Byte*⟩
	|	⟨*Short*⟩
	|	⟨*Long*⟩
	|	⟨*Float*⟩
	|	⟨*Double*⟩
	|	⟨*Char*⟩
	|	⟨*String*⟩
	|	⟨*Referenz*⟩

Tabelle 3.2: Java-Oz Protokoll: Value

Zu den oben erwähnten Wertebereichen existieren entsprechende Terminale, die innerhalb einer Nachricht signalisieren, wie die Parameter getypt sind. Dies ist notwendig, da alle Nachrichten als Zeichenketten und somit als ein Typ verschickt werden.[2] Eine komplette Auflistung der dem Interface bekannten Typterminale befindet sich in Tabelle 3.3.

[2]Später sollen ByteStrings verschickt werden, da diese die Nachrichtenmenge um einen Faktor 8 reduzieren.

⟨*Type*⟩	::=	void
	|	boolean
	|	int
	|	byte
	|	short
	|	long
	|	float
	|	double
	|	char
	|	string
	|	object:⟨*Class*⟩

Tabelle 3.3: Java-Oz Protokoll: Typen

Eine Produktion eines Satzes beginnt immer mit der Startvariablen. Das Interface kennt folgende Zwei Startvariablen: *Action* und *JavaResult*.Aus der Startvariablen *Action* lassen sich alle möglich auftretenden Anforderungsfälle konstruieren (siehe Tabelle 3.4).

⟨*Action*⟩	::=	⟨*ServerShutDown*⟩
	|	⟨*SendServerPort*⟩
	|	⟨*GetJClass*⟩
	|	⟨*NewJavaObject*⟩
	|	⟨*AccessJavaMethod*⟩
	|	⟨*AccessJavaField*⟩
	|	⟨*NewAdapterObject*⟩
	|	⟨*JavaResult*⟩
	|	⟨*AccessOzMethod*⟩

Tabelle 3.4: Java-Oz Protokoll: Aktionen

Da es zu Anfragen auch Antworten gibt, müssen diese ebenfalls gekennzeichnet werden. In unserer Grammatik ist die Startschlüsselvariable *JavaResult*, aus der sich alle vorkommenden Antworten ableiten lassen. Diese sind in Tabelle 3.5 aufgezählt. Alle diese Aktionen und deren Resultate werden im nächsten Kapitel bei der Betrachtung verschiedener Nachrichten genauer untersucht.

In der Erläuterung zum Protokoll wurde gesagt, daß die Zustände durch einen endlichen Automaten genau bestimmt werden können. Dies gilt allerdings nur für den

⟨*JavaResult*⟩	::=	⟨*ServerShutDownRes*⟩
	|	⟨*SendServerPortRes*⟩
	|	⟨*GetJClassRes*⟩
	|	⟨*NewJavaObjectRes*⟩
	|	⟨*AccJavaMethRes*⟩
	|	⟨*AccJavaFieldRes*⟩
	|	⟨*NewAdObjRes*⟩
	|	⟨*JavaException*⟩

Tabelle 3.5: Java-Oz Protokoll: JavaResult

Handlungsablauf, jedoch nicht für den Inhalt des Protokolls. Wie der Inhalt des Java-Oz Protokolls aussieht, wird in der hier angegebenen Grammatik dargestellt. Hierzu betrachtet man zunächst die Kodierung der Parameterübergabe, zum Beispiel bei der Objektinstanziierung oder beim Methodenaufruf (siehe Tabelle 3.6). Die Tatsache, daß zuerst nacheinander alle Parameter-Typen und erst dann nacheinander genau dieselbe Anzahl an Parameter-Werten angegeben ist, bringt mit sich, daß die Grammatik nicht mehr regulär und damit nicht äquivalent zum endlichen Automaten ist. Stattdessen entspricht sie der Sprache $L = \{a^n b^n | n \geq 1\}$, die bekanntlich kontextfrei ist. Aber auch die Angabe in ⟨*Parameter*⟩ ist relativ unpräzise. Die Anzahl der Typen und Werte ist hiernach nicht zwingendermaßen gleich. Tatsächlich müssen genausoviele Typtoken und genausoviele Werte wie in Number angegeben gesendet werden. Dieses Problem taucht an mehreren Stellen der Grammatikbestimmung auf, so daß die genaue Interpretation intuitiv verstanden werden sollte.

⟨*Parameter*⟩ ::= ⟨*Number*⟩:{⟨*Type*⟩:}{⟨*Value*⟩:}

Tabelle 3.6: Java-Oz Protokoll: Parameter

3.1.2 Betrachtung verschiedener Protokollfälle

In diesem Abschnitt werden verschiedene Fälle etwas genauer betrachtet. Hierzu wird für jede Aktion und jedes Resultat die Syntax, ein Beispiel und die Beschreibung angegeben. Die Syntax ist die Grundlage der Kommunikation, auf der verschiedene Aktionen sowohl in Oz als auch in Java aufbauen. sie werden ab Abschnitt 3.2 vorgestellt.

3.1.2.1 Anforderung: Senden des Oz-Server-Ports

Syntax:

⟨*SendServerPort*⟩ ::= 11:⟨*Port*⟩

Beispiel:

„11:1026”

Erläuterung: Wurde ein Oz-Server erfolgreich installiert, teilt er Java seinen Port mit.

3.1.2.2 Antwort: Senden des Oz-Server-Ports

Syntax:

⟨*SendServerPortRes*⟩ ::= 4:ack

Beispiel:

„4:ack”

Erläuterung: Java antwortet mit dem Wort „ack” (Acknowledge), daß es die Nachricht korrekt erhalten hat.

3.1.2.3 Anforderung: Kommunikation beenden

Syntax:

⟨*ServerShutDown*⟩ ::= 0

Beispiel:

„0”

Erläuterung: Dies ist im Grunde nur eine Stopanweisung. Sie enthält keine weiteren Informationen.

3.1.2.4 Antwort: Kommunikation beenden

Syntax:

⟨*ServerShutDown*⟩ ::= 4:RES:STOP

Beispiel:

„4:RES:STOP"

Erläuterung: Java gibt Rückmeldung, daß es die *Halt* Anweisung verstanden hat und daß der Server im Begriff ist runterzufahren.

3.1.2.5 Anforderung: Klasse

Syntax:

⟨*GetJClass*⟩ ::= 5:⟨*Class*⟩

Beispiel:

„5:java.lang.Object" [3]

Erläuterung: Wenn Oz eine Instanz von einer in Java vorhandenene Klasse kreieren möchte, muß es wissen, wie die Klasse strukturiert ist, d.h. welche Methoden enthalten sind und ob sie einen Konstruktor enthält. Dazu sendet Oz diese Nachricht an Java, mit der Oz signalisiert, daß es nun auf eine Antwort wartet.

3.1.2.6 Antwort: Klasse

Syntax:

⟨*GetJClassRes*⟩	::=	4:RES:CLASS:⟨*Interface*⟩:⟨*Abstrakt*⟩: ⟨*Constructor*⟩:⟨*SuperClass*⟩: ⟨*Meths*⟩:⟨*Fields*⟩
⟨*Interface*⟩	::=	INTERFACE:true
	|	INTERFACE:false
⟨*Abstrakt*⟩	::=	ABSTRACT:true
	|	ABSTRAKT:false
⟨*Constructor*⟩	::=	CON:⟨*Class*⟩
	|	CON:none
⟨*SuperClass*⟩	::=	SUPER:⟨*Class*⟩
	|	INTERFACE:null
⟨*Meths*⟩	::=	METHS:⟨*Number*⟩:{⟨*MethName*⟩:}{⟨*MethIsStatic*⟩:}
⟨*MethIsStatic*⟩	::=	MSTATIC:{⟨*IsStatic*⟩:}
⟨*Fields*⟩	::=	FIELDS:⟨*Number*⟩:{⟨*FieldName*⟩:}{⟨*FieldIsStatic*⟩:}
⟨*FieldIsStatic*⟩	::=	FSTATIC:{⟨*IsStatic*⟩:}
⟨*IsStatic*⟩	::=	is
	|	ns

Beispiel:

„4:RES:CLASS:INTERFACE:false:ABSTRACT:false:CON:java.lang.Object:
SUPER:null:METHS:9:clone:equals:finalize:getClass:hashCode:notify:
notifyAll:toString:wait:MSTATIC:ns:ns:ns:ns:ns:ns:ns:ns:ns:
FIELDS:0:FSTATIC"

Erläuterung: Hat Oz einen Wunsch nach einer Java-Klasse geäußert, stellt Java den Inhalt der Klasse für Oz verständlich dar. Zur Zeit sind nur der Konstruktor und die

[3] Dies ist nur ein Beispiel. Das System stellt dem Anwender keinen Zugriff auf diese Klasse.

Methoden interessant. Der Konstruktor ist je nach Klasse entweder ein voll qualifizierter Klassenname oder „none". Letzteres wird angegeben, wenn die Klasse keinen Konstruktor besitzt, also zum Beispiel nur aus statischen Methoden besteht, wie zum Beispiel die Klasse *com.sun.java.swing.BorderFactory*. Weiterhin werden Modifizierer zur Charakterisierung der klasse mitgeschickt. So erhält man Informationen darüber, ob eine Klasse abstrakt oder sogar ein Interface ist. Zusätzlich wird die SuperClass mitgeschickt, um eine hierarchische Abbildung der Klassen in Oz herzustellen. Hierzu werden immer nur die Methoden mitgeschickt, die auch innerhalb dieser Klasse deklariert wurden. Die geerbten Methoden stehen in der SuperClass. Ist diese in Oz noch nicht vorhanden, wird sie sofort angefordert. Die TopKlasse ist java.lang.Object. Sie wird von keiner weiteren Klasse mehr abgeleitet, so daß in ⟨*SuperClass*⟩ *null* steht.Die angegebene Anzahl der übertragenen Methoden ist zur bequemeren Programmierung gedacht. Es fällt auf, daß jede Methode genau einmal übertragen wird. Es stellt sich die Frage, wie überladene Methoden erkannt werden. Es ist so, daß Oz sich nicht darum kümmert, welche der überladenen Methoden die richtige ist; das übernimmt Java. Den Methoden folgen deren Modifier. Diese sind für den späteren Methodenaufruf notwendig. Hierbei steht *is* für eine statisch und *ns* für eine dynamisch deklarierte Methode. Zuletzt werden die Namen der Felder und deren statische Deklarierung wie bei den Methoden übermittelt.

3.1.2.7 Anforderung: Instanziierung eines Objektes

Syntax:

⟨*NewJavaObject*⟩ ::= 1:⟨*Class*⟩:⟨*Parameter*⟩

Beispiel:

a) mit Parametern, b) ohne Parameter

a) „1:com.sun.java.swing.JFrame:1:string:Hello World!"

b) „1:com.sun.java.swing.JPanel:0"

Erläuterung: Wenn eine von Java nach Oz abgebildete Klasse vorliegt, dann kann instanziiert werden. Hierzu teilt Oz Java den voll qualifizierten Namen der Klasse, die Anzahl der Argumente, die Typen der Argumente und die Werte der Argumente mit. Anschließend wartet Oz auf eine Antwort.

3.1.2.8 Antwort:Instanziierung eines Objektes

Syntax:

⟨*NewJavaObjectRes*⟩ ::= 4:RES:⟨*Referenz*⟩

Beispiel:

„4:RES:3”

Erläuterung: Ist eine Instanziierung erfolgreich, dann wird eine Referenz zurückgeschickt, unter der man das Objekt zukünftig in Java ansprechen kann.

3.1.2.9 Anforderung: Methodenaufruf

Syntax:

⟨*AccessJavaMethod*⟩ ::= 2:⟨*StaticMethod*⟩:
| 2:⟨*DynamicMethod*⟩:
⟨*StaticMethod*⟩ ::= ⟨*Class*⟩:⟨*MethName*⟩:⟨*Paramater*⟩
⟨*DynamicMethod*⟩ ::= ⟨*Referenz*⟩:⟨*MethName*⟩:⟨*Paramater*⟩

Beispiel:

„2:4:addActionListener:1:object:JOzActionListener:5”

Beispiel:

„2:com.sun.java.swing.BorderFactory:createEmptyBorder:4:
int:int:int:int:30:30:10:170”

Erläuterung: Um eine Methode aufzurufen, werden intern zwei verschiedene Nachrichten geschaffen. Zum einen ein Protokoll für einen Aufruf einer dynamischen Methode und zum anderen ein Protokoll zum Aufruf einer statischen Methode. Mit Ersteren beschäftigt sich dieser Abschnitt. Hierzu werden die Referenz, der Methodenname und die Argumente benötigt. Die Referenz ist die Art von Referenz, die bei einer Instanziierung als Resultat zurückgegeben wird und im neu enstandenen Oz-Objekt gespeichert ist. Mit Hilfe der Referenz erkennt Java, um welches Objekt es sich handelt und sucht durch Angabe des Methodennamens und der Argumente nach der verlangten Methode. Die Syntax der Argumente ist wie bei der Instanziierung von Objekten.

Der Aufruf einer statischen Methode unterscheidet sich von der dynamischen nur in einem Punkt. Statt einer Referenz wird der voll qualifizierte Klassenname verschickt. Java erkennt an dieser Position den Unterschied selbst und handelt. Unter statischer Methode versteht man an dieser Stelle eine Methode, die zu einer Klasse gehört, deren sämtliche Methoden statisch sind, und die keinen Konstruktor besitzt.

3.1.2.10 Antwort: statischer/dynamischer Java Methodenaufruf

Syntax:

⟨*AccessJavaMethRes*⟩ ::= 4:RES:⟨*Type*⟩:⟨*Value*⟩

Beispiel:

a) „4:RES:void"
b) „4:RES:boolean:true
c) „4:RES:object:6:com.sun.java.swing.border.Border"

Erläuterung: Nach einem gelungenen Methodenaufruf teilt Java Oz das Ergebnis mit. Das Ergebnis kann einen primitiven Typ oder einen Objekttyp beinhalten. Ein Objekttyp enthält Informationen über die Referenz und den voll qualifizierten Klassennamen. Hierbei sollte man beachten, daß der Typ „String" entgegengesetzt zu Java als ein primitiver Typ benutzt wird. Ist der Objekttyp ein neu generiertes Objekt, wird dieses in Oz registriert.

3.1.2.11 Anforderung: Feldzugriff

Syntax:

⟨*AccessJavaField*⟩ ::= 12:⟨*StaticField*⟩:
| 12:⟨*DynamicField*⟩:
⟨*StaticField*⟩ ::= unit:⟨*Class*⟩:⟨*FieldName*⟩
⟨*DynamicField*⟩ ::= ⟨*Referenz*⟩:⟨*FieldName*⟩

Beispiel:

„12:unit:java.awt.Color:orange"

Erläuterung: Auch beim Zugriff auf Felder muß man aufpassen, ob auf eine Klasse ohne Konstruktor zugegriffen wird. Ist dies der Fall, so wird *unit* als Kennzeichnung mitgeschickt. Außerdem muß anstelle der Referenz der voll qualifizierte Klassenname übermittelt werden.

3.1.2.12 Antwort: Feldzugriff

Syntax:

⟨*AccessJavaFieldRes*⟩ ::= 4:RES:⟨*Type*⟩:⟨*Value*⟩

Beispiel:

„4:RES:object:20:java.awt.Color”

Erläuterung: Das Ergebnis sieht genauso aus wie bei einem Methodenaufruf.

3.1.2.13 Anforderung: Instanziierung eines Adapterobjektes

Syntax:

Siehe Tabelle 3.7

Beispiel:

„3:JOzActionListener:EVENTS:actionPerformed”

Erläuterung: Sollen Ereignisse abgehört werden, dann wird ein Listener angefordert. Java stellt schon angepaßte Adapterklassen zur Verfügung, denen mitgeteilt werden muß, welche Ereignisse an Oz weitergeleitet werden sollen. Diese sind nacheinander in der Nachricht aufgelistet. Oz erkennt an den Methodennamen selbst, welche Ereignisse den Anwender interessieren, d.h. der Anwender muß nichts explizit angeben.

3.1.2.14 Antwort: Instanziierung eines Adapterobjektes

Syntax:

⟨*NewAdapterObject*⟩	::=	3:⟨*AdapterClass*⟩
⟨*AdapterClass*⟩	::=	JOzActionListener:EVENTS:actionPerformed
	|	JOzAdjustmentListener: EVENTS:adjustmentValueChanged
	|	JOzComponentListener: EVENTS:{⟨*ComponentEvents*⟩:}
	|	JOzContainerListener: EVENTS:{⟨*ContainerEvents*⟩:}
	|	JOzFocusListener:EVENTS:{⟨*FocusEvents*⟩:}
	|	JOzItemListener:EVENTS:itemStateChanged
	|	JOzKeyListener:EVENTS:{⟨*KeyEvents*⟩:}
	|	JOzMouseListener:EVENTS:{⟨*MouseEvents*⟩:}
	|	JOzMouseMotionListener: EVENTS:{⟨*MouseMotionEvents*⟩:}
	|	JOzTextListener:EVENTS:textValueChanged
	|	JOzWindowListener: EVENTS:{⟨*WindowEvents*⟩:}
⟨*ComponentEvents*⟩	::=	componentHidden
	|	componentMoved
	|	componentResized
	|	componentShown
⟨*ContainerEvents*⟩	::=	componentAdded
	|	componentRemoved
⟨*FocusEvents*⟩	::=	focusGained
	|	focusLost
⟨*KeyEvents*⟩	::=	keyPressed
	|	keyReleased
	|	keyTyped
⟨*MouseEvents*⟩	::=	mouseClicked
	|	mouseEntered
	|	mouseExited
	|	mouseReleased
	|	mousePressed
⟨*MouseMotionEvents*⟩	::=	mouseDragged
	|	mouseMoved
⟨*WindowEvents*⟩	::=	windowActivated
	|	windowClosed
	|	windowClosing
	|	windowDeactivated
	|	windowDeiconified
	|	windowIconified
	|	windowOpened

Tabelle 3.7: Java-Oz Protokoll: NewAdapterObject

⟨*NewAdObjRes*⟩ ::= 4:RES:⟨*Referenz*⟩

Beispiel:

„4:RES:5”

Erläuterung: Wie bei der Objektinstanziierung wird eine Referenz zurückgegeben, unter der man das neue Objekt aufrufen kann.

3.1.2.15 Anforderung: Oz Methodenaufruf

Syntax:

⟨*AccessOzMethod*⟩ ::= 7:⟨*Referenz*⟩:⟨*MethName*⟩:⟨*Parameter*⟩

Beispiel:

7:5:actionPerformed:1:object:java.awt.event.ActionEvent:13”

Erläuterung: Tritt ein Ereignis ein, das den Anwender interessiert, verschickt Java eine Nachricht an Oz, die den Listener, der das Ereignis aufgefangen hat, den voll qualifizierten Klassennamen des aufgetretenen Events und seine Referenz, enthält. Dieses Beispiel ist ein spezieller Methodenaufruf, der das Eventhandling anspricht. Eine generelle Schnittstelle von Java nach Oz gibt es noch nicht, obwohl oben genannte Syntax schon für diese gilt. Eine Anmerkung gilt der übertragenen Referenz des eigentlichen Events. Damit der Anwender mit dem Event arbeiten kann, muß Java das Objekt in seine Hastabelle aufnehmen. Hat der Anwender auf das Ereignis vollständig reagiert, könnte Java das Objekt wieder freigeben. Dies passiert jedoch nicht, da nicht bekannt ist, ob der Anwender das Objekt auf der Oz-Seite gespeichert hat, um es später noch ein weiteres Mal zu benutzen. Jetzt kann es passieren, daß unnötig viele Eventobjekte in Java mitgeschleppt werden. Dies kann bei einer Applikation, die viele Events abfängt, zu Speicherproblemen führen. Hier muß nach einer besseren Lösung oder einer Konvention gesucht werden.

3.1.2.16 Antwort: Oz Methodenaufruf

Erläuterung: Aufgrund einer fehlenden Schnittstelle wird zur Zeit kein Rückgabewert erwartet. [4]

[4] Dies kann unter anderem ungünstig für die Eventbehandlung sein, da Java nicht erfährt, ob Oz die Information erhalten hat.

3.1.2.17 Antwort auf eine Anfordung mit einer Exception

Syntax:

⟨*JavaException*⟩ ::= 4:EXC:⟨*Class*⟩:⟨*Detail*⟩:⟨*Stack*⟩

Beispiel:

4:EXC:java.lang.NoSuchMethodException:null:

java.lang.NoSuchMethodException
at Parse.invokeMethod(Parse.java:669)
at Parse.doAccessJavaMethod(Parse.java:468)
at Parse.analyzeRequest(Parse.java:161)
at SubServer.parseMsg(JOzServer2.java:156)
at SubServer.run(JOzServer2.java:115)

Erläuterung: Eine Anforderung kann auch einen Fehler hervorrufen, der sich in einer Exception niederschlägt. Diese wird nach Oz geschickt, um dem Anwender größte Flexibilität zu ermöglichen. Die Exception beinhaltet einen voll qualifizierten Klassennamen, so daß der Anwender erkennt, welche Ausnahme geworfen wird. Zuätzlich wird eine genauere Information, d.h. bei welcher Gelegenheit diese Ausnahme geworfen wurde, hinzugefügt. Zuletzt steht die Aufrufleiter der einzelnen Methoden bis zur Exceptiongenerierung.

3.2 Client Server Modell

Wie schon im Kapitel über das Protokoll erwähnt, ist das Protokoll die Sprache, in der der Client der einen Seite sich mit dem Server der anderen Seite mittels Nachrichten laut vereinbartem Protokoll unterhält. Die einzelnen Wörter dieser Nachrichten werden durch einen Separator (z.Z. „<<”) getrennt. Anhand dieser Separatoren kann Java die Nachrichten in einen *StringTokenizer* und Oz in eine Liste zur bequemen Weiterbearbeitung umwandeln.

Um einen Server anzusprechen, muß man einen Client bereitstellen. Es werden zu Beginn zwei Clients, gestartet einer in Java und einer in Oz. Beide bleiben im Speicher, bis eine Anforderung, das System zu beenden, erscheint.

Der Client in Java reduziert seine Aufgaben zur Zeit auf die Übertragung von Ereignissen. Er ist auch viel einfacher gebaut, als der in Oz. Der Client in Java beinhaltet lediglich eine Sendemethode und eine Schließmethode. Er wird im Zuge der Verbesserung der Schnittstelle von Java nach Oz verbessert.

Der Client in Oz ist viel komplexer. Er besitzt einen Mechanismus, mit dem er beliebig lange Nachrichten empfangen kann. Um etwas zu empfangen, muß er auch versenden können. Dies tut er mit einer einfachen Methode. Auf dieser basierend, sind Methoden zum Anfordern einer neuen Klasse, zum Anfordern eines neuen Objektes, zum Anfordern eines neuen Adapterobjektes, zum Ausführen einer Methode, zum Senden der Serverportnummer und zum Senden eines Signals der Beendigung der Kommunikation etc., gegeben.

Der Server hat in beiden Sprachen die Aufgabe, die Anfragen der Gegenseite anzunehmen, zur Bearbeitung an den jeweiligen Parser weiterzugeben, und eine zurückerhaltene Antwort zurückzuschicken.

Das ursprüngliche Client Server Modell war sehr ineffizient im Hinblick auf Speicher und Zeitressourcen. Für jede Anfrage wurde ein eigener Client Socket geschaffen, dem auf der Server Seite ein weiterer neuer Socket zur Kommunikation zugewiesen wurde. Sockets sind teuer in der Herstellung und belegen viel Speicherplatz.

Aus diesem Grunde belegt das neue Client-Server Modell eine konstante Zahl an Sockets. Zunächst bestand das Ziel darin, insgesamt vier Sockets zu betreiben, doch aufgrund der Konstruktion wurden es sechs (siehe auch Grafik 3.1). Diese teilen sich wie folgt auf:

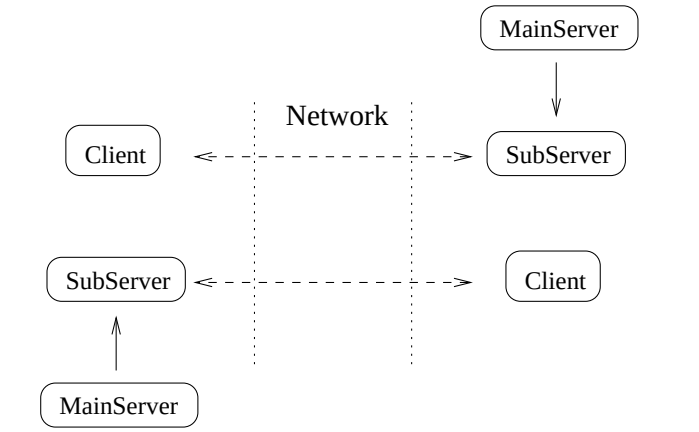

Abbildung 3.1: Kommunikationsteilnehmer

Auf jeder Seite ist ein Client Socket, über den eine Anforderung gestartet wird. Zusätzlich sind auf beiden Seiten ein Server und ein Subserver vorhanden. Der Subserver auf der Java-Seite ist das Ergebnis der Tatsache, daß die *ServerSocket.class* lediglich an einem bestimmten Port lauscht, bis eine Verbindung gewünscht wird. Ist dies der Fall, so wird sofort ein Socket instanziiert, der über den vom Server bestimmten Port mit Oz kommuniziert. Es besteht keine Möglichkeit, daß der ServerSocket die Kommunikation selbst abwickelt. Er überläßt es dem SubServer, der nach der Kommunikation die Aufgaben eines Servers übernimmt, d.h. er schließt die Leitung nicht, sondern er

horcht an dieser Leitung und wartet auf Anfragen. Der ursprüngliche Server hat mit der Annahme der ersten Anfrage seine Aufgabe erledigt. Es besteht nun die Frage, warum ein Socket nicht gleich wie ein Server implementiert wird. Dies läßt sich damit erklären, daß der Socket primär als ein Client angesehen wird, der an einen Host und Port anbindet. Dies widerspricht der Aufgabe eines Servers, der selbst einen Port belegt und auf Anweisungen wartet.

Außerdem verstößt es gegen die *dynamische Portzuweisung*, d.h. sowohl der Java-Server als auch Oz-Server haben keine feste Portnummer. Sie suchen nach noch nicht belegten Ports, und nehmen sich einen. Es wäre auch fatal, das System statisch zu machen; man sollte bedenken, daß eventuell andere Anwendungen den gewünschten Port belegen könnten.

In Java arbeitet der SubServer als ein Thread in einer Endlosschleife. In Oz ist der SubServer ebenfalls ein Thread, der die Klasse Open.socket expandiert.

Was nun übrig bleibt, ist die genaue Beschreibung des Startvorgangs des Systems. Dabei sieht man, wie die dynamische Portzuweisung funktioniert. Die unten angegebene Reihenfolge der Vorgänge kann man ebenfalls aus der Grafik 3.2 entnehmen.

1. Zunächst wird in Oz der Oz-Server initiiert.
2. Da der Oz-Server eine dynamische Portzuweisung genießt, gibt er nach dem Aufruf seine Portnummer zurück.
3. Als nächstes startet Oz den Java-Prozeß, vielmehr ein Bash-Skript, welches die Java-Server Applikation mit dem OzPort als Argument startet.
4. Java startet einen eigenen Server ...
5. ... und gibt die eigene Portnummer zurück (dynamisch).
6. Java instanziiert einen Client, ...
7. ... der sich beim Oz-Server anmeldet.
8. Aufgrund der neuen Anfrage, instanziiert der Server einen SubServer, der die Kommunikation übernimmt. Der Server hat dagegen seine Aufgabe erledigt.
9. Der SubServer in Oz meldet dem Client in Java seine Betriebsbereitschaft.
10. Der Java Client nimmt die Bereitschaft an und übermittelt Oz den ServerPort von Java.
11. Oz startet den eigenen Client mittels der angegebenen Portnummer.
12. Der Client meldet sich beim JavaServer an ...
13. ... der den SubServer startet und sich selbst beendet.
14. Zuletzt meldet der JavaSubServer seine Betriebsbereitschaft dem OzClient.

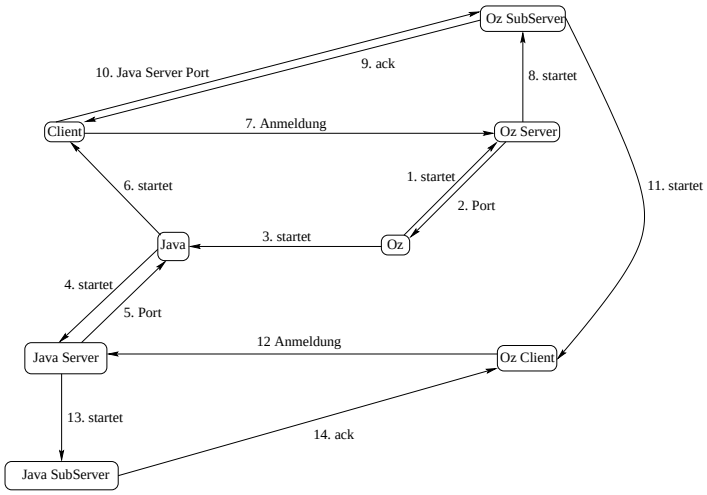

Abbildung 3.2: Starten der Java-Oz Umgebung

Nun ist das System bereit.

Desweiteren gibt es einen neuen Stopmechanismus. Wünscht Oz die Zusammenarbeit mit Java nicht mehr, schickt es ein *ServerShutDown* Signal. Java antwortet zunächst über den eigenen Client, so daß der Oz-Server eine letzte Nachricht erhält und sich selbst beendet. Erst dann gibt Java dem Oz Client bekannt, daß es die Nachricht verstanden hat, und beendet seinen Prozeß.

Intern werden die Nachrichten sowohl in Java als auch in Oz in der Klasse *JOzMessage* generiert und verwaltet. Sie bestimmt zusätzlich, welches Zeichen die einzelnen Wörter voneinander trennt.

3.3 Java starten und beenden

Damit man überhaupt Java Klassen nutzen kann, muß man Java, bzw. vielmehr den Server, der die Anfragen entgegennimmt und auswertet, starten. Insgesamt gibt es drei Möglichkeiten, wovon nur zwei praktikabel sind.

1. Java im Terminal starten
2. Java innerhalb von Oz auf Anfrage starten
3. Java innerhalb von Oz automatisch beim Hochfahren starten

Alle drei rufen ein Shellscript auf, welches Java mit zusätzlichen Parametern und der Hauptklasse startet. Das Skript (*rJos.sh*) ist aus Not entstanden. Ursprünglich habe ich versucht Java durch eine *Open.pipe* Klasse aufzurufen. Dies hat aber nie mit der

Parameterübergabe geklappt. Entweder wurde *java* oder die Hauptklasse nicht gefunden. Außerdem mußten noch zusätzliche Klassenpfade als Parameter angegeben werden. Nach etlichen Versuchen habe ich das Skript geschrieben, das bis heute gut funktioniert. Da *Open.pipe* auch während der Laufzeit nicht stabil genug lief, habe ich während der Entwicklungszeit das Shellskript separat vor Oz im Terminal gestartet.

Der zweite Anlauf bestand darin, dem Anwender über ein Modul Prozeduren zum Starten (*Java.start*) und Beenden (*Java.stop*) von Java Prozessen zur Verfügung zu stellen, damit dieser selber bestimmen kann, wann er Java startet. Der Gedanke dabei war, die Zeitverzögerung und den erhöhten Speicherverbrauch, der bei diesem Aufruf erfolgt, zu verhindern.

3.4 Abbildung einer Java-Klasse auf eine Oz-Klasse

Ein wesentlicher Punkt dieses Projektes ist die *Transparenz*. Darunter versteht man, daß dem Anwender verborgen bleibt, welche Prozesse im Hintergrund ablaufen. Der Anwender glaubt, daß die grafischen Elemente, die ihm zu Verfügung gestellt werden, in Oz implementiert sind, und daß Oz die komplette grafische Ausgabe verwaltet. Dies ist nicht der Fall, da Oz zwar die AWT und Swing Klassen zur Verfügung stellt, aber alle Informationen nach Java weiterleitet, so daß die grafische Verwaltung in Java stattfindet. Somit ist der Hauptpunkt dieses Kapitels, die Implementierung einer zur Laufzeit generierten Oz-Klasse, die denselben Aufbau hat, wie die AWT bzw. Swing Klasse, die sie repräsentiert. Hiermit wird der *hierarchische Aufbau* der Klasse impliziert, d.h. die Klasse bekommt nur die Methoden, die sie selbst deklariert. Alle anderen sind in ihrer SuperClass zu finden. Ist diese zur Generierungszeit in Oz noch nicht vorhanden, so wird sie zunächst angefordert und generiert.

In den folgenden Abschnitten werden die hauptsächlichen Implementationspunkte genauer betrachtet. Es wird erklärt, warum jede generierte Klasse eine Basisklasse erweitert. Ganz wichtig ist der Aufbau eines Konstruktors, einer Methode und einer Feldzugriffsmethode, den Elementen der Klasse, in denen die Kommunikation stattfindet. Desweiteren wird darauf eingegangen, wie eine Klasse zur Laufzeit generiert, und ihre Instanzen verwaltet werden. Ein weiterer wichtiger Punkt ist die Herstellung der zu verschickenden Nachricht, insbesondere im Hinblick auf das Erkennen von Parametern beim Aufruf eines Konstruktors oder einer Methode. Beim Aufruf einer Methode kommt noch die Auswertung des Ergebnisses hinzu.

3.4.1 Basisklasse

Jede zur Laufzeit generierte Oz-Klasse, die eine AWT bzw. eine Swing Klasse repräsentiert, erweitert eine Basisklasse, die Felder *id, type, isStatic* und Zugriffsmethoden auf diese Felder bereitstellt (Algorithmus 4). In dem Feld *id* wird die Referenz, die bei der Instanziierung eines Objektes zurückgeliefert wird, gespeichert. Somit kennt jedes Objekt den Schlüssel, unter dem das Pendant in Java verwaltet wird. Das Feld *type* speichert den voll qualifizierten Java Klassennamen, z.B. ,,java.lang.Object”. In

isStatic wird festgehalten, ob die ganze Klasse nur aus statischen Methoden besteht. Dies ist wichtig, weil die Nachricht sich bei Anwendung einer Methode aus einer dynamischen Klasse von der aus einer Klasse mit nur statischen Methoden unterscheidet (3.1.2.9).

Algorithmus 4 Basis-Klasse in Oz

```
class BaseJOz from BaseObject
    attr id type
        isStatic: false
    meth getId(?Id)
        Id=@id
    end
    meth setId(Id)
        id←Id
    end
    meth getType(?Type)
        Type=@type
    end
    meth setType(Type)
        type←Type
    end
    meth setIsClassStatic(Static)
        isStatic←Static
    end
    meth getIsClassStatic(?Static)
        Static=@isStatic
    end
    meth getSelf(?Self)
        Self=self
    end
end
```

Ursprünglich hatte die Basisklasse noch zwingendere Gründe für ihre Anwendung, da die Spezifikation des Objekt- und Klassenmodells von Oz anders war. Damals wurden Klassen mittels eines Prozeduraufrufs generiert, der Methoden als Tupel von Paaren (Methodenname und unäre Prozedur) aufnahm. Diese Prozeduren konnten keine Objektoperatoren benutzen. So konnten sie nicht auf ihre eigene Referenzen (id) zugreifen. Insbesondere konnten die Prozeduren den **self**-Operator nicht ausdrücken. Lediglich *Static Method Calls* waren innerhalb von Prozeduren erlaubt.

Algorithmus 5 Generieren einer Methode

```
fun {GetOneMethClass MethName IsInterf SuperClass IsStatic}
        ⋮
    class $ from SuperClass
        meth !MethName(...) = M
            ⋮
        end
    end
end
```

Durch das Ändern der Spezifikation wurden wesentlich Prozeduren, die für den damaligen Implementierungsstand wichtig waren, aus dem *Object* Modul herausgenommen.

Daraufhin wurde die gesamte Konstruktion überdacht und neu bestimmt. Hierbei wurde intensiv die Eigenschaft der Bindung von Variablen genutzt. Es gibt drei Funktionen, jede gibt einen bestimmten Teil einer Klasse zurück. Eine den Konstruktor, eine die Methode und eine das Feld. Diese Funktionen nehmen Parameter auf, deren Bindung an die Grundkonstrukte weitergereicht werden. So kann man zum Beispiel den Namen einer Methode weitergeben. Siehe hierzu Algorithmus 5.

Bei genauerer Betrachtung, erkennt man daß diese Funktionen eine Klasse zurückgeben, die genau eine Methode beinhalten. In dieser Methode steht ein möglicher Handlungsablauf, der in Abhängigkeit der Parameter unterschiedlich abläuft. So wird eine Methode andere Rechenschritte einleiten, wenn sie sich in einer dynamischen Klasse befindet, als wenn sie in einer statischen Klasse ist. Ganz wichtig ist die *SuperClass* Variable. Sie beinhaltet im allgemeinen Fall die Klasse von der sie abgeleitet ist. Da aber die konstruierten Grundelemente ebenfalls Klassen sind, müssen diese zu einer Klasse zusammengeführt werden. Dies geschieht über die *SuperClass* Variable. Dadurch ergibt sich der Aufbau einer Klasse wie in Abbildung 3.3.

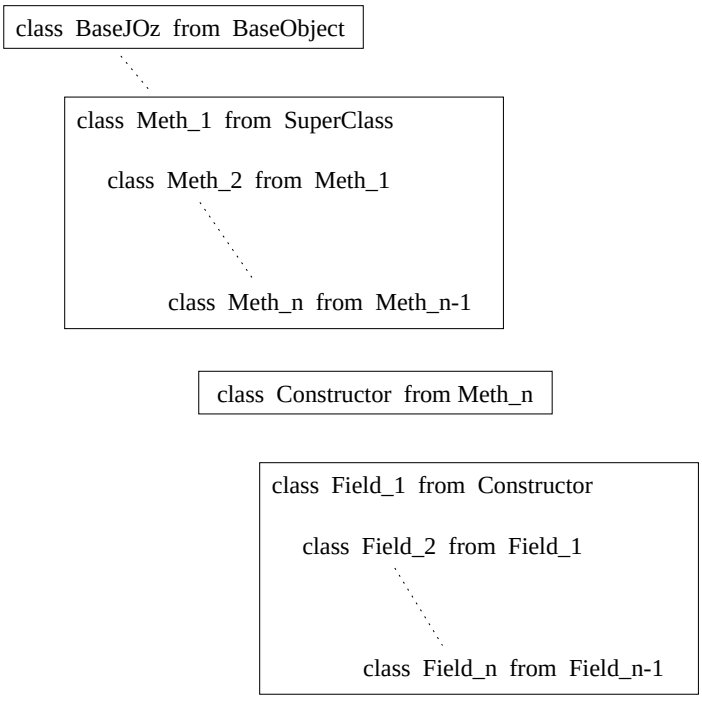

Abbildung 3.3: Oz Aufbau einer abgebildeten Java Klasse

3.4.2 Struktur eines Konstruktors

Wie oben erwähnt, besitzt Oz im Sinne von Java keinen Konstruktor. Da aber Java einen verlangt, muß Oz einen simulieren. Dazu wird die Methode *init* für alle gültigen Klassen ausgewählt.

Eine Klasse kann zwei verschiedene *init*-Methoden besitzen. Besteht die Klasse nur aus statischen Methoden, dann besitzt sie keinen Konstruktor. Als *init*-Methode wird somit Algorithmus 6 benutzt. Hier werden lediglich in der Basisklasse die Einträge *isStatic* auf *true* und *type* auf den voll qualifizierten Klassennamen gesetzt.

In der Regel ist ein Konstruktor vorhanden, so daß etwas mehr Arbeit anfällt (siehe Algorithmus 7). Zunächst werden die Argumente mit *CheckArgs* (siehe: 3.4.6) in entsprechende Protokollform gebracht. Dies geschieht in dem Objekt *Message* der Klasse *JOzMessage*. Anschließend wird diese Nachricht nach Java geschickt. Gibt Java eine Antwort, wird die übertragene Referenz und der voll qualifizierte Klassenname gespeichert. Zuletzt trägt sich das Objekt selbst unter der Referenz in eine Tabelle mit Objekten ein. War die Anforderung fehlerhaft, verschickt Java eine Exception, die aufgefangen und in eine Oz Exception umgewandelt wird, die dann ebenfalls geworfen wird. Es bleibt dem Anwender überlassen, ob er diese auffangen möchte.

Algorithmus 6 Generieren einer leeren Konstruktors

```
class $ from SuperClass
    meth init(...) = M
        BaseJOz, setIsClassStatic(true)
        BaseJOz, setType(ClassStr)
    end
end
```

3.4.3 Struktur einer Methode

Die eigentlichen Methoden im Sinne von Java, die nun konstruiert werden müssen, werden alle durch dieselbe Klasse dargestellt. Diese hat den Aufbau, der im Algorithmus 8 zu sehen ist.

Zunächst wird geklärt, ob die Methode zu einer Klasse mit ausschließlich statischen Methoden gehört. In Abhängigkeit davon verändert sich das Protokoll; ist dies der Fall, wird an Position *Id* der voll qualifizierte Klassenname, andernfalls die Referenz geschickt. Danach werden die Argumente in *CheckArgs* überprüft und in die richtige Nachrichtenform gebracht. Damit ist die zu verschickende Nachricht vollständig und ein Client übernimmt die Kommunikation. Der Client sendet die Nachricht an Java und wartet auf Antwort. Kommt diese an, wird sie in *EvalResult* (3.4.7) ausgewertet.

Zuletzt wird geprüft, ob der Anwender ein Ergebnis zurückerwartet. Dies erkennt man daran, daß er in seinem Methodenaufruf das Feature *result* angegeben hat. Ist dies der Fall, so wird das in *EvalResult* berechnete Ergebnis zurückgegeben.

Algorithmus 7 Generieren eines Konstruktors

```
class $ from SuperClass
   meth init(...) = M
      JOzClient Id Message
   in
      Message = {New JOzNet.jozMessage init(newJavaObject)}
      {Message insert(Type)}
      {CheckArgs {Record.subtract M result} Message}
      JOzClient={JOzNet.getClient}
      {JOzClient sendNewJObject(Message Id)}
      local RES Ref RefStr in
         RES={Nth Id 1}
         if RES==”RES” then
            RefStr={Nth Id 2}
            Ref={String.toInt RefStr}
            BaseJOz, setId(Ref)
            BaseJOz, setType(ClassStr)
            {Dictionary.put JavaObjDict {StringToInt RefStr} self}
         elseif RES==”EXC” then
            raise {RaiseExc Id} end
         end
      end
   end
```

Algorithmus 8 Java-Oz Methode

```
class $ from SuperClass
   meth !MethName(...) = M
      JOzClient Result Id SpecRes Message
   in
      if IsStatic==true then
         BaseJOz, getType(Id)
      else
         BaseJOz, getId(Id)
      end
      Message = {New JOzNet.jozMessage init(accessJavaMethod)}
      {Message insert(Id)}
      {Message insert(MethName)}
      {CheckArgs {Record.subtract M result Message}
      JOzClient={JOzNet.getClient}
      {JOzClient sendAccessJMethod(Message Result)}
      SpecRes=EvalResult Result
      if {HasFeature M result} then
         M.result=SpecRes
      end
   end
end
```

3.4.4 Generierte Klassen und Objekte in Oz speichern

Eine Klasse, die bei Java angefordert wurde, wird in Oz in einer Tabelle gespeichert. Dies ist vernünftig, wenn eine Klasse oft gebraucht und dementsprechend oft bei Java angefordert wird. Unnötig strapazierte Bandbreiten sowie Geschwindigkeiten der Applikationen wären die Folge. Das Merken der Klassen ist eine Abhilfe.

Es gibt noch eine weitere Tabelle. In dieser Tabelle werden generierte Objekte gespeichert. Das System muß sich die Objekte merken, damit, sollte Java eine Referenz von einem schon existierenden Objekt als Ergebnis zurückschicken, man leicht über die Referenz an das gesuchte Objekt herankommt. Eine weitere Bedeutung gibt es nicht.

3.4.5 Struktur einer Feldzugriffsmethode

Der Aufbau der Feldzugriffsmethode ist etwas einfacher als der einer Methode. Zunächst wird gemäß des Protokolls (siehe Abschnitt 3.1.2.11) die Nachricht für die Anfrage generiert. Da hier die Betrachtung der Parameter entfällt, kann die Nachricht sofort verschickt werden. Nachdem eine Antwort angekommen ist, muß der Rückgabewert aus ihr gewonnen werden.

An dieser Stelle muß begründet werden, warum der Zugriff auf Felder über eine Methode stattfindet. Die erste Idee bestand darin, daß beim Generieren einer Proxyklasse in Oz alle Felder als Features repräsentiert sein sollten. Der Zugriff auf ein solches Feature sollte eine Funktion aufrufen, die die Aufgabe hat, den aktuellen Wert von Java in Erfahrung zu bringen und ihn im Feature zu speichern. Dies würde bedeuten, daß die Funktion verloren geht, und daß nur einmal dieser Wert erfragt werden kann. Dies ist jedoch nur für Konstanten sinnvoll, nicht für variable Felder. Es ist notwendig, daß Oz zu jeder Zeit den aktuellen Wert eines Feldes erfragen kann, und dies geht nur über diese konzipierten Methoden.

3.4.6 Automatisches Erkennen der Parametertypen

Java stellt eine *Reflection API* zur Verfügung, mit deren Hilfe alle Informationen über alle Klassen erfragt werden können. Speziell in diesem Fall ist es sinnvoll, einen Zugang zu Konstruktoren und Methoden einer bestimmten Klasse zu haben. Neben der Informationsbereitstellung ermöglicht die *Reflection API* einen bestimmten Konstruktor oder eine bestimmte Methode aufzurufen. Um diese eine bestimmte Methode oder diesen bestimmten Konstruktor zu finden, müssen seine Parameter, genauer, die Klassen der Parameter, bekannt sein. Da die Nachrichten alle als Zeichenkette in Java ankommen, müssen die Klassen dieser Parameter innerhalb der Zeichenkette vorhanden sein. Dies heißt wiederum, daß die Klassen der Parameter schon in Oz erkannt werden müssen. Diese Aufgabe übernimmt *CheckArgs* (Algorithmus 9).

Da die Argumente einer Oz-Methode als ein Record aufgefaßt werden, lassen sie sich durch ein *Record.forAll* schnell behandeln. Jedes Argument wird auf seinen Typ gete-

Algorithmus 9 Oz: CheckArgs

```
proc {CheckArgs M ?Message}
   ValueList Number
in
   Number={Width M}
   {Message insert(Number)}
   ValueList = {List.make Number}
   {Record.forAllInd M
      proc {$ I Arg}
         if Arg==unit then
            {Message insert("void")}
            {Nth ValueList I "null"}
         elseif {IsBool Arg} then
            ⋮
      }
   {List.forAll ValueList
      proc {$ S}
         {Message insert(S)}
      end}
end
```

stet. Bisher können nur die in Tabelle 3.4.6 aufgeführten Typen erkannt und miteinander ausgetauscht werden.

Java Typen	Oz Typen
java.lang.Boolean (boolean)	bool
java.lang.Character (char)	string
java.lang.Byte (byte)	JavaClass.byteInt
java.lang.Short (short)	JavaClass.shortInt
java.lang.Integer (int)	integer
java.lang.Long (long)	longInt
java.lang.Float (float)	float
java.lang.Double (double)	DoubleFloat
java.lang.Void (void)	unit

In Zukunft sollen noch weitere Typen, falls notwendig, hinzugefügt werden. Besonders die Darstellung eines Arrays wird auf Benutzbarkeit überprüft. Ist ein Typ erkannt, so wird er sofort in die Nachricht eingefügt, der Wert wird in einer Liste zwischengespeichert. Sind alle Parameter abgehandelt, wird der Inhalt der Liste ebenfalls an die Nachricht angehängt. Dadurch werden erst die Typen und danach die Werte verschickt. Der Grund hierfür liegt in der wesentlich einfacheren Erkennung der Signatur einer Java Methode durch die *Reflection API*. Dieser Algorithmus erkennt selbst die AWT und Swing Objekte. Er entnimmt dann der Basisklasse die nötigen Informationen.

Wird ein Parameter angegeben, dessen Typ noch nicht unterstützt wird, wird eine *JOz-Exception* geworfen.

3.4.7 Auswertung von Ergebnissen

Nach jedem Methodenaufruf bzw. nach jedem Zugriff auf ein Feld wird ein Ergebnis von Java erwartet. Die Auswertung des Ergebnisses ist die umgekehrte Prozedur zu *CheckArgs* und wird in *EvalResult* (Algorithmus 10) bearbeitet. Ein Ergebnis kann ein normales gültiges Ergebnis sein oder eine Ausnahme, die signalisiert, daß während der Methodenausführung ein Fehler aufgetreten ist. Ist letzteres der Fall, wird die Information über die Java-Ausnahme aus der Antwort entnommen und in eine eigene Oz-Ausnahme gepackt. Der Anwender muß zur Sicherheit nur eine *javaException* abfangen, wenn er auf einen Fehler reagieren will. In dieser Ausnahme sind weitere Informationen über das genaue Auftreten enthalten. Diese können dazu benutzt werden, auf bestimmte Fehler zu reagieren.

Angenommen, daß der Normalfall eintritt, d.h. daß ein korrektes Ergebnis übertragen wird, kann der als String mitgelieferte Wert über die Typinformation in der Nachricht in den entsprechenden Typ umgewandelt werden. Bei den primitiven Typen und bei String ist das schnell gemacht. Lediglich bei Rückgabe einer Referenz kommt es zu einem etwas größerem Aufwand. Zunächst wird nachgeschaut, ob die Referenz in Oz bekannt ist, d.h. ob das Objekt in Oz schon existiert. Ist dies der Fall, dann wird das Objekt, welches unter der Referenz gespeichert ist, zurückgegeben. Andernfalls ist das Ergebnis ein neues Objekt und die gelieferte Referenz zum neuen Objekt zugehörig. Nun muß eine Abbildung des in Java vorhandenen Objektes geschaffen werden. Dazu wird die Klasse des Objektes angefordert, da der voll qualifizierte Klassenname mit dem Ergebnis übertragen wurde. Schließlich kann man instanziieren, einige Eintragungen machen und das Objekt in die Verwaltungstabelle einfügen. Zuletzt wird das neu kreierte Objekt zurückgegeben. Im Grunde ist dieser Vorgang unkompliziert.

3.5 Das Arbeitszentrum in Java

Bisher haben wir besprochen, wie Nachrichten in Oz generiert und verschickt bzw. wie aus den Informationen in den Nachrichten zur Laufzeit kreierte Klassen strukturiert werden. Nun wird geklärt, wie Java diese Nachrichten entgegen nimmt, und mit welchen Aktionen Java reagiert. Insbesondere ist es interessant, wie eine Klasse in Nachrichtenform gebracht, wie ein Objekt instanziiert, wie eine Methode ausgeführt, und wie auf ein Feld zugegriffen wird. Dies sind die Kernpunkte des Kapitels.

3.5.1 Verwaltung

Genauso wie auf der Oz-Seite die Objekte in einer Tabelle unter der Referenz gespeichert sind, sind sie auch in Java unter der selben Referenz in einer Tabelle gespeichert. Die Referenz ist der Schlüssel, unter dem die Objekte in der Tabelle abgelegt werden. Da nur Java die gewünschten Objekte instanziiert, bestimmt Java auch den Schlüssel; dieser ist ein Integerwert, der bei jeder Instanziierung um eins erhöht wird.

Algorithmus 10 Oz: EvalResult

```
fun {EvalResult ResArgs}
   Type SpecType Res Key JClass NewObject
in
   if {Nth ResArgs 1}==”RES” then
      Type={Nth ResArgs 2}
      Res={Nth ResArgs 3}
         if Type==”void” then
            unit
         elseif Type==”boolean” then
            ⋮
         elseif Type==”object” then
            SpecType={Nth ResArgs 4}
            Key={StringToInt Res}
            if {Dictionary.member JavaObjDict Key} then
               {Dictionary.get JavaObjDict Key}
            else
               JClass={NewClass SpecType}
               NewObject={New JClass noop}
               {NewObject setType(SpecType)}
               {NewObject setId(Key)}
               {Dictionary.put JavaObjDict Key NewObject}
               NewObject
            end
               ⋮
end
```

3.5.2 Anfrageauswertung

Jede Nachricht, die von Oz kommt, führt an erster Position ein Aktionswort. Dieses erste Wort wird in der Methode *analyzeRequest* untersucht. In seiner Abhängigkeit werden anschließend verschiedene Aktionen in Gang gesetzt. Erscheint die Anweisung, die Kommunikation zu schließen, macht Java dies unverzüglich. Eine weitere kurze Anweisung ist die Information über den Port, an dem Oz seinen Server installiert hat. Diese Information merkt sich Java und antwortet mit einer Bestätigung. Kommt eine Anfrage mit einem unbekannten Aktionswort an, dann wird eine Ausnahme generiert und nach Oz verschickt. Diesen Fehler sollte nur der Entwickler der Software zu Gesicht bekommen. Die restlichen umfangreichen Anforderung werden in den nächsten Abschnitten im Detail besprochen.

3.5.3 Anfrage: Klasse

Bei allen wichtigen Anfragen, wie z.B. Klasse, Objekt, Methode oder Feld, sind die wichtigsten Programmierinstrumente die *Reflection API* und die Klasse *Class*. Mit ihrer Hilfe kann man zur Laufzeit Informationen über eine Klasse, einen Konstruktor, eine Methode oder sogar ein Feld einholen. Ein Konstruktor, eine Methode und ein Feld sind ebenfalls Klassen, die in der *Reflection API* enthalten sind.

Um an Informationen über eine Klasse zu gelangen, muß man wissen, wie die Klasse genannt wird. Hier wird der voll qualifizierte Name einer Klasse als String verlangt, wie z.B. „java.lang.Object". Diesen Namen kennt Oz und gibt ihn bei seinen Anfragen an. Nun sucht Java mit *Class.forName* nach der gewünschten Klasse. Ist diese gefunden, dann kann weiter mittels *Class.getConstructors* nach Konstruktoren gesucht werden. Sind keine da, dann wird „none" in die Nachricht geschrieben. Andernfalls wird der voll qualifizierte Klassenname angegeben. Anschließend werden mit *Class.getDeclaredMethods* und *Method.getName* alle Methodennamen herausgesucht und in die Nachricht eingetragen. Überladene Methoden werden nur einmal eingetragen,da nur Methodennamen ohne die zugehörigen Parameter verschickt werden. Derselbe Prozess wird nochmal mit *Class.getDeclaredField* und *Field.getName* für Felder wiederholt.

Dies sind die wesentlichen Schritte für eine Klassennanfrage und für den Aufbau eines Antwortprotokolls (siehe 3.1.2.6).

3.5.4 Anfrage: Objekt generieren

Kommt die Anfrage, ein neues Objekt zu generieren, wird in der Nachricht der voll qualifizierte Klassenname des Objektes mitgeschickt. Anhand dieses Namens sucht das System mit *Class.forName* die Klasse. Wird diese gefunden, bieten sich zwei Möglichkeiten fortzufahren. Wurden keine Parameter mitgeschickt, wird versucht, mittels *Class.newInstance* direkt zu instanziieren. Andernfalls muß man sich zunächst mit den Parametern beschäftigen. So werden erst die Klassen der Parameter und da-

Algorithmus 11 Java: Objekt generieren

```
private JOzMessage doNewJavaObject(StringTokenizer tokens) {
        ⋮
    try {
        objClass=Class.forName(ClassName);
        if (keineParameter) {
            newObject=objClass.newInstance();
            HashKey = jozObjectTable.insert(newObject);
            rMessage.insert(String.valueOf(HashKey));
            return rMessage;
        } else {
            Class[] parameterTypes=getClassArgs(anzPar,tokens);
            Object[] arguments=getObjectArgs(anzPar,parameterTypes,tokens);
            objConstr=objClass.getConstructor(parameterTypes);
            newObject=objConstr.newInstance(arguments);
            HashKey = jozObjectTable.insert(newObject);
            return rMessage;
        }
    } catch (NoSuchMethodException e) {
        try {
            Vector overCons=getOverConstructors(objClass);
            Constructor objConstr=filterOverCons(overCons,parameterTypes);
                ⋮
        } catch (Throwable e2)
            {return buildException(e2);}
    }
}
```

nach die Werte der Parameter aus den gelieferten Angaben in *tokens* ermittelt (siehe 3.5.6). Um den richtigen Konstruktor zu ermitteln, müssen die Klassen der Parameter bekannt sein. Erst danach wird versucht, durch *Constructor.newInstance* ein neues Objekt zu schaffen. Scheitert der Versuch an dieser Stelle, so wird ein zweiter Anlauf gestartet, indem aus allen Konstruktoren der Klasse der richtige herausgefiltert wird. Wie diese Filterung funktioniert, entnimmt man dem Abschnitt 3.5.7. Findet sich immer noch kein geeigneter Konstruktor, liegt ein Fehler des Anwenders vor. Dies wird ihm durch eine Ausnahme mitgeteilt. Wurde jedoch der Konstruktor gefunden, wird ein zweites Mal versucht, mit *Constructor.newInstance()* ein neues Objekt zu instanziieren. Gelingt dieser Versuch, wird das Objekt in die Verwaltungstabelle mit aufgenommen und aus seiner Referenz eine Antwort für Oz gebaut (siehe 3.5.9), die die Rückgabe der Methode ist. Der grobe Ablauf des Vorgangs, steht in Algorithmus 11.

Einen anderen Weg als die allgemeine Objektgenerierung beschreitet die Instanziierung von Adapterobjekten. Hier wird direkt abgefragt, ob eine bestimmte Klasse gewünscht wird. Trifft dies auf eine Adapterklasse zu, wird diese direkt instanziiert. Hier wird also die *Reflection API* nicht benutzt. Allerdings kann ich mir sehr gut vorstellen, die Generierung eines Adapterobjekts in die allgemeine Objektgenerierung zu integrieren. Ob dies möglich ist, zeigen weitere Tests.

3.5.5 Anfrage: Methode ausführen

Mit Objekten alleine kann man nicht leben, man muß mit ihnen arbeiten können. Diese Aufgabe übernehmen Methoden. Java kennt zwei Arten von Methoden: statische und dynamische. Statische Methoden werden auch *class methods* bezeichnet. Sie werden gewöhnlich zur Erledigung klassenspezifischer Angelegenheiten verwendet. Dies geschieht auf klassenbezogenen Datenfeldern (*class variables*) und nicht auf spezifischen Instanzen dieser Klasse. Da es zwei Arten von Methoden gibt, muß Java erken-

Algorithmus 12 Java: statische/dynamische Methode bestimmen

```
private JOzMessage doAccessJavaMethod(StringTokenizer tokens) {
        ⋮
    if (jozObjectTable.containsKey(class_or_id)) {
        idObj=jozObjectTable.get(classORid);
        idObjClass=idObj.getClass();
        return invokeMethod(idObjClass,idObj,idMeth,idAnzArgs,tokens);
    } else {
        try {
            idObjClass=Class.forName(classORid);
            return invokeMethod(idObjClass,null,idMeth,idAnzArgs,tokens);
        } catch (Throwable e)
            {return buildException(e);}
    }
}
```

nen, ob Oz eine statische Methode verlangt. Dies ist bei üblichen Klassen auch kein Problem in der Verarbeitung. Schwierigkeiten gibt es erst, wenn auf eine Klasse zu-

gegriffen wird, deren Methoden alle statisch sind. Diese Klassen haben somit keinen Konstruktor, was zur Folge hat, daß es ebenfalls keine Referenz dazu gibt. Anstelle der Referenz verschickt Oz den voll qualifizierten Klassennamen. Welche Angaben Oz verschickt hat, erkennt Java daran, daß es prüft, ob diese Angabe ein Schlüssel zur Objektverwaltungstabelle ist. Dies ist fehlerfrei, da der Schlüssel eine Stringdarstellung eines Integerwertes ist und keine Klasse mit dem vollständigen Namen nach einem Integerwert benannt wird. Ist die Angabe eine Referenz, gibt die Verwaltungstabelle das entsprechende Objekt zurück. Dieser Sondierungsorgang ist in Algorithmus 12 dargestellt. An dieser Stelle sind wir in der Startposition, um eine Methode auszuführen. Zunächst wird die Klasse gebraucht, in der die Methode steckt. Bei einer ,,statischen" Klasse wird der Name im Protokoll mitgeschickt, und die Klasse kann mit *Class.forName* gefordert werden. Im anderen Fall bekommt man die Klasse über das erhaltene Objekt mit *Object.getClass*.

Nun beschreiten beide Methodenaufrufe denselben Weg. Als nächstes müssen die Parameter vorbereitet werden, d.h. die Klassen der Parameter und die Werte der Parameter werden in Erfahrung gebracht (siehe 3.5.6). Mit den Parameterklassenangaben ist Java in der Lage, die Methode mit *Class.getMethod* zu erhalten. Jetzt sind alle notwendigen Variablen für einen Methodenaufruf vorhanden. Mit *Method.invoke* wird die Methode ausgeführt. Dabei sollte beachtet wertden, daß für *Method.invoke* das Objekt, welches die Methode ausführen will, als Parameter angegeben wird. Bei einer Klasse mit nur statischen Methoden wird **null** als Objektparameter angegeben.

Algorithmus 13 Java: Methodenaufruf

```
private JOzMessage invokeMethod(Class idObjClass, Object idObj,
                                String idMeth, int idParam,
                                StringTokenizer tokens) {
    ⋮
  try {
    Class[] parameterTypes=getClassArgs(idParam,tokens);
    Object[] arguments=getObjectArgs(idParam,parameterTypes,tokens);
    Method idObjMethod=idObjClass.getMethod(idMeth,parameterTypes);
    Class resType=idObjMethod.getReturnType();
    Object Result=idObjMethod.invoke(idObj, arguments);
    return checkInvokeResult(resType, Result);
  } catch (NoSuchMethodException e) {
    Vector overMeths=getOverMethods(idObjClass, idMeth);
    Method idObjMethod=filterOverMeths(overMeths,parameterTypes);
      ⋮
}
```

Wie schon bei der Instanziierung von Objekten (siehe 3.5.4) kann der erste direkte Aufruf scheitern. So wird bei falschen Klassenangaben keine entsprechende Methode gefunden. Hier wird wiederum ein Filterungsprozeß in Gang gesetzt (siehe 3.5.7). Schlägt auch der fehl, muß der Anwender den Fehler begangen haben. Eine Ausnahme macht ihn darauf aufmerksam. Im besten Falle funktioniert alles reibungslos und der Methodenaufruf liefert ein Ergebnis, welches in eine entsprechende Antwortnachricht umgewandelt wird (siehe 3.5.5.1). Algorithmus 13 zeigt den gesammten Vorgang eines

Methodenaufrufes.

3.5.5.1 Ergebnis analysieren

Die Ergebnisanalyse benötigt für ihre Arbeit den Rückgabetyp einer Methode und das Ergebnis. Den Rückgabetyp erhält man durch *Method.getReturnType*. Dieser wird einzeln mit den primitiven Typen, die das Protokoll bereitstellt, verglichen, um festzustellen, ob der Rückgabetyp einer davon ist. Ist dies der Fall, wird eine Antwort für Oz mit dem Typen und dem Ergebnis generiert. Ist das Resultat ein Objekt, wird in der Verwaltungstabelle nachgeschaut, ob das Objekt schon vorhanden ist. Falls ja, wird die Referenz zurückgeschickt, andernfalls ist das Ergebnis ein neues Objekt, welches eine neue Referenz bekommen und in die Tabelle eingetragen werden muß, bevor die Antwort gemäß Protokoll zurückgeschickt wird. Den Source hierfür spare ich mir, da es ein pendant zu Algorithmus 10 ist, der die Antwort in Oz auswertet.

3.5.6 Klassen und Werte von Parametern ermitteln

Sowohl für die Suche nach einem geeigneten Konstruktor, als auch für die Suche nach der geeigneten Methode wird ein Class-Array gebraucht, welches die Typen, ausgedrückt in Klassen, beinhaltet. Anhand dieses Arrays und dem Namen der Methode erkennt Java, wonach genau gesucht wird. Das Ermitteln der Klassen ist relativ einfach aus der gelieferten Nachricht zu entnehmen, da diese schon als String enthalten sind. Hier wird per Stringvergleich erkannt, welche Klasse gewünscht wird. Bei primitiven Typen wird daraufhin das Array mit deren Klasse, wie z.B. *int.class* oder *boolean.class*, gefüllt. Bei nichtprimitive Typen, also bei Objekten, wird über *Class.forName* die Klasse ermittelt.

Auf diesem Array basierend wird ein dazugehöriges Wertearray erstellt. Hier wird ähnlich verfahren, jedoch werden nicht Strings sondern Klassen verglichen. Trifft eine Klasse zu, dann wird für das Wertearray an der selben Position ein Wert generiert, wie z.B. „new Boolean(...);". Ist die Klasse ein nichtprimitiver Typ, also ein Objekt, dann wird anhand der mitgeschickten Referenz das Objekt aus der Verwaltungstabelle eingesetzt.

3.5.7 Konstruktoren und Methoden Filtern

Wie schon bei der Objektinstanziierung und Methodenausführung erwähnt, kann es vorkommen, daß ein Aufruf nicht direkt funktioniert, und daß in Folge dessen die richtige Methode gefiltert werden muß (siehe Algorithmus 14).

Dahinter steht folgende Überlegung. Angenommen wir haben ein grafisches Objekt der Klasse *JPanel*. Um ein weiteres grafisches Objekt, beispielsweise *JLabel* in dieses Panel einzufügen, wird die Methode *JPanel.add* aufgerufen. Diese Methode erwartet als Argument ein Objekt der Klasse *java.awt.Component*. In der normalen direkten

Programmierung kann man problemlos *add* mit *JLabel* als Parameter aufrufen, da *JLabel* eine Unterklasse von *java.awt.Component* ist. Beim indirekten Aufruf über die *Reflection API* funktioniert das nicht auf Anhieb. Dies liegt daran, daß in der Anweisung ,,Methode ausführen" ein Parameter der Klasse *JLabel* angegeben ist, woraufhin eine Methode *add* mit einer Klasse *JLabel* als Parameter gesucht wird. Da es solch eine Methode nicht gibt, wird eine Ausnahme geworfen und ein Filtervorgang gewünscht. Dieser Filtervorgang muß nun herausfinden, ob *JLabel* eine Unterklasse von *Component* ist. Falls ja, wird das Array der Parametertypen modifiziert, d.h. *JLabel* wird durch *Component* ersetzt.

Algorithmus 14 Java: Methoden filtern

```
private Method FilterOverMeths(Vector overMeths, Class[] parTypes) {
    Method overMeth=null;
    Class[] overMethParTypes=null;
    for (int i=0;i<overMeths.size();i++) {
        overMeth=(Method)overMeths.elementAt(i);
        overMethParTypes=overMeth.getParameterTypes();
        if (parTypes.length != overMethParTypes.length) {
            overMeths.removeElement(overMeth);
            i–;                          //Positionsausgleich im Vector
        } else {
            for (int j=0; j<parTypes.length; j++) {
                if (!overMethParTypes[j].isAssignableFrom(parTypes[j])) {
                    overMeths.removeElement(overMeth);
                    i–;
                    break;
                }
            }
        }
    }
    if (overMeths.isEmpty()) {
        return null;
    } else {
        return (Method)overMeths.firstElement();
    }
}
```

Am Anfang ist ein Vector gegeben, der alle Methoden mit dem gesuchten Namen einer bestimmten Klasse und aller Superklassen enthält. Dies kann mehrere überladene Methoden, sowie eine einzelne Methode beinhalten. Der erste Schritt der Filterung besteht darin, die Anzahl der übergebenen Argumente mit der der Methoden im Vector zu vergleichen. Stimmt die Anzahl nicht überein, wird die Methode aus dem Vector herausgenommen. Im zweiten Schritt werden die Argumente der verbleibenden Methoden verglichen. Ist im Protokoll ein Parameter des Typs String angegeben, muß die gesuchte Methode an derselben Stelle den gleichen Typ haben. Alle anderen Methoden werden aus dem Array herausgenommen. Der dritte Schritt ist der Test auf Ableitbarkeit. So kann es sein, daß der Typvergleich negativ ausfällt, der Test auf Ableitbarkeit jedoch positiv ist. Stimmt auch dieser nicht, so wird die Methode aus dem Vector entnommen. Am Ende sollte genau eine Methode, nämlich die gesuchte, im Vector übrig bleiben. Falls nicht, liegt ein Fehler des Anwenders vor.

3.5.8 Anfrage: Zugriff auf Felder

Bei einem Zugriff auf ein Feld muß man wie bei der Methode unterscheiden, ob auf eine Instanz oder auf ein klassenspezifisches Feld, d.h. auf ein *class field* zugegriffen wird. Im Falle eines klassenspezifischen Feldes wird keine Referenz, sondern *unit* zusammen mit dem voll qualifizierten Klassennamen mitgeschickt. Im Gegensatz dazu wird auf eine Instanzvariable zugegriffen, indem aus der Nachricht zunächst die Referenz des zugehörigen Objektes herausgelesen wird. Dieses Objekt wird anschließend aus der Hashtabelle entnommen und seine Klasse bestimmt. Nun verläuft der Vorgang gleich. Über die Klasse kann mittels *Class.getField* das gewünschte Feld beantragt werden. Weiterhin braucht man die Klasse des Feldes, welche man mit *Field.getType* erhält. Zuletzt wird mit *Field.get* und der Instanz bzw. *null* als Argument der gesuchte Wert des Feldes gefunden.

Auch hier gilt, daß bei einem Fehler eine Ausnahme geworfen und nach Oz geschickt wird.

Algorithmus 15 Java: Zugriff auf ein Feld

```
private JOzMessage doAccessJavaField(StringTokenizer tokens) {
        ⋮
    try {
        if (isReferenz("unit")) {
            Object idObj = null;
            String idClassStr = tokens.nextToken();
            Class idObjClass = Class.forName(idClassStr);
        } else {
            Object idObj = jozObjectTable.get(Referenz);
            Class idObjClass = idObj.getClass();
        }
        Field field = idObjClass.getField(Feldname);
        Class fieldClass = field.getType();
        Object fieldValue = field.get(idObj);
        return checkInvokeResult(fieldClass, fieldValue);
    } catch (Throwable e)
        {return buildException(e);}
}
```

3.5.9 Rückgabewert erstellen

Am Ende einer Anforderung muß eine einheitliche Antwort produziert werden. Hierbei können zwei verschiedene Ergebnisse vorliegen. Entweder ist es ein korrektes Ergebnis, welches eine Antwort, wie sie in Kapitel 3.1.2 beschrieben ist, generiert, oder eine Ausnahme ist aufgetreten sein. Aus ihr wird die Information, was für eine Ausnahme und was die Ursache für die Ausnahme ist, entnommen, und in die Antwort für Oz eingepackt.

3.6 Event Handling

Das Benutzen von grafischen Java-Elementen ist ein sehr mächtiges Feature. Dieses reicht aber nur für das Anzeigen von Informationen. Was fehlt, ist die Interaktivität mit dem Benutzer. So kann nicht reagiert werden, wenn ein Ereignis stattgefunden hat, z.B. das Drücken eines Buttons durch den Benutzer.

Das Event Handling in Java hat eine eigene Logik, die verstanden werden muß, damit die Möglichkeiten, die dieses Projekt zur Verfügung stellt, genutzt werden können. Gleichzeit kann man den Grundlagen die Grenzen entnehmen. Da dies ein ganz besonders wichtiger Punkt ist, beschäftigt sich der erste Abschnitt mit den Grundlagen des Java Event Handlings auf der Basis von JDK 1.1. Darauf aufbauend erläutern die nachfolgenden Abschnitte die konkrete Implementierung und ihre Schwierigkeiten.

3.7 Der Lebenszyklus eines Java Events

Bevor die einzelnen Schritte eines Events aufgezählt werden, müssen wir zunächst definieren, was eigentlich ein Event ist:

- Events bieten eine Möglichkeit, mit der ein Objekt eine Methode eines anderen Objektes mit einer Verzögerung aufruft. Der Aufrufer kümmert sich nicht um die anstehende Arbeit, sondern gibt diese weiter an eine Queue und setzt seine eigene Arbeit fort. Der Aufrufer ist demnach der Entstehungsort des Events und wird *Event Source* genannt.
- Events bieten eine flexible Möglichkeit, zur Laufzeit genau zu bestimmen, welche Objekte ihre Methoden aufzurufen haben, wenn etwas Interessantes passiert. Diese Objekte bestimmen das Ziel, an das ein Event gelangt und werden *Event Target* genannt. Dies können alle Objekte sein, die ein entsprechendes Listener Interface implementieren oder eine Adapterklasse eines Listeners erweitern.
- Events bieten eine Möglichkeit, den Aufgerufenen entscheiden zu lassen, wann er aufgerufen werden will, anstatt dies den Aufrufer entscheiden zu lassen. Dies bedeutet, daß eine vom Anwender implementierte Methode, sei es als Implementierung eines Interfaces oder innerhalb einer Adapterklasse, erst dann sinnvoll ausgeführt wird, wenn sie Anweisungen enthält. Besteht die Methode aus einem leeren Rumpf, so ist dies gleichzusetzten mit der Tatsache, daß der Anwender kein Interesse an diesem Ereignis hat.

Insgesamt gibt es drei Stellen, an denen auf Events reagiert werden kann:

- Üblicherweise reagiert man auf Events in sogenannten Listenern. Ein Listener kann jedes beliebige Objekt sein, zum Beispiel ein Frame, die Source Komponente selbst, ein nicht GUI Objekt, ein Kontainer etc. Ein Listener kann von verschiedenen Quellen aus aufgerufen werden, wobei eine Quelle mehrere verschiedene Listener aufrufen kann.

- Bei jedem Ereignis, welches in der *eventMask* registriert bzw. für welches ein Listener innerhalb dieser Komponente angemeldet ist, wird die Methode *processEvent* in dem Objekt, in dem es entstanden ist, aufgerufen. Sollte diese Methode überschrieben worden, so muß man sicher stellen, daß *super.processEvent* aufgerufen wird, um alle Events an die spezifischen Handler *processXXX* weiterzuschicken bzw. die Listener alarmiert.

- Jedes spezifische Event wird von einem eigenen Handler bearbeitet. Dieser hat den Namen *processXXX* und wird von *processEvent* aufgerufen.

In der Regel reagiert der Anwender auf ein Event, indem er ein Listenerinterface implementiert oder von einer Adapterklasse ableitet, und diese Instanz mittels *source.addXXXListener(target)* bei der entsprechenden Quelle anmeldet. Diese Quelle führt eine Liste über alle Listener, die bei ihr aufgeführt sind.

Nun widmen wir uns dem Lebenszyklus eines Events. Events können in vier verschiedenen Quellen geboren werden:

- *native event handling* kreiert einen Input Event (z.B. KeyEvent)

- *native GUI Objekte* kreieren *semantic events* (z.B. ActionEvent)

- Methoden einer Komponente, wie *Component.setVisible(true)* kreieren ein *ComponentEvent* mit einer *id* belegt mit *COMPONENT_SHOWN*

- eine Applikation selbst kann ein Event kreieren

Für das Verständnis weiterer Vorgänge , müssen noch weiter Begriffe geklärt werden.

peer component Zu jeder Komponente gehören drei Objekte, so besitzt zum Beispiel ein Button standardmäßiges AWT Button Object, eine Abbildung des AWT Button Objekts in ein native GUI Objekt (*peer*), und eventuell noch ein verstecktes Button Objekt, welches nur intern für das native GUI sichtbar ist. Das Verhalten dieser bestimmten Komponente kann stark vom *peer* abhängen. Das *peer* Objekt eines Buttons zum Beispiel beinhaltet einen platformabhängigen Code, um sich an den spezifischen *button-drawing code* des *native GUI* von Windows 95 zu hängen, wohingegen das AWT Button Objekt identisch auf allen Platformen ist. Das gleiche Java Programm wird sich etwas anders auf verschiedenen Platformen verhalten.

lightweight components Man kann eigene einfache Komponenten durch Überschreiben einiger Methoden der offiziellen Komponenten kreieren. Diese reinen Java Komponenten, zu denen es kein *peer object* gibt, werden als *lightweights* bezeichnet. Da der gesamte Source in Java vorhanden ist, verhält sich die Komponente auf jeder Platform gleich. Sie hat ihre eigenen Zeichenmethoden, basierend auf den Subkomponenten wie *Panel* oder *Canvas*. Dies gibt ihnen volle Kontrolle über ihr Aussehen.

heavyweight components Im Gegensatz zu den *Lightweights* haben *heavyweight components* ein *native peer GUI object*, welches auf jeder Platform anderes Verhalten aufweist. Dies heißt jedoch, daß sie keine volle Kontrolle über ihr eigenes Aussehen haben, da sie sich den entsprechenden *native GUI* Regeln fügen. Hierzu gehören die üblichen Standardkomponenten.

In platformabhängigen Programmiersprachen muß ein applikationsspezifischer Code, der Events in einer Schleife abfängt und bearbeitet, geschrieben werden. Das AWT wird damit automatisch, für den Anwender unsichtbar, in *EventDispatchThread.run* fertig. Es liest einen Strom von native GUI Ereignisnachrichten und übersetzt diese in Java Events, die in die SystemEventQueue eingefügt werden.

Diese befindet sich in *Toolkit.getDefaultToolkit().getSystemEventQueue()*. Das Einfügen geschieht mit der Methode *postEvent(Event e)*. In diese Schleife kann jedes Event eingefügt werden, also auch ein neu definiertes. Desweiteren kann neben dem native GUI auch der Anwender ein Ereignis auslösen.

Die meisten Events entstehen im native GUI, und betreten das Java System auf einem der folgenden Wege:

1. Das Auftreten interessanter Aktionen im GUI kommt als ein einzelner Strom von Nachrichten mit der Erwähnung der native GUI Komponente an. Ein typischer, versteckter native Code ruft in einer Schleife immerzu eine *getMessage()* Methode auf, die die Ankunft von native GUI Ereignissen abfragt. Das AWT benutzt eine Hashtabelle, um Referenzen der native GUI Komponente in die korrespondierenden Java Komponenten und Peers umzuwandeln, und bildet entsprechende Java Events und fügt diese an das Ende der SystemEventQueue ein.

2. Java Peer Komponenten generieren Ereignisse basierend auf den Informationen vom native GUI, die gleichfalls an die SystemEventQueue angehängt werden.

Lightweight Components generieren zwar auch Ereignisse, schicken diese jedoch direkt an sich selbst, ohne den Umweg über die SystemEventQueue.

Der Dispatcher, liest asynchron aus dieser Queue die einzelnen Events aus, und schickt sie an die korrespondierenden Java GUI Komponenten, indem *event.getSource().processEvent(event)* aufgerufen wird, unter der Vorraussetzung, daß das Event in der *eventMask* enthalten ist, d.h. die Komponente ist gewillt, dieses Event anzunehmen. Man kann mit den Methoden *enableEvents* und *disableEvents* genauer bestimmen, welche Events ausgeschlossen bleiben sollen. Gleichfalls wird geprüft, ob die Komponente einen zum Ereignis passenden Listener besitzt. Ist dies der Fall, so wird das Ereignis auch falls es durch die *eventMask* geblockt sein sollte, zugestellt. Die Information über die Source Komponente führt das Event mit sich mit.

Die Effizienz des Dispatching Mechanismus leitet sich aus der nicht vorhandenen Notwendigkeit, Informationen aus einer Art Verwaltungstabelle herauslesen bzw. verschachtelte Abfragen (mittels switch) machen zu müssen, ab.

Eine erste Klassifizierung von Ereignissen findet in *processEvent* statt. Dort werden in Abhängigkeit des Typs die Events an die spezifischen *processXXX* Handler weitergeleitet. Diese Handler können das Ereignis weiter unterteilen. Letztendlich benachrichtigen sie die zugehörigen Listener in der Reihenfolge, in der sie angemeldet wurden. Die Listener werden in der Struktur *AWTEventMultiCaster* verwaltet, die einen generischen Code beinhaltet, den die meisten Komponenten zur Implementierung von *addXXXListener* und der Benachrichtigung eben dieser nutzen. Die Benachrichtigung besteht aus dem Aufruf der zum Event zugehörigen Methode. In diesem Aufruf wird eine geklonte Kopie des Events als Argument übergeben. Leider verschwendet dieser Vorgang viel CPU Zeit und verursacht viel Garbage zur Laufzeit. So liegt die Absicht darin, die Events so zu gestalten, daß sie praktisch nur ,,leseberechtigt" sind, d.h. daß sie nicht verändert werden, und daß keine Referenz auf sie gespeichert wird.

Man muß beachten, daß das gesammte System während einer Eventbehandlung stillsteht und nicht in der Lage ist, weitere Events zu behandeln, solange der Listener seine Arbeiten nicht abgeschlossen hat. Dauert die Aktion mehr als einige Millisekunden, hat man folgende Alternativen:

- Einen seperaten Thread starten, der die Arbeit erledigt und sofort vom Event Handler zurückkehrt. Hier muß man vorsichtig sein, da Swing Komponenten nicht Thread sicher sind. Diese dürfen nur mit dem orginalen Dispatching Thread betreten werden.
- Man kann auch einen Teil der Arbeit ausführen und einen künstlichen Event erzeugen, um sich selbst zu einem späteren Zeitpunkt daran zu erinnern, die Arbeit fortzuführen.
- Einen zweiten Thread erzeugen, der die Events der SystemEventQueue entnimmt.

Das Leben eines Events endet mit der Abarbeitung durch alle Listener, falls keine weitere Bindung vorhanden ist. In diesem Fall wird das Event Objekt mittels Garbage Collection aus dem Speicher entfernt.

3.7.1 PaintEvent

Ein Event, das sich nicht durch einen Listener abfangen läßt, ist das PaintEvent. Dieses erscheint in der SystemEventQueue, wenn ein *paint* oder *update* der jeweiligen Source Komponente aufgerufen werden soll. Das PaintEvent ist komplexer als es aussieht. Ich will es hier anhand dreier Methoden vorstellen:

update Diese Methode hat die Aufgabe mittels der *fillrect* Methode die Komponente in der Hintergrundfarbe auszufüllen. Anschließend wird *paint* aufgerufen. Oft wird *update* überschrieben, um das Bildschirmflickern zu reduzieren.

paint Diese Methode übernimmt das eigentliche Zeichnen, wobei sie nicht den Hintergrund löscht. Ebensowenig ist diese Methode dafür verantwortlich, daß mögliche

children Komponenten gezeichnet, noch daß deren Zeichnungen geregelt werden. Zeichenaufrufe geschehen in Abhängigkeit des native GUI, welches paint zu jeder Zeit aufrufen kann, gegebenfalls auch mehrmals hintereinander, jedesmal mit einer anderen *clipping region.* Bezieht sich der Aufruf auf einen Kontainer, so ruft dieser die paint Routine jeder registrierten Komponente. Aus Effizienzgründen wird ein Aufruf unterlassen, befindet sich die Komponente definitiv außerhalb der *clipping region.* Das native GUI behandelt heavyweight Komponenten ohne Einbeziehung des AWT.

repaint *repaint()* beantragt ein Löschen und eine Neuzeichnung (update) nach einer kurzen Verzögerung. Dies stellt mehr Zeit zur Verfügung, um Veränderung des Bildschirms zu berechnen, bevor diese gezeichnet werden. Ohne diese Verzögerung würde man ständig mehrmals pro Sekunde den Bildschirm neu zeichnen, sobald eine Kleinigkeit geändert würde. Bei einem Aufruf von repaint() wird eine Nachricht an das native GUI geschickt, mit dem Vorschlag, irgendwann in nächster Zeit, zu einem günstigen Zeitpunkt, eine Neuzeichnung zu starten. Das native GUI merkt sich diese Anfrage, reiht sie aber nicht in die SystemEventQueue ein. Wenn Fenster verdeckt oder wieder aufgedeckt werden, entscheidet das native GUI selbst, ob bestimmte Komponenten oder Teile von Komponenten ebenfalls neu gezeichnet werden müssen. Das native GUI fügt diese Anfragen zusammen und eliminiert alle Duplikate. Es kann sogar die Reihenfolge ändern, so daß Hintergrundkomponenten immer vor den darüberliegenden gezeichnet werden.

Das native GUI zeichnet einige Komponenten selbst neu und generiert indirekt *Component.COMPONENT_RESIZED*, *PaintEvent.UPDATE* und *PaintEvent.PAINT* Ereignisse, um die Java Seite damit zu beauftragen, eigene Berechnungen diesbezüglich durchzuführen. Diese Events werden in die SystemEventQueue eingereiht. Sie bestimmen den Zeitpunkt der Aufrufe von update und paint Methoden der betroffenen Komponenten.

Was aber bestimmt den Aufruf von repaint? Da ist eine Reihe von Ursachen:

1. *invalidate()* markiert einen Kontainer, und seine Unterkontainer mit dem Befehl, auf dem Bildschirm ausgelegt zu werden. Manchmal wird invalidate() von der Applikation direkt aufgerufen, aber in der Regel geschieht der Aufruf als Seiteneffekt von Hinzufügen oder Herausnehmen eine Komponente, bzw. wenn die Komponente sichtbar bzw. unsichtbar gemacht wird. Das AWT merkt selbst, daß ein Aufruf unterdrückt werden soll, sollte z.B. eine Komponente schon sichtbar sein. Desweiteren wird invalidate ausgeführt, wenn die Größe oder Position der Komponente mit setSize(), setLocation() oder setBounds() verändert wurde. Es wird ebenfalls sofort aufgerufen, tritt ein COMPONENT_RESIZED Ereignis ein. Ein Selbstaufruf von invalidate führt zu keinen weiteren Aufrufen von repaint oder validate.
2. *validate* ist in der Regel als *pack* bekannt. Dieses berechnet das Layout neu, sollte es notwendig sein, um auf neue Größen und Positionen aller Komponenten im Kontainer zu reagieren. Meistens wird es von einer Applikation aufgerufen, sobald ein Fenster oder ein anderer Kontai-

ner zusammengestellt wurde. Dies geschieht jedoch noch vor dem Aufruf Frame.setVisible(true). validate() wird ebenfalls im zweiten Schritt in der Bearbeitung eines COMPONENT_RESIZED Ereignisses aufgerufen. Der Selbstaufruf bewirkt wiederum nichts.

3. *setVisible(true)* angewendet auf Kontainer bewirkt in der Regel einen Aufruf von validate(). Es ist jedoch sicherer, den Aufruf von validate() separat zu tätigen. Ist die Komponente noch nicht sichtbar, wird setVisible(true) auf Komponenten angewendet, ein invalidate() auf den *parent* Kontainer und alle zugehörigen Kontainer ausführen. Ist der Kontainer noch sichtbar, wird ein setVisible(false) angewendet auf Kontainer, ein invalidate() auf den übergeordneten Kontainer und alle daran angeschlossenen ausführen. Ist die Komponente noch sichtbar, wird ein setVisible(false) angewendet auf Komponenten, einen Aufruf von invalidate() des übergeordneten Kontainers und alle damit betroffenen ausgeführen. setVisible führt, falls notwendig, ein repaint durch.

4. *repaint()* führt eigentlich keine Zeichenoperationen aus. Stattdessen ruft sie die repaint Methode des Peers auf, womit erstmal eine Anfrage innerhalb des native GUI eingereiht wird. repaint() wird auch als dritter Verarbarbeitungsschritt von einem COMPONENT_RESIZED Ereignis ausgeführt. Der direkte Aufruf von repaint() führt zu keinen weiteren setVisible oder validate Anweisungen.

5. *SystemEventQueue*. Wenn das native GUI es für notwendig hält, reiht es indirekt ein PaintEvent.PAINT oder ein PaintEvent.UPDATE Ereignis in die SystemEventQueue ein. Der Event Dispatcher in EventDispatcherThread nimmt Ereignisse aus dieser Queue heraus und sendet sie schnell weiter. Die PAINT und UPDATE Ereignisse werden besonders behandelt. So wird bei einem UPDATE Event, die update Methode der betroffenen Komponente aufgerufen. Ihr wird ein Graphics Objekt mit dem aktuellen Zustand der betroffenen Region übergeben.

6. Die update Methode löscht typischerweise die Komponente, wobei das Löschen automatisch geclipped wird. Manchmal macht update nichts, sollte beispielsweise die paint Methode den gesamten Bereich beanspruchen.

7. Die paint Methode vollführt das eigentliche Zeichnen. Ihr wird ebenfalls das Graphics Objekt mit dem aktuellen Zustand der clipping reagion übergeben. Die paint Methode beachtet normalerweise den übergebenen Zeichnungsbereich nicht, führt Zeichenanweisung aus und überläßt es dem AWT, Zeichnungen, die aus dem Bereich fallen, zu unterdrücken. Container.update() und Container.paint() veranlassen, daß alle beinhalteten lightweight Komponenten nach dem Löschen ihrer Hintergründe neu gezeichnet werden. Um die heavyweight Komponenten kümmert sich das native GUI selbst. updateAll() und paint() verursacht, daß alle Komponenten des Kontainers gezeichent werden sollen.

3.7.2 Eventbehandlung in Oz

Die BasisAdapter Klasse in Oz verhält sich wie eine normale JavaOz Basisklasse mit einer zusätzlichen Konstruktoraufrufmethode, welche wie üblich eine Nachricht für Java aufsetzt, diese verschickt und auf eine Antwort wartet, die entweder ein korrektes Ergebnis oder eine Exception sein kann. Ist es ein korrektes Ergebnis, wird die Referenz gespeichert und das Objekt in eine Verwaltungstabelle eingetragen. Wird eine Exception zurückgeschickt, wird diese nach Oz umgewandelt und an den Anwender weitergegeben. Das wirklich Interessante an dieser Basisklasse ist der Aufruf von *CheckEvents*, eine Prozedur, die selbst erkennt, an welchen Events der Anwender interessiert ist. Erst müssen jeoch die Zusammenhänge für das Verständnis erklärt werden.

Algorithmus 16 Oz: ActionListener

```
class JOzActionListener from JOzBaseAdapter
    feat events: [actionPerformed]
    meth init()
        JOzBaseAdapter, initialize("JOzActionListener")
    end
end
```

Eine Erweiterung der Basisklasse stellen die einzelnen Listenerklassen, auf die der Anwender zugreifen kann, zur Verfügung. Diese haben einen einfachen Aufbau, wie in Algorithmus 16 zu erkennen ist. Die einzige Methode, die die Listener besitzen ist *init*, die die allgemeine Konstruktormethode ist. Sie sagt der Basisklasse, welcher Listener initialisiert wird. Desweiteren besitzt die Klasse ein Feature *events*, das eine Liste von Atomen ist. Diese Atome repräsentieren Methoden eines typischen Listeners. In dieser Liste müssen alle Methoden des Listeners vertreten sein. Es fällt auf, daß die Klasse *JOzActionListener* keine Ereignismethoden bereitstellt. Dies ist beabsichtigt, da der Anwender bestimmen soll, was bei einem Ereignis passiert. So ist der nächste Schritt, den der Anwender einleitet, eine Klasse zu schreiben, die von einem solchen Listener abgeleitet ist. Diese sollte mindestens eine Methode, deren Name in der Liste *events* enthalten ist, implementieren. Ein korrektes Beispiel hierfür ist Algorithmus 1 auf Seite 12.

Jetzt haben wir folgende Ableitungsleiter: Der Anwender leitet eine Klasse von einer Listenerklasse ab, die wiederum von der Basisadapterklasse abgeleitet ist, die den Aufruf „Events={CheckEvents **self** Message}" beinhaltet. **self** bezieht sich auf das Objekt, welches von der Klasse des Anwenders instanziiert wurde. Über **self** kann *CheckEvents* mit *Class.methodNames* alle Methoden einschließlich der Event-Methoden von der Klasse des Anwenders in Erfahrung bringen. Nun werden alle Namen dieser Methoden mit denen in der Eventliste verglichen. Kommt ein Name darin vor, ist bekannt, für welches Ereignis sich der Anwender interessiert. Diese Namen werden nun in eine Nachrichtform für das Protokoll umgewandelt und an Java verschickt. Das Vorgehen von *CheckEvents* entnimmt man aus Algorithmus 17.

Algorithmus 17 Oz: Events auswerten

```
declare
   JOzAdapters
   [BootObject] = {Module.link ['x-oz://boot/Object']}
   [BootName] = {Module.link ['x-oz://boot/Name']}
in local
   OoMeth = {BootName.newUnique 'ooMeth'}
   proc {CheckEvents Obj Message}
      Meths C
   in
      C = {BootObject.getClass Obj}
      Meths = {Dictionary.keys C.OoMeth}
      {Message insert("EVENTS")}
      {ForAll Obj.events proc {$ Event}
                            if {Member Event Meths} then
                               {Message insert(Event)}
                            end
                         end
      }
   end
      ⋮
```

3.7.3 Eventbehandlung in Java

Wie am Anfang dieses Abschnitts erwähnt, besteht die Lösung der Eventbehandlung zwischen Oz und Java aus einer Mischung von Adapterklassen und Implementierungsklassen. Dies muß man etwas relativieren. Es gibt eine Klasse, die Adapter Basisklasse *JOzBaseAdapter*, von der alle späteren Listener abgeleitet werden. Sie selbst implementiert keine Eventlistener-Schnittstellen, sondern stellt stattdessen die Methode *performEvent* für alle Listener zur Verfügung. Diese macht nichts anderes, als daß sie dem Parser das aufgetretene Event zum speichern gibt und eine Antwort für Oz generieren läßt. Diese Antwort wird über einen Client an den Oz-Server geschickt, der angewiesen wird, auf das bestimmte Event zu reagieren. Desweiteren beinhaltet diese Klasse eine statische Methode *newAdapter*, die im Parser aufgerufen wird und einen gewünschten Proxylistener zurückgibt. Somit sind die Listener versteckt und nicht für die Öffentlichkeit einsehbar.

Die eigentlichen Eventlistener werden als Proxyklassen programmiert, d.h. sie werden nicht wie üblich vom Anwender programmiert, sondern sind schon alle vorhanden und verweisen auf die Ausführung ihrer Methoden nach Oz. Alle Listener sind von der Basisadapterklasse abgeleitet und implementieren gleichzeitig ein Interface. Der Aufbau dieser Klassen sieht wie folgt aus: Jede Klasse besitzt zusätzlich zu ihren Methoden eigene Booleanvariablen, die signalisieren sollen, ob der Anwender sich für ein bestimmtes Ereignis interessiert. Am Anfang sind alle diese Variablen auf **false** gesetzt. Wird eine solche Klasse instanziiert, wird im Konstruktor aus der Nachricht von Oz entnommen, für welche Methoden sich der Anwender interessiert. Die zu diesen Methoden zugehörigen Booleanvariablen werden somit auf **true** gesetzt. Die implementierten Methoden des Interface sind nun so konzipiert, daß ein Ereignis nur

Algorithmus 18 Java: ActionListener

```
class JOzActionListener extends JOzBaseAdapter
                                implements ActionListener {

    private boolean doActionPerformed=false;

    JOzActionListener(StringTokenizer Events, JOzClient Client) {
        super(Client);
        String nextEvent;
        if (Events.nextToken().equals("EVENTS")) {
            while (Events.hasMoreElements()) {
                nextEvent=Events.nextToken();
                if (nextEvent.equals("actionPerformed")) {
                    doActionPerformed=true;
                }
            }
        }
    }

    public void actionPerformed(ActionEvent e) {
        if (doActionPerformed) {
            this.performEvent(e,"actionPerformed",
                    "java.awt.event.ActionEvent");
        }
    }
}
```

dann an Oz weitergegeben wird, wenn die Booleanvariable wahr ist. Auf diese Weise wird der Datenverkehr zwischen Java und Oz reduziert.

Gerade durch diese Proxyprogrammierung geht der Ansatz des generischen Programmierens in diesem Fall verloren. Das muß man in Kauf nehmen, dies ist jedoch nichtt schlimm, da nur wenige Interfaces implementiert werden müssen. Da sich die Arbeit gleicht, kann man mit Tools wie dem *Emacs* 500 Zeilen in einer Stunde schreiben. Das einfachste Beispiel für einen Proxylistener ist in Algorithmus 18 zu sehen.

3.7.4 Behandlung von PaintEvent

Wie oben erwähnt, ist PaintEvent das einzige Ereignis, das nicht über einen Listener abgefangen wird. Das heißt, daß Oz nicht erfährt, wann es eine paint oder update Methode ausführen soll. Eine Abhilfe schafft ein Monitor, der sich an die SystemEventQueue anhängt, und nach PaintEvents horcht. Ist ein solches Ereignis vorhanden, wird ein Aufruf der paint oder update Methode in Oz in Gang gesetzt, wobei das abgefangene Event danach weiter zur source component geschickt wird. Dies ist insbesondere dann sinnvoll, wenn Standardkomponenten erneuert werden müssen. Das Interface, oder vielmehr Oz kann dies nicht mehr selbst übernehmen. Es würde auch einen unnötigen Mehraufwand bedeuten.

Der Monitor alleine hilft nicht weiter, denn, obwohl wir einen Schritt weiter gekom-

men sind, entstehen neue Probleme. Diese sind auf den typischen Event Mechanismus von Java zurückzuführen, wie schon im Kapitel über die Grundlagen dargestellt wurde.

Dort wurde beschrieben, daß lightweight components kein PaintEvent in die SystemEventQueue setzen, stattdessen alarmieren sie sich selbst, also am Monitor vorbei. Die Alarmierung erzeugt keinen Fehler, da die Pseudo Komponente in Java in der Hashtabelle vorhanden ist; nur wird hier die leere paint bzw. update Methode dieser Pseudo Komponente aufgerufen. Da diese keine Proxyklassen sind, werden keinerlei Aufrufe nach Oz verschickt. Dieses Problem ist demnach eine Sackgasse und somit unlösbar.

Weiterhin wird ein PaintEvent meist an die Topkomponente, oder an den Topkontainer geschickt, der dann, falls notwendig, selbständig die paint oder update Methode seiner Subkomponenten aufruft. Wiederum sind diese Komponenten Pseudo Komponenten, das Problem zeigt sich als das gleiche wie oben. Eine kleine Lösung besteht darin, daß der Anwender selbst in der paint Methode seines Topkontainers in Oz seine Subkomponenten aufruft.

Ein gravierendes Problem, welches das freie Zeichnen in Swing beeinträchtigt ist eines, daß noch wenig dokumentiert ist, und auf welches ich per Zufall gestoßen bin. Möchte man von einer Swing Komponente eine Graphics Struktur haben, so wird ein *com.sun.java.swing.SwingGraphics* geliefert, eine Klasse, die java.awt.Graphics erweitert, die jedoch weder protected, noch private, noch public ist. Diese Klasse ist irgendwo intern in Swing vorhanden. Dort wird sie instanziiert und an die Außenwelt gegeben. Aufgrund fehlender Rechte ist es nicht möglich, über die Reflection API auf die Methoden dieser Instanz zuzugreifen. Damit ist es nicht möglich, irgendwelche Zeichenmethoden sowohl von SwingGraphics als auch vom normalen Graphics zu nutzen.

Man kann zwar zum reinen Zeichnen statt eines JPanel oder einer JComponent ein Canvas benutzen, jedoch entstehen Inkonsistenzen, sobald AWT und Swing Komponenten gleichzeitig benutzt werden. So zeigt sich zum Beispiel ein Swing Menu nicht über einem Canvas, sondern es wird hinter das Canvas gezeichnet, was sinnlos ist. Als ich diese Problem in der Newsgroup comp.lang.java.gui gestellt habe, kam die Erklärung, daß lightweight componenten wie eigenständige Fenster behandelt werden, die sofort in den Vordergrund gelegt werden. Diese Antwort nehme ich jedoch mit Vorbehalt an, da dieses Problem auch für Swing existieren sollte, praktisch jedoch nicht vorhanden ist. Letztendlich habe ich keine sinnvolle Erklärung für dieses Verhalten, als daß es an der Mixtur von Swing und AWT liegt.

Drei gravierende Probleme, die alle mit paint oder update zu tun haben. Kann man auf diese Methoden nicht verzichten, so empfehle ich dringend, ein eigenes Modul zu schreiben und dieses ins System einzuklinken. Will man nur Standardkomponenten benutzen, dann reicht das hier vorgestellte Interface genügend aus.

3.7.5 Fazit des Event Handlings

Obwohl ich schon mehrfach auf die Nachteile des Event Handling in bezug auf PaintEvent hingewiesen habe, möchte ich noch einmal alle Punkte zusammenstellen, die

von der grafischen Nutzung abraten:

- AWT und Swing sind langsamer als Bibliotheken auf Basis von C/C++. Selbst ein kommerzielles Produkt wie *NetBeans*, ein zu 100% in Java geschriebener FrontEnd-Generator für AWT und Swing, bietete auf einem Doppel-P300-Prozessor eine erschreckend langsame Performance. Auf meinem HomePC, einem CyrixP150 mit 80MB RAM brauchte er Sekunden für den Aufbau eines Menus.
- Der Java Server horcht immer in bestimmten Intervallen, ob eine Nachricht vorliegt. Somit ist immer eine Standardverzögerung bei jeder Anfrage vorhanden.
- Es gibt in Java einen riesigen Overflow bei der Verarbeitung von Anfragen:
  - Nachricht extrahieren
  - Objekte aus Hashtabelle heraussuchen
  - Klassen mittels Reflection API suchen
  - Methoden mittels Reflection API suchen
  - Parameter generieren (Klassen finden und Instanziieren)
  - Felder mit Reflection API suchen
  - Klasse Instanziieren oder Methode ausführen oder auf ein Feld zugreifen
  - Antwort generieren
  - Antwort verschicken
- Zusätzlich existiert ein Overflow in Oz:
  - Klasse generieren
  - Anfrage generieren
  - Anfrage verschicken
  - Antwort empfangen
  - Antwort auswerten und verwalten
- Zugriff auf SwingGraphics wird mit einer IllegalAccessException verweigert
- PaintEvent erfährt eine Sonderbehandlung durch einen Monitor
- PaintEvent nur an TopKontainer
- PaintEvent von heavyweight componenten nicht abfangbar
- Mit der Zeit starke Belastung des Speichers, da jedes generierte Objekt, welches Oz mitgeteilt wird, in einer Hashtabelle dauerhaft gespeichert ist

3.8 Lazy Packages

In früheren Entwicklungsstadien wurden die Klassen sofort beim Starten von Oz generiert. Dies lag daran, daß das Interface eines Moduls ein Record ist, dessen Features mit Werten belegt werden müssen. Diese Werte sind die Klassen, die zur Laufzeit dementsprechend bekannt sein müssen.

Algorithmus 19 Lazy-Funktionalität

```
declare
   Awt
in
local
   fun {MakeMaker Prefix}
      fun lazy {$ Suffix}
         {JavaClass.newClass {VirtualString.toString Prefix#Suffix}}
      end
   end
   AWTM = {MakeMaker 'java.awt.'}
in
   Awt={Record.map
      awt(abstractAction: 'AbstractAction'
          aWTEvent: 'AWTEvent'
          aWTEventMulticaster: 'AWTEventMulticaster'
          borderLayout: 'BorderLayout'
          ⋮
          window: 'Window'
      )
   AWTM}
end
```

Besser wäre es, wenn die Klassen erst dann generiert werden, wenn sie wirklich gebraucht werden, d.h. erst dann, wenn ein Objekt instanziiert wird. Dies ermöglich die Lazy-Funktionalität in Oz, welche wie folgt funktioniert (vergleiche Algorithmus 19): Der Aufruf von *MakeMaker* gibt eine Funktion zurück, die mit **lazy** markiert ist. Diese Funktion soll eine Klasse zum angegebenen Argument generieren. Der Aufruf von *MakeMaker* geschieht im Interface selbst, d.h. der Wert eines Features ist die oben angegebene Lazy-Funktion. Dadurch daß die Funktion mit **lazy** versehen ist, wird sie nicht sofort ausgeführt. Stattdessen gibt sie ein „Versprechen", erst bei Anwendung eines Features des Interface zu evaluieren. Dies ist der Fall, wenn mit einer Instanziierung auf das Feature zugegriffen wird. Erst dann wird die Lazy-Funktion ausgeführt, und eine Klasse generiert, die im Anschluß instanziiert wird. In Algorithmus 19 wird *MakeMaker* auf alle Features des Interface bequem mit *Record.map* angewendet. Die Lazy-Funktionalität wird bei folgenden Modulen angewendet:

- AWT

- Swing

Der Aufbau von Algorithmus 19 kann dazu benutzt werden, eigene Module zu schreiben, die eine Schnittstelle zu eigenen Java Klassen bilden. Dies ist insbesondere dann effizient, wenn komplexe Klassen anstehen, die intensiv von lightweight components Gebrauch machen. So werden grafische Anweisungen praktisch ohne Verzögerung gegenüber Java ausgeführt. Somit kann eine Arbeitsteilung aufgestellt werden, in der Oz die Kontrolle über alles einnimmt, während Java die grafisch ausführende Komponente im System bleibt.

3.9 Fazit

In diesem letzten Abschnitt möchte ich von meinen Erfahrungen mit diesem Projekt berichten. Insbesondere will ich erläutern, was ich gelernt habe, und darauf aufbauend, welche Verbesserungen ich noch für dieses Projekt plane.

Ich habe beide Sprachen, Java und Oz, im Wintersemester 97/98 kennengelernt. Das Praktikum gab mir die Chance, meine Kenntnisse zu vertiefen. In Java habe ich mich viel mit den unterschiedlichen Paketen, die es für diese Sprache gibt, beschäftigt. Dies war am Anfang notwendig, um die ideale Implementierungsform zu finden. Die Pakete *Net* und *IO* geben Auskunft über allgemeine Netzprogrammierung. Ein Blick in die Pakete *RMI* und *CORBA* verschafften mir Kenntnisse über verteilte Programmierung. Das *Java Native Interface* ist eine Schnittstelle zu C/C++, über die ein Weg nach Oz geschlagen werden könnte. Doch letztendlich wurde die *Reflection API* zum Zentrum meiner Aktivitäten. Diese hatte das oberste Ziel, transparent auf Klassen von AWT und Swing zugreifen zu können. Im Zuge dieser Aufgabe habe ich mich also über den Aufbau und die Konzepte von AWT und Swing informiert. Nachdem erste gravierende Probleme mit dem Event Handling aufgetreten sind, habe ich mich intensiv mit der internen Bearbeitung auseinandergesetzt. Da es hierzu keine offizielle Literatur gibt, war ich auf Informationen aus dem Newsnet angewiesen. Viele Leute, die sich mit ähnlichen Problemen auseinandergesetzt haben, haben viele Antworten auf meine Fragen geschrieben.

Einen Großteil meiner Arbeit fand aber auch in Oz statt. Dort habe ich gelernt, mit einer Sprache, die sich erst im Entwicklungsstadium befindet, Software zu entwickeln. Eine große Schwierigkeit bestand darin, mit undokumentierten Modulen umzugehen. So hatte ich große Probleme, Klassen generisch zu erzeugen. Dies lag nicht nur an der Dokumentation, sondern auch an der Stabilität und an dem sich schnell wandelnden System. Ich habe lange Zeit das Projekt „blind" implementiert. Hält man sich jedoch an die Vorgaben, geht das erstaunlich gut. Weitere Probleme bestanden darin, diese etwas eigenwillige Sprache zu begreifen, und deren syntaktischen Eigenwilligkeiten, man kann es auch „Zucker" nennen, effizient einzusetzen. Ein sehr schöner Gesichtspunkt von Oz ist die Kombination von objektorientierten, funktionalen und logischen Paradigmen innerhalb einer Sprache. Besonders das Constraint Konstrukt erleichtert oft die Programmierung. Allerdings sind mir in der Anwendung Mängel bezüglich

der Methodenkonstruktion in Oz aufgefallen. So ist es nicht möglich, eine Methode wie eine Funktion zu behandeln. Damit kann man keine *nested applications* benutzen, d.h., daß das folgende Beispiel nicht möglich ist:

{Prozedur {Objekt tuWas(Parameter)}}

Das Erlernen einer Programmiersprache ist eine Sache. Sehr viel bedeutender ist die abstrakte Planung eines großen Projektes. So habe ich am Anfang aufgrund meiner Unkenntnis den Fehler gemacht, sofort ,,loszuhacken". Später mußten viele Korrekturen gemacht werden. Die endgültige Erkenntnis kam mit der Erstellung der Dokumentation. So hab ich begriffen, daß eine Grammatik für ein Protokoll sehr hilfreich sein kann. Das aktuelle Protokoll ist aus einer anfangs relativ ungeordneten Syntax entstanden. Insgesamt ist das Design eines Projektes sehr wichtig, und sollte immer am Anfang der Arbeit stehen.

Mehr Übersicht über ein Projekt beschafft man sich, indem man, ähnlich dem Schichten-Modell für Netzkommunikation, Implementationsschichten plant. Jede Schicht sollte eine Schnittstelle zur nächsthöheren Schicht anbieten. Dies verschafft mehr Überblick und einen guten Leitfaden für eine Implementierung.

Doch auch auf anderen Gebieten habe ich während der letzten zwölf Monate Erfahrungen gesammelt. So hab ich den Umgang mit Emacs und LaTeX näher kennengelernt. Bei LaTeX war das Einbinden von weiteren Paketen interessant. Im Zusammenhang mit Emacs kam somit das Paket *AucTeX* zum Einsatz.

Zusammenfassend kann ich sagen, daß ich ein größeres Verständnis für Konzepte von Programmiersprachen und Pakete erworben habe. Wenn ich auf neue Sachen stoße, dann denke ich oft intuitiv und assoziierend mit meinen Erfahrungen in die richtige Richtung.

Wie schon öfter in der Dokumentation erwähnt, beherbergt das System noch einige Fehler bzw. Designungeschicklichkeiten. Neben diesen notwendigen Verbesserungen stelle ich mir für die Zukunft noch einige neue Implementationskorrekturen bzw. -ergänzungen vor.

Insgesamt muß der Source im Hinblick auf Lesbarkeit und Effizienz überarbeitet werden. Hier will ich über den Einsatz von ,,Functoren" nachdenken.

Eine leichte Verbesserung wäre für das Client-Server-Modell und dem zugehörigen Protokoll günstig. Zur Zeit werden hier sechs Sockets gestartet. Eine Minimierung auf vier ist wünschenswert.

Auf der unteren Ebene sollen weitere Typen ermöglicht werden. Insbesondere sollen Arrays eine Verwendung finden.

Sind alle Typen, insbesondere Arrays, eingebaut, kann man die Argumente bei Methodenaufrufen auf Nebenläufigkeit, d.h. auf ihre Zustandsänderung, überprüfen. Dies muß der anderen Seite mitgeteilt werden, die zusätzlich zum Ergebnis des Methodenaufrufs, die durch die Nebenläufigkeit hervorgetretenen Zustandsänderungen der Argumente mitteilt.

Die Möglichkeiten von Swing noch nicht hinreichend ausgeschöpft, so daß ich noch einige weitere Swing Pakete zur Verfügung stellen möchte. Im Zusammenhang mit AWT und Swing muß das Event-Modell verbessert werden. Zur Zeit werden alle Events, die gefragt sind abgespeichert und nach ihrer Auswertung nicht mehr freigegeben.

Ein weiteres Thema, das ich bisher nicht erwähnt habe, obwohl schon erste Ansätze vorhanden sind, ist der Miteinbezug von weiteren Java Interfaces (keine Listener). Oz behandelt Java Interfaces wie eigene Klassen, obwohl es weiß, daß es eigentlich abstrakte Schnittstellen sind. Dadurch daß sie wie Klassen behandelt werden und erst bei Benutzung zur Laufzeit generisch erzeugt werden, war es notwendig, die Methoden lediglich mit dem Befehl *skip* zu füllen. Jetzt muß man Java noch erklären, daß bei Nutzung eines implementierten Interfaces, die Abarbeitung in Oz stattfinden soll. Dazu ist ein Interface von Java nach Oz notwendig. Dieses soll schließlich das sein, das auch der Endanwender benutzen wird.

Das aktuelles Design von Oz sieht vor, daß die Klassenspezifikation nur *single inheritance* erlaubt. In Java ist es für die Interfaces jedoch erlaubt, mittels *multiple Inheritance* zu vereben. Es stellt sich heraus, daß keines der Interfaces in Awt und Swing *multiple Inheritance* benutzt. Somit kann dieses Feature vorerst vernachlässigt werden.

Ein schwieriges Problem in bezug auf Interfaces liegt darin, daß in Java Proxy-Interfaces geschrieben werden müßten, die eine statische Methode *handle* aufrufen, welche die Anfrage weiter nach Oz leitet. Diese statische Methode muß natürlich variabel sein, da jede I.-Methode andere Argumente enthält. Außerdem darf der Rückgabewert nicht vergessen werden, falls einer gebraucht wird. *handle* wiederum ist nur ein Verbindungsstück zum eigentlichen Kommunikationsinterface mit Oz.

Es stellt sich die Frage, ob solch eine Implementierung notwendig ist. Notwendig ist es in dem Fall, in dem eine Methode ein Argument verlangt, welches ein bestimmtes Interface implementiert hat.

Beispiel: Angenommen es existiert eine Klasse *Brief* mit Methode *schreibe*, die als Parameter ein Objekt erwartet, welches ein Interface implementiert, das eine für sich bestimmte Formatierungsfunktion beinhaltet. Diese kann z.B. eine Methode *format* sein, die den Brief entsprechend auf ein bestimmtes Papierformat vorbereitet. Nun kann man über das Objekt diese Methode aufrufen, und sie wird für jede Implementierung anders sein. Siehe *Tests/Buch.java*.

Die einzige mir bisher bekannte Anwendung, bei der es notwendig ist, ist die Methode *setBorder* in *JComponent*. Ein entsprechendes Objekt kann man durch die *BorderFactory* kreieren lassen. Gibt noch keine Methode, die solch ein Objekt generiert, muß ich doch noch dieses Feature einfügen.

Was in die eine Richtung geht, muß auch in die andere Richtung funktionieren. Hier ist ein geeignetes Interface von Java nach Oz gefragt. Bisher existiert es in einem rudimentären Zustand. Hier besteht die Aufgabe, die Schnittstelle zu verallgemeinern, so daß Ereignisse darauf aufbauen können. Insbesondere soll der Anwender

die Möglichkeit erhalten, transparent auf Oz zugreifen zu können. Außerdem sollen, wie schon in der Aufgabenstelleung beschrieben, Applets, die Berechnungen in Oz ausführen, gestartet werden können.

Letzten Endes müssen noch sehr viele Tests gemacht und Beispiele geschrieben werden.

Ich habe sehr gerne an diesem Projekt gearbeitet, und sollte das Interesse seitens des Lehrstuhls daran bestehen bleiben, würde ich daran gerne weiterentwickeln, da ich die Vorzüge, insbesondere von Java, zu schätzen gelernt habe.

Literaturverzeichnis

[1] comp.lang.java.gui.

[2] comp.lang.java.programmer.

[3] comp.lang.java.machine.

[4] Roedy Green. Java glossary. http://mindprod.com.

[5] Martin Henz. *Objets in Oz*. PhD thesis, Universität des Saarlandes, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, May 1997. http://www.ps.uni-sb.de/ns3/oz2/documentation/.

[6] Martin Henz, Martin Müller, Christian Schulte, and Jörg Würtz. *The Oz standard Modules. DFKI Oz documentation series*. German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, first edition, December 1997. http://www.ps.uni-sb.de/ns3/oz2/documentation/.

[7] *Java Tutorial*. JavaSoft, 2550 Garcia Avenue, Mountain View, CA 94043 U.S.A, 408-343-1400, second edition. http://java.sun.com/docs/books/tutorial/.

[8] *Java Core Reflection: API and Specification*. JavaSoft, 2550 Garcia Avenue, Mountain View, CA 94043 U.S.A, 408-343-1400, January 1997. http://java.sun.com/products/jdk/1.1/download-pdf-ps.html.

[9] *Java Native Interface: Specification*. JavaSoft, 2550 Garcia Avenue, Mountain View, CA 94043 U.S.A, 408-343-1400, May 1997. http://java.sun.com/products/jdk/1.1/download-pdf-ps.html.

[10] Christian Schulte. *Open Programming in DFKI Oz. DFKI Oz documentation series*. German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, first edition, December 1997. http://www.ps.uni-sb.de/ns3/oz2/documentation/.

[11] Christian Schulte and Michael Mehl. *Window Programming in DFKI Oz. DFKI Oz documentation series*. German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, first edition, December 1997. http://www.ps.uni-sb.de/ns3/oz2/documentation/.

[12] Christian Schulte, Michael Mehl, Tobias Müller, Konstantin Popov, and Ralf Scheidhauer. *Window Programming in DFKI Oz. DFKI Oz documentation series*. German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, first edition, December 1997. http://www.ps.uni-sb.de/ns3/oz2/documentation/.

[13] *Java Object Serialization: Specification*. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043-1100 U.S.A,, October 1997. http://java.sun.com/products/jdk/1.1/download-pdf-ps.html.

[14] *Java Remote Method Invocation: Specification*. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043-1100 U.S.A,, February 1997. http://java.sun.com/products/jdk/1.1/download-pdf-ps.html.

[15] *Java RMI Tutorial*. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043-1100 U.S.A,, February 1997. http://java.sun.com/products/jdk/1.1/download-pdf-ps.html.