Transparente Objektverteilung in Java

Studienarbeit

Matthias Zenger

Februar 1997

Inhaltsverzeichnis

1 Einleitung

Java wurde als Programmiersprache für das Internet konzipiert. Seine plattformunabhängige Architektur erlaubt es, völlig neue Arten von Anwendungen zu entwickeln. Programme in Form von sogenannten Applets können beispielsweise über das Netz übertragen und lokal ausgeführt werden. Da Java als Programmiersprache für das Internet, durch seine Klassenbibliothek selbst auch eine große Anzahl von Netzprotokollen unterstützt, scheint es als Grundlage für die Entwicklung verteilter Anwendungen geradezu prädestiniert zu sein. So ist es ohne größeren Aufwand möglich, verteilte Clients und Server zu realisieren, die über Sockets kommunizieren. Eine Kommunikation über Sockets ist zwar sehr flexibel und ausreichend für einen allgemeinen Datenaustausch, allerdings verlangt diese vom Programmierer auch, auf Programmebene Protokolle zu entwickeln, die festlegen, wie die zu übermittelnden Daten zur Übertragung codiert werden.

Eine Alternative zur direkten Socket-Kommunikation stellt ein *Remote Procedure Call* (kurz *RPC*) dar. Bei diesem muss sich der Programmierer keine Gedanken über die Repräsentation der gesendeten und empfangenen Daten machen. Prozeduraufrufe, die der Kommunikation mit einer entfernten Instanz dienen, sehen aus der Sicht des Programmierers aus, wie lokale Prozeduraufrufe. Das zugrundeliegende System sorgt für die Verpackung und Übertragung der Argumente und des Resultats, ohne dass der Benutzer mit diesen technischen Details konfrontiert wird. Allerdings passt das Konzept des entfernten Prozeduraufrufs nicht sonderlich gut zu verteilten objektorientierten Systemen, bei denen Objekte aus verschiedenen Adressräumen miteinander kommunizieren müssen.

Ein zum RPC äquivalentes Konzept, das auf die Bedürfnisse verteilter objektorientierter System zugeschnitten ist, stellt der entfernte Methodenaufruf (*remote method invocation*, kurz *RMI*) dar. Ab Java-Version 1.1 wird ein entsprechendes RMI-Packet, das speziell für den Einsatz in Java konzipiert wurde, offiziell zur Java Klassenbibliothek gehören [RMI_96]. Damit ist es dann möglich, über Methodenaufrufe mit Objekten aus Adressräumen anderer virtueller Maschinen zu kommunizieren. Im Gegensatz zu anderen RMI-Systemen, wie beispielsweise *CORBA* [OMG_95], die von einer heterogenen, programmiersprachenunabhängigen Umgebung ausgehen, basiert das RMI-System von Java auf der homogenen Umgebung der *Java Virtual Machine*. Damit ist zwar einerseits nur ein gegenseitiger Methodenaufruf zwischen Java-Objekten möglich, bei der Entwicklung des System konnten jedoch andererseits auch speziell die Eigenschaften des Java Objektmodells, wo immer dies nur möglich war, ausgenutzt werden. Nur so wurde eine Integration des verteilten Java Objektmodells in die Sprache, unter größtmöglicher Beibehaltung der Java Objekt-Semantik, möglich.
Obwohl eines der Ziele bei der Entwicklung des Java RMI darin bestand, technische Aspekte des Systems, wie z.B. der des verteilten Garbage Collectors, für den Programmierer möglichst transparent zu gestalten, muss der Benutzer, vor allem bei der Implementierung von Servern, dennoch viele Details des Java RMI-Systems kennen.

Ziel dieser Arbeit war es, von diesen Details soweit zu abstrahieren, dass für den Programmierer kein Unterschied mehr zwischen der Implementierung einer lokalen Klasse, auf deren Objekte von einer entfernten Java Virtual Machine nicht zugegriffen werden kann, und einer Klasse, deren Objekte sich in einer verteilten Umgebung befinden können, zu erkennen ist.

Um einen solchen transparenten Umgang mit verteilten Objekten realisieren zu können, mussten prinzipiell auf zwei Ebenen Erweiterungen an der Java Umgebung vorgenommen werden. Zum einen musste eine verteilte Laufzeitumgebung geschrieben werden, die dynamisch für die

Objektverteilung und für administrative Aufgaben zuständig ist. Zum anderen war es notwendig, einen Übersetzer zu entwickeln, der herkömmlichen Java Quellcode so übersetzt, dass eine verteilte Ausführung in der Laufzeitumgebung möglich wird. Die Funktionalität des RMI reicht als Basis für die Erfüllung dieser Aufgaben bei weitem nicht aus. Es werden grob folgende Eigenschaften zusätzlich benötigt:

- Eine, zum lokalen Objektmodell möglichst äquivalente verteilte Semantik für Klassen und Objekte.
- Zusammenarbeit lokaler und entfernt liegender Objekte innerhalb der Laufzeitumgebung.
- Realisierung des vom RMI nicht unterstützten entfernten statischen Methodenaufrufs.
- Zugriff auf statische und nicht-statische Variablen entfernt liegender Objekte (ebenfalls von RMI nicht unterstützt).
- Ermöglichung von Objekt-Migrationen, um Objekte als mobile Agenten einsetzen zu können oder um später Lokalitätsoptimierungen vornehmen zu können.
- Generierung und Zugriff auf entfernt ablaufende Threads, um echte Nebenläufigkeit in der verteilten Umgebung nutzbar zu machen.

In den Abschnitten 4 und 5 wird die Realisierung dieser Aufgaben vorgestellt. Zuvor wird jedoch das aus dem Java RMI resultierende verteilte Objektmodell beschrieben, auf dem das entwickelte System basiert. Dabei wird auch untersucht, wie Unterschiede zum lokalen Objektmodell durch geeignete Transformationen lokaler Java Klassendefinitionen möglichst verdeckt werden können und an welchen Stellen dies nicht möglich ist.

2 Verteiltes Objektmodell

2.1 Java Remote Method Invocation

Im verteilten Objektmodell von Java's RMI ist ein *remote object* ein Objekt, das Methoden enthält, die von einer anderen Java Virtual Machine aus aufgerufen werden können. Für jedes *remote object* müssen die entfernt aufrufbaren Methoden durch ein oder mehrere Interfaces definiert werden. Um mit entfernten Objekten überhaupt umgehen zu können, müssen lokale Stellvertreter die Rolle der eigentlichen Objekte übernehmen. Diese Stellvertreterobjekte, genannt *stubs* oder *proxies*, sehen bezüglich der entfernt aufrufbaren Methoden genauso aus, wie die entsprechenden *remote objects*; d.h. sie implementieren genau die gleichen Interfaces, die die entfernte Schnittstelle des Objekts beschreiben. Diese Interfaces werden im folgenden stets als *remote interfaces* bezeichnet. Die Stubs übernehmen den Transfer der Argumente und die Initiierung des eigentlichen Methodenaufrufs auf der virtuellen Maschine, auf der sich das zugehörige *remote object* befindet. Dazu sprechen sie ein weiteres Stellvertreterobjekt, das sogenannte *skeleton* auf der Seite des *remote objects* an. Das Skeleton empfängt die Paramter, die vom Stub gesendet werden und ruft schließlich lokal die korrespondierende Methode auf. Das Resultat des Methodenaufrufs wird vom Skeleton dann wieder an den Stub zurückgeschickt.
Das Skeleton wird also auf der Seite des *remote objects* vom RMI-System dazu verwendet, einen an das *remote object* gerichteten entfernten Methodenaufruf auf der Heimatmaschine in einen lokalen Methodenaufruf bezüglich des ansässigen *remote objects* umzusetzen. Damit es möglich ist, dass nebenläufig mehrere Methodenaufrufe an ein *remote object* gerichtet werden können, werden die vom Skeleton initiierten lokalen Aufrufe jeweils in eigenen Threads ausgeführt.[1] Die Auftrennung der Funktionalität in den Verwaltungsteil im Skeleton und den benutzerdefinierten Teil im eigentlichen *remote object*, erlaubt es, ein entferntes Objekt ähnlich wie ein lokales Objekt zu implementieren und zu übersetzen und gleichzeitig die RMI-spezifischen Anweisungen automatisch von einem Stub-/Skeleton-Übersetzer zu generieren.

Zum Übermitteln der Argumente und des Resultats bei einem entfernten Methodenaufruf verwendet das RMI-System die Technik der *Object Serialization* [Ser_96]. Das *Object Serialization*-Paket für Java erlaubt es, ein als serialisierbar gekennzeichnetes Objekt, in einen Stream zu schreiben und daraus wieder zu lesen. Damit ist es möglich, ein Objekt in einer Bytefolge zu codieren und es auf einer beliebigen anderen virtuellen Maschine wieder aus der Bytefolge zu rekonstruieren. Effektiv wird hiermit eine Kopie des ursprünglichen Objekts auf einer anderen Maschine angelegt. Aus Sicherheitsgründen ist dies natürlich nicht mit allen Objekten möglich. Nur Objekte einer Klasse, die das `java.io.Serializable`-Interface implementieren, können „serialisiert“ werden.

Da nun Stub und *remote object* bezüglich der entfernt aufrufbaren Methoden den gleichen Typ besitzen, gestaltet sich ein RMI-Aufruf für den Programmierer völlig transparent. Unabhängig ob das Objekt lokal oder entfernt liegt, ist die Syntax für einen Methodeaufruf völlig identisch. Außerdem ist die Verwendung des `instanceof`-Operators genauso wie eine Typkonversion bezüglich genau der gleichen *remote interfaces* möglich, wie dies beim *remote object* der Fall ist. Neben dieser syntaktisch transparenten Verwendung des Java RMI-Systems, unterscheidet sich das verteilte Objektmodell vom lokalen Modell jedoch in folgenden Punkten [RMI_96]:

[1] Stammen zwei entfernte Methodenaufrufe von der gleichen Java Virtual Machine, so behält sich das Java RMI-System vor, diese nacheinander in einem einzigen Thread auf der Skeleton-Seite auszuführen [RMI_96].

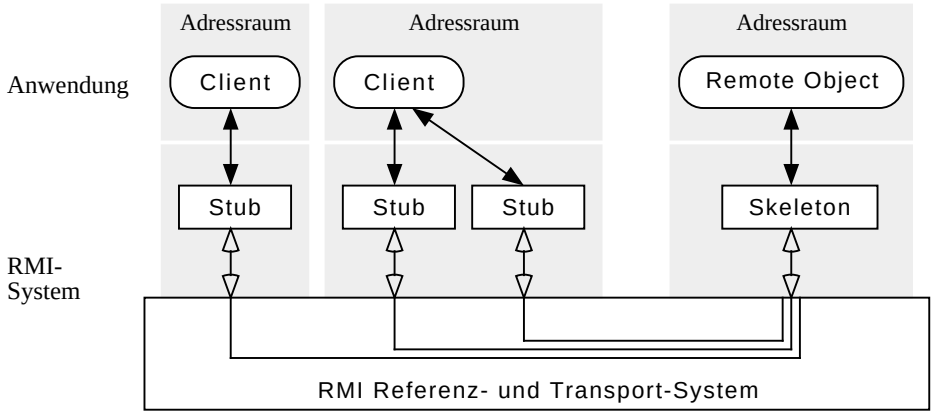


Abbildung 1: Prinzip des Java RMI-Systems

- Die Interaktion mit *remote objects* geschieht immer über deren *remote interfaces* und niemals über die Implementierung dieser Interfaces; d.h. *remote objects* werden in einem Programm stets über ihre *remote interface*-Typen deklariert.
- Werden bei einem entfernten Methodenaufruf lokale Objekte, also keine *remote objects* als Argumente übergeben oder als Ergebnis zurückgeliefert, so geschieht dies per Kopie und nicht, wie sonst in Java üblich, als Referenzübergabe. Eine Referenzübergabe würde in diesem Kontext keinen Sinn ergeben, da eine Referenz auf ein lokales Objekt in einer anderen JVM nicht verwendbar wäre. Da RMI zur Parameterübergabe und Resultatsrückgabe Object Serialization verwendet, sind nur serialisierbare Objekte als Argumente für einen entfernten Methodenaufruf zulässig.
- *Remote objects* werden stets per Referenz übergeben.
- Einige, der in der Klasse `java.lang.Object` definierten Methoden, sind für *remote objects* ungeeignet und müssen extra angepasst werden.
- Da bei einem entfernten Methodenaufruf Fehler bei der Parameterübergabe oder bei der Lokalisierung des entfernten Objekts bzw. dessen Skeletons auftreten können, müssen entsprechende, möglicherweise auftretende Exceptions zusätzlich bei einem RMI-Aufruf abgefangen werden.

Die Einschränkung, dass *remote objects* stets über einen Interface-Typ angesprochen werden, hat noch weitere bedeutende Konsequenzen zur Folge, vor allem im Hinblick auf das Ziel, lokale Klassendefinitionen, für die Verwendung im verteilten Objektmodell, aufzubereiten:

- Es ist für ein *remote object* nicht möglich, statische Methoden zu definieren, die von einer anderen JVM aus aufgerufen werden können.
- Da jedes Feld eines Interfaces implizit den Modifikator `public` erhält, sind automatisch alle entfernt aufrufbaren Methoden eines Objekts `public` deklariert. Andere Modifikatoren sind nicht zulässig [GJS_96].
- Auf statische und nicht-statische Variablen eines *remote objects* kann von außen nicht zugegriffen werden.

Jedes Java Objekt ist mit einem *lock*, das die Realisierung eines gegenseitigen Ausschlusses beim Zugriff auf das Objekt ermöglicht, und einem *wait set* assoziiert, welches die Menge von Threads angibt, die gerade auf eine Zuteilung der Zugriffsrechte warten. Durch den indirekten Zugriff auf *remote objects* über Stubs und Skeletons, ergeben sich deswegen große Unterschiede beim Umgang mit *remote objects* auch im Hinblick auf Threads und die Objekt-Synchronisation:

- `synchronized`-Anweisungen der Form `synchronized (remoteObj) {...}` haben nicht den gewünschten Effekt zur Folge, da hier lediglich der Zugriff auf den Stub `remoteObj` geregelt wird. Da aber für ein *remote object* im Allgemeinen beliebig viele Stubs existieren können, hat diese Synchronisation keine Auswirkungen auf das entfernte Objekt selbst.

- Die Methoden `wait`, `notify` und `notifyAll` der Klasse `java.lang.Object` funktionieren angewendet auf *remote objects* nicht wie erwartet. Dies liegt zum einen daran, dass bei entferntem Zugriff wieder auf das *wait set* des Stubs statt des entfernten Objekts Bezug genommen wird. Zum anderen arbeiten die Methoden, auch wenn sie innerhalb des *remote objects* selbst aufgerufen werden, nicht korrekt, da bei einem entfernten Methodenaufruf die Methode auf der Skeleton-Seite in einem, vom RMI-System instanziierten Thread, und nicht im aufrufenden Thread ausgeführt wird.

Da es mangels Alternativen nötig war, eine transparente Übersetzung lokaler Klassen in Klassen, die in einem verteilten System interagieren können, auf dem Java RMI aufzubauen, musste bei der Tranformation natürlich versucht werden, möglichst viele dieser Unterschiede zum lokalen Objektmodell zu verdecken. Wie man im folgenden sehen wird, gelingt dies in einigen Fällen, in andern Fällen sind die Unterschiede nicht zu umgehen. Die Gründe hierfür sind zum einen technischer Art, zum anderen liegen sie in der Natur der Sache: mit Objekten in einer verteilten Umgebung muss eben in vielen Situationen anders umgegangen werden als mit Objekten, die sich in einem einzigen Adressraum befinden. Die Hauptunterschiede liegen hier in der Berücksichtigung von Latenzzeiten, unterschiedlichen Speicherzugriffsmodellen, dem Umgang mit der Nebenläufigkeit sowie Fehlern in Teilkomponenten, wie z.B. Netzwerkfehlern oder der eingeschränkten Verfügbarkeit von entfernten Objekten.
Das in dieser Arbeit vorgestellte System verdeckt zwar vor dem Benutzer alle diese Aspekte, jedoch geschieht dies oftmals zu Lasten der Robustheit und Zuverlässigkeit des Systems. Eine tiefergehende Diskussion dieser Problematik ist in [W^{3}K_94] zu finden.

2.2 Begriffsklärung

Um die Bedeutung einiger, im folgenden verwendeter Begriffe, in Zusammenhang mit Java und dem, in dem weiteren Verlauf vorgestellten System zu verdeutlichen und zu präzisieren, folgen einige Begriffsbeschreibungen. Viele der Details werden erst in den einzelnen Abschnitten des nächsten Kapitels motiviert und dienen vorab nur zur Beschreibung der angestrebten verteilten Objekt-Semantik.

Verteilte Umgebung: Eine verteilte Laufzeitumgebung besteht aus einer Menge von lokalen Java Virtual Machines (*lokale JVM*) mit jeweils eigenem Adressraum, die auf einem oder mehreren Rechnern verteilt laufen. Die Kommunikation zwischen Objekten auf verschiedenen Maschinen erfolgt über RMI. Sieht man eine verteilte Umgebung als Einheit an, so stellt diese praktisch eine verteilte virtuelle Maschine dar.

Lokale JVM: Virtuelle Maschine, auf der ein spezieller Distributed-Java-Daemon läuft. Dieser Thread, eigentlich ein *remote object*, dient als Ansprechpartner für die Kommunikation mit dieser Maschine.

Remote Class: Menge von Java Klassen, die zusammen das Verhalten einer lokalen Klassendefinition für die Verwendung in einer verteilten Umgebung möglichst korrespondierend nachbilden. Im Gegensatz zu einer lokalen Java Klasse, die möglicherweise auf mehreren Java Virtual Machines der Umgebung unabhängig voneinander geladen werden, wird eine *remote class* nur einmal innerhalb eines verteilten Systems initialisiert und kann auch nicht mehr auf eine andere Maschine umziehen, wie das *remote objects* können. Ein spezieller Java-Übersetzer wird benötigt, um eine lokale Java Klassendefinition automatisch in eine *remote class* zu übersetzen. Ob eine Klasse als *remote class* zu übersetzen ist, wird im Quelltext durch den Klassenmodifikator `remote` festgelegt.

Lokales Objekt: Objekt einer normalen lokalen Java Klasse. Eine Instantiierung eines solchen Objekts ist nur innerhalb der lokalen Java Virtual Machine möglich. Methodenaufrufe bezüglich lokaler Objekte erfolgen niemals über RMI, da auf lokale Objekte auch nur lokal zugegriffen werden kann. Lokale Objekte werden bei RMI-Aufrufen stets per Kopie übergeben. Beim Kopieren werden nur die nicht-statischen und nicht als `transient` deklarierten Variablen mitkopiert. Wenn auf statische Variablen einer Kopie, auf einer anderen virtuellen Maschine zugegriffen wird, bezieht sich dieser Zugriff stets auf die Variablen der dort lokal initialisierten Klasse. Damit lokale Objekte mit *remote objects* interagieren können, müssen deren Klassen ebenfalls mit einem speziellen Java-Übersetzer übersetzt werden.

Remote Object: Objekt einer *remote class*, das sich in einer verteilten Umgebung auf einer beliebigen lokalen JVM befinden kann. Auf *remote objects* kann von allen Maschinen der verteilten Laufzeitumgebung aus zugegriffen werden. Sie können auf einer beliebigen virtuellen Maschine instaniiert werden und es ist jederzeit möglich, solche Objekte auf eine andere Maschine umziehen zu lassen. Nicht-statische Methodenaufrufe finden bei *remote objects* nur dann lokal, d.h. ohne Umweg über RMI, statt, wenn der Aufruf entweder unqualifiziert oder bezüglich `this` bzw. `super` qualifiziert erfolgt. Bei statischen Methoden wird ein Aufruf genau dann lokal ausgeführt, wenn der Aufruf aus der gleichen Klasse stammt, wie die auszuführende Methode. In allen anderen Fällen finden entfernte Methodenaufrufe statt.[2] *Remote objects* werden als Argumente bei lokalen oder entfernten Methodenaufrufen stets per Referenz übergeben und ebenso bei der Ergebnisrückgabe per Referenz zurückgeliefert.

[2]Diese Forderung kann, zur Optimierung des Laufzeitverhaltens, noch insoweit gelockert werden, dass hier als entfernt geforderte Methodenaufrufe auch lokal erfolgen können, wenn Aufrufer und „gerufenes“ Objekt sich auf der gleichen lokalen JVM befinden und sich die Semantik bei der Argumentübergabe und Ergebnisrückgabe nicht ändert. In Abschnitt 3.2 wird diese Lokalitätsoptimierung noch näher untersucht. Für den Benutzer spielt diese Abwandlung aber keine Rolle.

3 Übersetzung von Klassendefinitionen

In diesem Abschnitt wird diskutiert, wie man unter Verwendung des Java RMI eine lokale Klasse so in eine *remote class* übersetzt, dass Instanzen dieser *remote class* in einer verteilten Umgebung interagieren können. Die Umsetzung soll dabei, bezogen auf das lokale Objektmodell, möglichst semantikerhaltend geschehen. Da allerdings, wie bereits in Abschnitt 2.1 erläutert wurde, zwischen dem Java Objektmodell und dem Objektmodell des Java RMI substantielle Unterschiede bestehen, wird dies nicht in allen Fällen möglich sein.

Die Umsetzung einer lokalen Klassendefinition wird also von zwei Seiten maßgebend beeinflusst: von den Anforderungen an *remote objects* und *remote classes* wie sie bereits in den Begriffsdefinitionen unter 2.2 kurz skizziert wurden und den Möglichkeiten, die das Java RMI-System zur Verfügung stellt.
Andererseits bestimmen die Umsetzungsregeln für die lokalen Klassendefinitionen die Verwendung von *remote objects*; d.h. sie geben an, wie man beispielsweise auf Variablen von *remote objects* zugreift oder statische Methoden aufruft.

Die Programmtransformationen, die einen adäquaten Umgang mit *remote objects* ermöglichen, werden auf den folgenden Seiten anhand von Beispielen erläutert. Ein zu transformierender Codeteil `P`[3] wird in den folgenden Übersetzungsbeispielen durch den Ausdruck `toRemote(P)` gekennzeichnet. `toRemote` bezeichnet also die rekursive Abbildungsfunktion, die den Originalquelltext einer Klasse in die gewünschte Form, zur Verwendung in einem verteilten System bringt.

3.1 Statischer Anteil und Instanzenanteil einer Klasse

Der wohl größte Unterschied zwischen lokalen Klassen und *remote classes*, liegt im Umgang mit dem statischen Anteil einer Klassendefinition. Im lokalen Modell wird eine Klasse, beim ersten aktiven Zugriff (siehe hierzu §12.4.1 in [GJS_96]) auf diese, durch die JVM geladen. Dabei werden statische Variablen initialisiert und die Klasseninitialisierungsblöcke ausgeführt. Bevor eine Klasse initialisiert wird, werden erst alle ihre Oberklassen initialisiert, wenn dies zuvor nicht schon bereits geschehen ist.
In einer verteilten Umgebung, die aus mehreren lokalen JVMs besteht, sollte dies möglichst analog ablaufen. Eine Klasse darf hier, bezogen auf das Gesamtsystem, nur genau einmal, also auch nur auf genau einer lokalen JVM initialisiert werden. Alle statischen Zugriffe müssen sich dann auf diese initialisierte Klasse beziehen und damit auch auf dieser JVM ausgeführt werden.

Ein entfernter Zugriff auf statische Methoden oder gar Variablen wird jedoch durch das Java RMI-System nicht unterstützt. Als Konsequenz hieraus ergibt sich eigentlich nur eine Möglichkeit, dieses Problem in den Griff zu bekommen: der statische Klassenanteil wird selbst als ein *remote object* im Sinne von RMI realisiert. Von diesem Objekt gibt es genau eine Instanz auf einer lokalen JVM der Umgebung. Die Instantiierung eines solchen *Klassenobjekts* im verteilten System ist demnach gleichzusetzen mit dem Initialisieren einer Klasse für den lokalen Fall. Das

[3] In allen Übersetzungsbeispielen werden Platzhalter bzw. Bezeichner stets gemäß der folgenden Konventionen vergeben:

A,B,C,...	bezeichnen Klassen oder Interfaces,
x,y,z,...	entsprechen Variablennamen,
I,J,K,...	stellen Ausdrücke dar,
P,Q,R,...	werden anstelle von Anweisungsfolgen geschrieben,
T,U,V,...	stehen für beliebige Typen, können also auch Klassen oder Interfaces bezeichnen.

Laufzeitsystem muss entscheiden, auf welcher Maschine eine Klasse initialisiert wird und muss im folgenden dafür sorgen, dass jeder Zugriff auf die Klasse sich genau auf das instantiierte Klassenobjekt bezieht.

Aus einer lokalen Klassendefinition müssen also zwei verschiedene Klassen generiert werden,[4] wobei die eine Klasse den nicht-statischen Anteil, den sogenannten *Instanzenanteil*, und die andere Klasse alle statischen Felder zugewiesen bekommt. Bei beiden generierten Klassen muss es sich um Klassen für Objekte handeln, auf die über RMI zugegriffen werden kann. Im folgenden Codefragment wird für eine Klasse `B` exemplarisch die Aufteilung der lokalen Klassendefinition in eine Klasse `B_impl`, die den Instanzenanteil enthält, und eine Klasse `B_class_impl`, die den Klassenanteil realisiert, gezeigt. Dabei wurde auf die genaue Angabe der zugehörigen Interfaces `B_intf` und `B_class` verzichtet. Diese entsprechen, den für das Java RMI benötigten *remote interfaces*, die die Signaturen, aller per RMI aufrufbaren Methoden enthalten. Nur über deren Typ werden die `_impl`-Klassen angesprochen.

```
remote class B extends A
                implements C
{
   T         x = I;
   static U  y = J;

   T foo(V z) { P }
   static void foo2() { Q }

   static { R }
}
```

```
// Instanzenanteil
interface B_intf extends A_intf
{ ... }
class B_impl extends A_impl implements B_intf
{
   T         x = I;
   ...
   public T foo(V z) throws RemoteException, MovedException
   { ... toRemote(P) ... }
}

// Klassenanteil
interface B_class extends RemoteClassIntf
{ ... }
final class B_class_impl extends RemoteClass implements B_class
{
   U         y;
   // Initialisierung des Klassenobjekts
   protected void _init()
   {
      y = J;       // statische Variableninitialisierung
      toRemote(R); // Klasseninitialisierung
   }
   ...
   public void foo2() throws RemoteException
   { toRemote(Q) }
}
```

Die Aufteilung einer Klassendefinition in zwei Klassen, auf die entfernt zugegriffen werden kann, erfordert natürlich, im Vergleich zur lokalen Klasse, einen anderen Umgang mit der Klasse selbst und Instanzen davon. Außerdem fällt am Übersetzungsbeispiel auf, dass die generierte Klasse `B_impl` das Interface `C` nicht mehr implementiert. Dies ist jedoch, ohne Änderungen an `C`, auch gar nicht möglich, da sich, schon alleine durch die Anforderungen des RMI, die Signatur der Methoden bei der Transformation verändert. Wie man in Abschnitt 3.2 sehen wird, werden diese Probleme weitestgehend durch eine weiteren Stellvertreterklasse gelöst, die anstelle eines direkten Umgangs mit Stubs verwendet wird. Das obige Übersetzungsbeispiel ist also nur eine vorläufige Lösung, es wird in 3.2 noch vervollständigt.

[4] Wie man in 3.3 sehen wird, besteht eine vollständige Aufteilung einer lokalen Klassendefinition aus drei Teilen: dem Instanzenanteil, dem statischen Anteil und dem Konstruktoranteil.

Schaut man sich am obigen Beispiel insbesonders die Umsetzung der ursprünglichen Klassenhierarchie an, so erkennt man, dass aus einer Hierarchie in der `A` eine Oberklasse von `B` ist, zwei getrennte Klassenhierarchien von entfernten Objekten aufgebaut werden. Abbildung 2 zeigt die näheren Zusammenhänge.

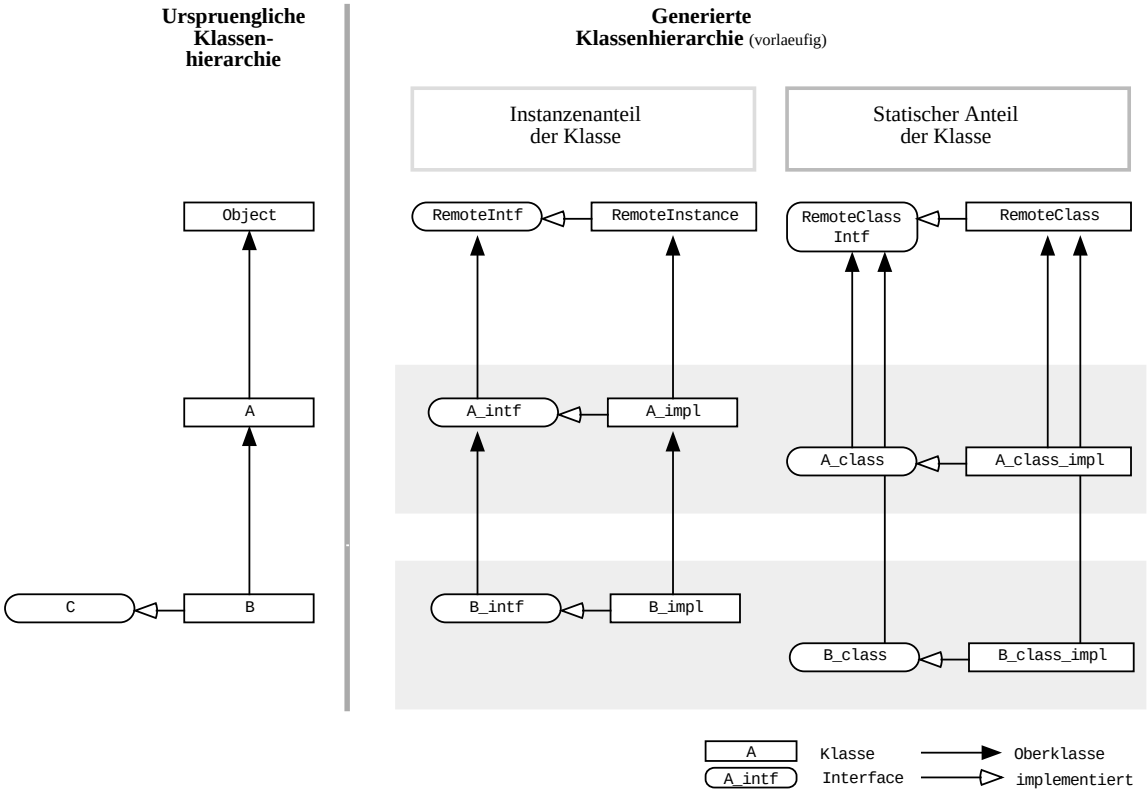


Abbildung 2: Aufteilung der ursprünglichen Klassenhierarchie

Die Wurzel entlang der Oberklassenordnung, der den Instanzenanteil realisierenden Klassen, stellt die Klasse `RemoteInstance` dar. Diese Klasse ist das „entfernte“ Gegenstück zur lokalen Oberklasse `java.lang.Object`. Der dazu gehörige Klassenanteil ist in der Klasse `RemoteClass` zu finden. Diese Klasse ist zugleich auch direkte Oberklasse jeder Klasse, die einen statischen Anteil einer lokalen Klasse realisiert.

Es fällt auf, dass es für die *Klassenobjekte* keine Hierarchie, wie bei den Instanzenklassen gibt. Klassen, die einen statischen Anteil einer lokalen Klasse implementieren, haben niemals eine Unterklasse, können also auch als `final` deklariert werden. Der Grund liegt darin, dass es, gemäß dem lokalen Verhalten, möglich sein muss, dass beim erstmaligen Zugriff auf ein statisches Feld von Klasse `B` auch tatsächlich nur Klasse `B` initialisiert und nicht automatisch Oberklasse `A` mitinitialisiert wird, da `A` ja möglicherweise bereits initialisiert sein könnte. Dies wäre im verteilten System aber automatisch der Fall, wenn die Klassen `A_class_impl` Oberklasse von `B_class_impl` wäre. Wird die Klassenhierarchie aber gemäß Abbildung 2 aufgebaut, sind die Klassenobjekte für `A` und `B` völlig unabhängig und können sich in der verteilten Umgebung sogar auf unterschiedlichen lokalen JVMs befinden. Gemäß dem lokalen Verhalten stellt das Laufzeitsystem aber sicher, dass, bevor eine Unterklasse initialisiert wird, alle Oberklassen bereits initialisiert sind.

Ein weiteres Problem bei der Transformation statischer Definitionen in nicht-statische, besteht

in der Umsetzung statischer Initialisierungen. Diese müssen alle, in der Reihenfolge der in der Klasse gemachten Vereinbarungen, eingesammelt und beim Aufruf eines Klassenobjekt-Konstruktors ausgeführt werden. Wie im letzten Übersetzungsbeispiel zu sehen ist, werden die pseudo-statischen Initialisierungen alle in einer generierten Methode `_init` abgelegt. Diese wird automatisch vom Konstruktor der Oberklasse `RemoteClass` beim Anlegen eines Klassenobjekts aufgerufen. Im hier vorgestellten System bestehen an dieser Stelle einige Einschränkungen bezüglich zulässiger statischer Initialisierungen. In Abschnitt 4.2.2 werden diese Beschränkungen ausführlich erläutert.

Bisher wurde für das Beispielprogramm implizit angenommen, dass die Oberklasse `A` von Klasse `B` ebenfalls als *remote class* vorliegt. Dies ist auch eine sinnvolle Einschränkung, wenn man bedenkt, dass bei der Verwendung von RMI, Stubs und *remote objects* nur bezüglich ihrer entfernten Oberklassen, gleiche Typen besitzen. Wäre Klasse `A` im Beispiel lokal, so wäre `B` zwar trotzdem als *remote class* realisierbar, nur könnte der aus `A` vererbte Klassenanteil nicht angesprochen werden. Es entstünde eine „Hybrid-Klasse“, deren lokales Verhalten vom entfernten Verhalten abweichen würde. Eine solche, mit schwer durchschaubaren Nebeneffekten verbundene Objekt-Semantik, ist jedoch nicht wünschenswert. Daher darf eine als `remote` deklarierte Klasse nur `remote`-deklarierte Oberklassen besitzen.
Da die lokale Klasse `java.lang.Object` implizit Oberklasse aller Java Klassen, also auch der *remote classes* ist, treten in diesem Zusammenhang hier unglücklicherweise automatisch Probleme auf, wenn es darum geht, die von `java.lang.Object` bereitgestellten Methoden auch auf *remote objects* anzuwenden. Eine entsprechende Anpassung der Methoden an die Semantik von *remote objects* durch die Klasse `RemoteInstance` kann im Allgemeinen alleine dieses Problem nicht lösen. In 3.4 wird diese Problematik näher untersucht.

3.2 Objekt-Handles

3.2.1 Indirekter Zugriff auf Klassen- und Instanzenanteil

Wie bereits erwähnt, ist es im vorigen Beispiel nicht mehr möglich, dass die generierte Klasse `B_impl` das Interface `C` implementiert, ohne das Interface `C` selbst zu ändern. Außerdem wurde, gemäß den Überlegungen von 3.1, eine ursprünglich einzige Klasse in zwei Klassen entfernter Objekte aufgesplittet, auf die über RMI zugegriffen werden kann. Der Umgang mit den neuen Klassen wird dadurch wesentlich verkompliziert.

Ein Problem das noch gar nicht angesprochen wurde, ist das der Objekt-Migration. Ein *remote object* auf das man über einen Stub zugreift, lässt sich nicht ohne weiteres auf eine andere lokale JVM transferieren. Der Stub referenziert ein festes Objekt auf einer bestimmten virtuellen Maschine und lässt sich nicht auf eine neue Lage des Objekts „umbiegen“.

Alle diese Probleme lassen sich einfach dadurch lösen, dass man nicht direkt mit Stubs arbeitet, sondern vor diese Stubs noch ein weiteres Objekt schaltet, welches einen aktuellen Stub referenziert. Diese Referenz auf einen Stub, die im folgenden als *object handle* auf ein *remote object* bezeichnet wird, erlaubt es, fast alle angesprochenen Probleme zu entschärfen. Im Gegensatz zum Stub wird das *object handle* so angelegt, dass es, abgesehen von den Variablen, genau die gleiche Signatur wie die ursprüngliche Klasse besitzt. Dies schließt neben den nicht-statischen Methoden auch alle statischen Methoden mit ein. Das *object handle* ist somit in der Lage, auch alle Interfaces der ursprünglichen Klasse zu implementieren. Erst diese Klasse genügt damit den Anforderungen an ein entferntes Gegenstück zur lokalen Klasse. Abbildung 3 zeigt schematisch, wie Methodenaufrufe über Objekt-Handles abgewickelt werden. Für Zugriffe auf

statische Methoden und Variablen benötigt das Handle zusätzlich einen Stub für das betreffende Klassenobjekt. Wie Abbildung 3 zeigt, besorgt sich das Handle diesen Stub bei Bedarf über die Laufzeitumgebung. Wie dies genau abläuft soll an dieser Stelle außer Acht gelassen werden. Dies wird dann in Kapitel 4 diskutiert.

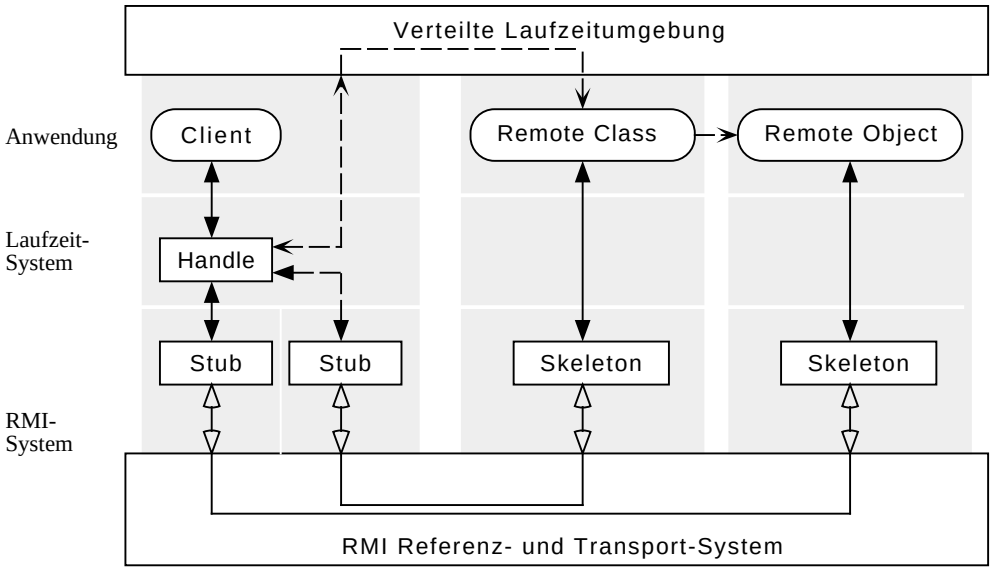


Abbildung 3: Methodenaufrufe über Objekt-Handles

3.2.2 Verhalten bei Objekt-Migration

Das *object handle* regelt über den intern referenzierten Stub den gesamtem Zugriff auf ein *remote object*. Durch die indirekte Referenzierung kann auch relativ einfach das Migrations-Problem gelöst werden: ruft das Handle eine Methode eines umgezogenen Objekts auf, so liefert dieser Methodenaufruf eine Exception `MovedException` zusammen mit einem Stub auf das aktuell gültige Objekt. Nun kann die interne Referenz auf den aktuellen Stub gesetzt und der Methodenaufruf erneut initiiert werden. Der Zugriff auf ein migrationsfähiges Objekt läuft auf diese Weise völlig transparent ab und erfordert keinen besonderen Umgang mit Handles. Sie verhalten sich, abgesehen von Variablenzugriffen, genauso wie die ursprüngliche Klasse.

Diese Lösung hat jedoch auch einige Nachteile. Beim Umzug eines Objekts auf eine andere JVM, muss das alte Objekt weiter auf der bisherigen Maschine zurückbleiben, da es möglicherweise noch Stubs gibt, die dieses mittlerweile veraltete Objekt referenzieren. Das alte Objekt hat dann den Zweck, auf das Umzugsziel zu verweisen. Dazu muss jedes *remote object* eine Variable enthalten, die, falls das Objekt umgezogen ist, einen neuen Stub für das migrierte Objekt enthält.
Zieht ein Objekt jetzt mehrmals zwischen zwei Maschinen um, dann wird im schlimmsten Fall bei jedem Umzug eine neue Kopie des Objekts erzeugt, obwohl sich auf einer Maschinen möglicherweise bereits mehrere, sozusagen identische, veraltete Objekte, jetzt in der Rolle von Verweisen, befinden. Gibt es nun eine Handle, die nicht weiter verwendet wird, die aber ein solches altes Objekt referenziert, dann kann der Garbage Collector keines der veralteten Objekte entlang der Verweiskette zum aktuellen Objekt löschen. Handles auf die kaum Methoden angewendet werden, können also die Ursache dafür sein, dass von einem einzigen Objekt beliebig

viele Kopien im verteilten System entstehen können.
Deswegen bietet das vorgestellte System für jedes Handle eine Dereferenzierungsfunktion an, die die interne Stubreferenz aktualisiert, d.h. indirekte Referenzierungen beseitigt, und dies auch, falls gewünscht, rekursiv für alle vom *remote object* referenzierten Handles durchführt. Vom Benutzer in den Quelltext eingestreute Dereferenzierungen erlauben es wenigstens manuell diese Gefahr einzuschränken und hiermit auch automatisch den Zugriff auf ein *remote object* zu beschleunigen. Aus dem nächsten Programmfragment kann man entnehmen, wie die hierfür generierte Dereferenzierungsfunktion `_deref` aussieht. Das Programmbeispiel soll als Erweiterung des Beispiels aus 3.1 angesehen werden.

```
remote class B extends A
                implements C
{
   T   x = I;
   ...
}
```

```
class B_impl extends A_impl implements B_intf
{
   T         x = I;
   ...
   // Dereferenzierungsfunktion fuer Instanzvariablen
   public void _deref(int level) throws RemoteException
   {  ...
      // wenn T eine remote class bezeichnet, dann Ein-
      // trag der Form:
      if (x != null) x._deref(level);
      ...
   }
   ...
}
```

3.2.3 Einordnung in die generierte Klassenhierarchie

Da Handles für eine Klasse an die Stelle von Objekten der Klasse treten, erhalten die Handle-Klassen den Namen der ursprünglichen lokalen Klassendefinition. Insgesamt werden also anstelle einer, als `remote` deklarierten Klasse `B`, fünf verschiedene Klassen `B`, `B_intf`, `B_impl`, `B_class_intf` und `B_class` generiert, die zusammen mit dem Laufzeitsystem das Verhalten der *remote objects* von `B` realisieren. Abbildung 4 ist eine Erweiterung von Abbildung 2 und zeigt jetzt vollständig den Zusammenhang zwischen den einzelnen Klassen und Interfaces.

Das folgende Beispielprogramm vervollständigt den Code aus 3.1 nun endgültig. Es soll hier insbesonders gezeigt werden, wie die Methoden von Handles aussehen müssen, um eine statische oder nicht-statische Methode für ein indirekt referenziertes *remote object* korrekt aufzurufen. Die Referenz auf den Stub ist dabei in der Variable `ref` der Oberklasse aller Handles, `ObjectHandle`, abgelegt. Auf die Angabe von Variablen wurde im Beispiel diesmal verzichtet, da die Umsetzung von Zugriffsmethoden analog zu normalen, vom Benutzer deklarierten Methoden erfolgt.

Das Auffälligste am generierten Rumpf einer nicht-statischen Handle-Methode, wie z.B. bei Methode `foo`, ist die vermeintliche Endlosschleife. Diese Schleife ist deswegen notwendig, da bei einem Zugriff auf ein migriertes *remote object*, das Objekt nach der Stubaktualisierung bereits wieder umgezogen sein könnte.
Da Klassen, wie in Abschnitt 4 erläutert wird, stets fest auf einer virtuellen Maschine liegen, gibt es für den Aufruf statischer Methoden keine `MovedException`s, weswegen ein entfernter Aufruf aus diesem Grund auch nicht scheitern kann. Eine Schleife ist in diesem Fall also genauso unnötig, wie das Abfangen von `MovedException`s.

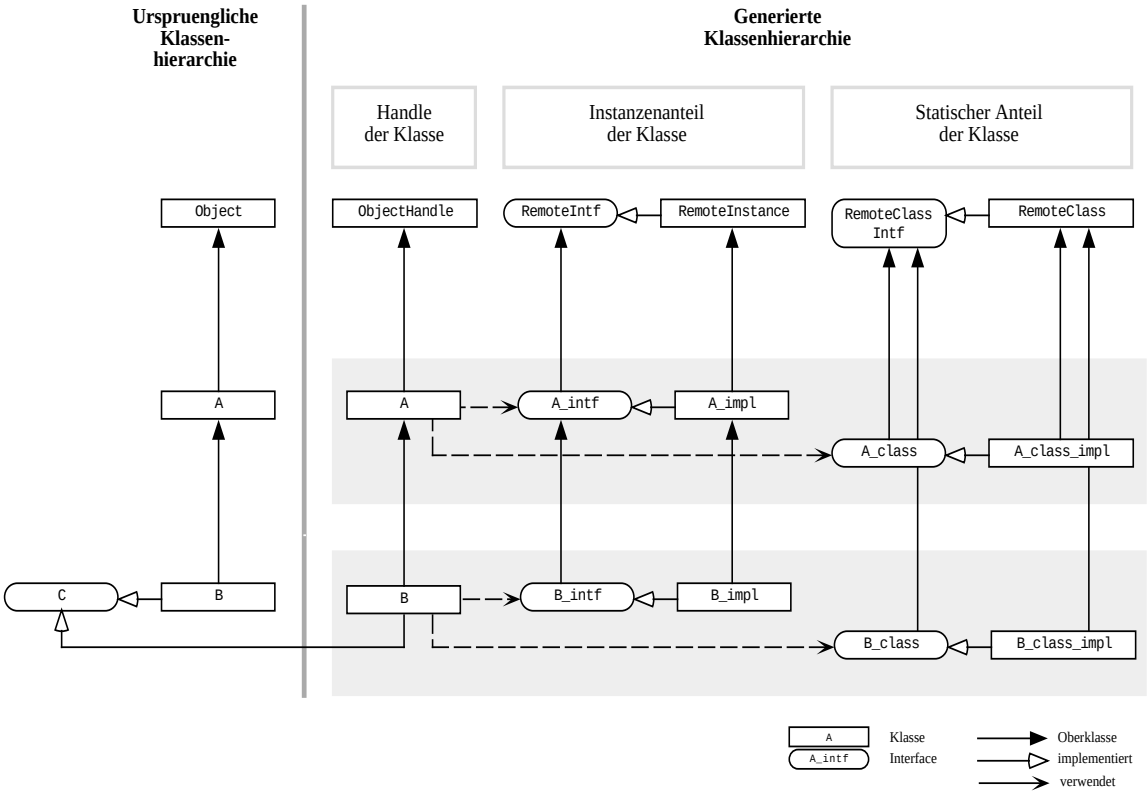


Abbildung 4: Umsetzung der ursprünglichen Klassenhierarchie

```
remote class B extends A
                implements C
{  ...
   T foo(V z) { ... }
   static void foo2() { ... }
   ...
}
```

```
class B extends A
        implements C
{  ...
   T foo(V z)
   {
      while (true)
         try { return ((B_intf)ref).foo(z); }
         catch (MovedException _e)
               { _adaptRef(_e); }
         catch (RemoteException _e)
               { _handleRemoteException("B.foo",_e); }
   }
   static void foo2()
   {
      try
      { return ((B_class)RuntimeEnvironment.
                              getClassObj("B")).foo2(); }
      catch (RemoteException _e)
            { _handleRemoteException("B.foo2", _e); }
   }
   ...
}
```

3.2.4 Lokalitätsoptimierung

Bisher wurde stets davon ausgegangen, dass ein Handle intern einen Verweis auf einen Stub enthält. Befindet sich ein *remote object* auf der gleichen Maschine wie ein Client, der ein Handle

auf das Objekt besitzt, so laufen auch hier sämtliche Methodenaufrufe über das RMI-System ab. Da entfernte Methodenaufrufe, egal wo sich das *remote object* befindet, prinzipiell teuer sind, wäre eine Umgehung vom RMI durch einen lokalen Methodenaufruf hier sicher wünschenswert. Da, wie aus 2.1 bekannt, Stub und Implementation eines *remote objects* bezüglich ihrer entfernt aufrufbaren Methoden den gleichen Typ besitzen, könnte man im Handle statt des Verweises auf den Stub, direkt das *remote object* eintragen und somit wesentlich effizienter auf das Objekt zugreifen. Aus diesem Grund wird bei jeder Objekt-Migration oder Handle-Dereferenzierung vom Laufzeitsystem gepüft, ob ein *remote object* sich auf der gleichen Maschine befindet wie das Handle, über das die Migration oder Dereferenzierung initiiert wurde. Trifft dies zu, so verweist das Handle zukünftig direkt auf das *remote object*.

Man muss sich aber im Klaren sein, dass diese, auf den ersten Blick sehr vielversprechende Optimierung, auch ein grosses Probleme mit sich bringt: bei der Paramterübergabe und der Ergebnisrückgabe verhält sich ein lokaler Methodenaufruf anders als ein entfernter Aufruf. Instanzen lokaler Klassen werden nämlich bei einem RMI-Aufruf per Kopie übergeben, wohingegen lokal natürlich, abgesehen von den Basistypen, stets Referenzübergabe verwendet wird. Methodenaufrufe, in denen ein lokales Objekt übergeben wird, müssen also über RMI abgewickelt werden, damit die in 2.2 definierte Semantik von Methodenaufrufen in der verteilten Umgebung konsistent eingehalten wird. Bei der Umsetzung von Methodenaufrufen bezüglich eines *remote objects* muss also sichergestellt werden, dass in dem hier beschriebenen Fall, ein Handle stets eine Methode über einen Stub aufruft und nicht über ein möglicherweise direkt referenziertes *remote object*. Diese Absicherung muss vom Übersetzer, und nicht vom Laufzeitsystem vorgenommen werden, wenngleich zur Übersetzungszeit nicht immer sichergestellt werden kann, ob ein Argument eines Methodenaufrufs ein lokales oder entferntes Objekt ist. Beispielsweise können Variablen vom Typ `java.lang.Object`[5] sowohl lokale Objekte, als auch *remote objects* enthalten. Wird eine solche Variable als Argument eines Methodenaufrufs übergeben, so muss der Übersetzer pessimistisch vorgehen und in einem solchen Zweifelsfall immer einen Aufruf über RMI generieren.[6]

3.2.5 Trennung zwischen Stub und Handle

Bleibt abschließend noch zu klären, wieso auf der Client-Seite für ein *remote object* zwei Stellvertreterobjekte, nämlich der Stub und die Handle verwendet werden und man nicht gleich dazu übergeht, Stub und Handle zusammenzufassen. Von der Architektur des Java RMI-Systems wäre dies durchaus denkbar. Selbst die Objekt-Migration könnte man, bei dieser Lösung ohne ein explizites Handle, in den Griff bekommen, da jeder Stub intern eine austauschbare Referenz vom Typ `java.rmi.server.RemoteRef` besitzt, welche den konkreten Ort eines *remote objects* beschreibt.
Eine Zusammenfassung der beiden Stellvertreterklassen ist andererseits nur möglich, wenn man die Generierung des integrierten Stellvertreters selbst durchführt und nicht den Standard-Stubübersetzer `rmic` verwendet. Hierfür müsste man viele systeminterne, nicht dokumentierte Details des Java RMI kennen, um die volle Funktionalität des ursprünglichen Stubs auch im integrierten Stellvertreter zur Verfügung stellen zu können. Da zu Beginn dieser Arbeit nur

[5] Man wird später sehen, dass die Klasse `java.lang.Object` noch an mehreren Stellen große Probleme bereiten wird und in vielen Fällen speziell behandelt werden muss.

[6] Alternativ könnte der Übersetzer auch Code erzeugen, der mögliche lokale Argumente bei einem lokal auszuführenden Methodenaufruf, vor dem Aufruf kopiert und die Kopien als Parameter übergibt. Damit würde das Verhalten des RMI-Systems simuliert werden, was im Allgemeinen sicher wesentlich effizienter als ein richtiger RMI-Aufruf wäre.

ein Alpha-Release des Java RMI zur Verfüngung stand, wurde darauf verzichtet, RMI-Interna irgendwelcher Art direkt zu verwenden. Das vorgestellte System benutzt deswegen auch nur die für den RMI-Endbenutzer gedachten und dokumentierten Schnittstellen.

3.3 Objekt-Instantiierung

Normalerweise bietet eine Java Klasse mehrere verschiedene Konstruktoren an, die es erlauben, ein Objekt dieser Klasse zu instantiieren. Das Objekt wird dabei natürlich auf der JVM angelegt, auf der die `new`-Anweisung ausgeführt wird. Es gibt nun prinzipiell zwei Möglichkeiten, um ein neues *remote object* innerhalb einer verteilten Umgebung zu erzeugen: man ruft den Konstruktor auf der lokalen Maschine auf und führt anschließend eine Objekt-Migration auf die gewünschte Zielmaschine aus oder der entsprechende Konstruktor für den Instanzenanteil des *remote objects* wird unmittelbar auf der gewünschten Zielmaschine aufgerufen.

Wie man in 4.2.3 sehen wird, ist die erste Variante ziemlich ineffizient, da eine Objekt-Migration relativ aufwendig ist und mehrere RMI-Aufrufe zur Folge hat. Außerdem ist es durchaus vorstellbar, daß Instanzvariablen eines *remote objects* nicht serialisierbare Objekte enthalten.[7] In diesem Fall ist eine Objekt-Migration nicht möglich, weshalb hier die erste Variante nur lokale Instantiierungen dieser *remote class* zulassen würde.
Bei der zweiten Möglichkeit tritt diese Problematik nicht auf. Sie ist auch wesentlich kostengünstiger als die erste Variante, kann jedoch mit den bisherigen Mitteln nicht direkt realisiert werden. Das Problem liegt darin, dass ein Konstruktoraufruf für verschiedene Konstruktoren und Klassen natürlich stets unterschiedlich ist, sich also schlecht im Laufzeitsystem generalisiert verpacken lässt. Für jede Klasse müsste man zusätzlich noch ein entferntes Objekt auf jeder lokalen Maschine der Umgebung anlegen, das für jeden Konstruktor der Klasse eine Methode anbietet, die genau diesen Konstruktor über `new` aufruft und einen Stub auf das generierte Objekt zurückliefert. Will man nun ein *remote object* auf einer bestimmten lokalen JVM anlegen, so beschafft man sich vom Laufzeitsystem einen Stub auf dieses Konstruktorobjekt der Klasse auf der gewünschten JVM und ruft über RMI die entsprechende *Konstruktormethode* auf.

Dieses Prinzip erfordert jedoch, dass neben den vier zu generierenden Klassen, die den statischen Anteil und den Instanzenanteil einer, als `remote` deklarierten Klasse realisieren, noch zusätzlich ein Interface und eine Klasse erzeugt werden müssten, die mit Konstruktoraufrufen umgehen können. Rechnet man noch die Handle-Klasse und die für das RMI-System nötigen Stub- und Skeletonklassen dazu, käme man auf insgesamt 13 Klassen, die aus einer einzigen Klassendefinition entstehen würden.

Die Idee, wie man doch ohne zusätzliche Konstruktorklassen auskommt, besteht darin, dass man die Funktionalität des Klassenanteils und des Konstruktoranteils vereinigt. Da Konstruktoren auch logisch zum Klassenanteil einer `remote`-deklarierten Klasse gehören, werden die Konstruktormethoden einfach noch dem statischen Klassenanteil hinzugeschlagen. Wie bereits

[7] Man kann zur Übersetzungszeit im Allgemeinen nicht feststellen, ob eine Variable ein nicht-serialisierbares Objekt enthält. Selbst wenn der Typ der Variable einer nicht-serialisierbaren Klasse entspricht, kann die Variable jederzeit ein Objekt referenzieren, das Instanz einer serialisierbaren Unterklasse ist. Als Konsequenz kann der Übersetzer nicht entscheiden, ob ein Objekt einer Klasse nun migrationsfähig ist, oder nicht. Dies ist erst zur Laufzeit möglich, wobei ein gescheiterter Migrationsversuch hier automatisch zu einer Auslösung einer Exception führt. Um dies zu verhindern, erlaubt das hier vorgestellte System, dass man für Objekte explizit festlegen kann, ob ein Umzug möglich ist (näheres siehe 4.2.3).

unter 3.1 erläutert, existiert jedoch im gesamten verteilten System nur auf einer JVM ein Klassenobjekt das eine bestimmte *remote class* repräsentiert. Nur auf dieser Maschine könnte man dann über die Konstruktormethoden *remote objects* anlegen. Man muss also zulassen, dass mindestens auf jeder JVM genau ein solches Klassenobjekt instantiiert wird. Damit aber nicht das Prinzip verletzt wird, dass eine Klasse auf genau einer JVM „geladen“ ist, muss das Laufzeitsystem sicherstellen, dass es unter den Klassenobjekten genau ein Objekt gibt, dessen Variablen, die statischen Variablen der Klasse darstellen. Ursprünglich statische Methodenaufrufe müssen dann stets in RMI-Aufrufe bezüglich dieses Objekts umgesetzt werden. Die Variablen in den restlichen Klassenobjekten haben also keine Bedeutung und werden noch nicht einmal explizit initialisiert. Deswegen werden die restlichen Objekte auch als *Konstruktorobjekte* bezeichnet. Diese Vorgehensweise bedeutet sicherlich, im Vergleich zur ersten Lösung mit expliziten Konstruktorklassen, dass Speicher verschenkt wird. Da die Umgebung jedoch meist aus relativ wenigen lokalen JVMs besteht und bei den meisten Klassen auch die Anzahl der statischen Variablen sich in Grenzen hält, macht sich das nicht weiter negativ bemerkbar.

Zusammenfassend kann man also feststellen, dass der statische Anteil einer Klassendefinition neben statischen Variablen und Methoden auch noch sogenannte Konstruktormethoden anbietet, die ein zur Klasse passendes *remote object* erzeugen. Instanzen der Klasse, die einen solchen statischen Klassenanteil realisieren, lassen sich in zwei Kategorien einteilen: *Klassenobjekte* und *Konstruktorobjekte*. Für eine `remote` deklarierte Klasse gibt es in der ganzen Umgebung nur genau ein Klassenobjekt, das die Klasse im verteilten System repräsentiert. Alle Zugriffe auf statische Variablen oder Methoden beziehen sich auf dieses Objekt. Neben dem Klassenobjekt gibt es auf jeder lokalen JVM des Systems maximal ein Konstruktorobjekt, das es erlaubt, auf der beheimateten JVM entsprechende *remote objects*, also Objekte des Instanzenanteils, anzulegen.

Die vorgestellte Lösung des Instantiierungsproblems bietet, im Hinblick auf spätere Lokalitätsoptimierungen, noch den Vorteil, dass die Konstruktorobjekte als Replikate des Klassenobjekts lokal verwendet werden können. Damit wäre es dann auch für statische Methodenaufrufe möglich, das RMI-System zu umgehen und somit Latenzzeiten zu vermeiden.

Abbildung 5 veranschaulicht den Ablauf einer Objekt-Instantiierung eines *remote objects*. Dabei müssen mehrere Stationen durchlaufen werden, bis schließlich auf der Zielmaschine das tatsächliche *remote object* erzeugt wird. Die Quellcodeangaben im Diagramm werden mit dem darauffolgenden Beispielprogramm verständlich.

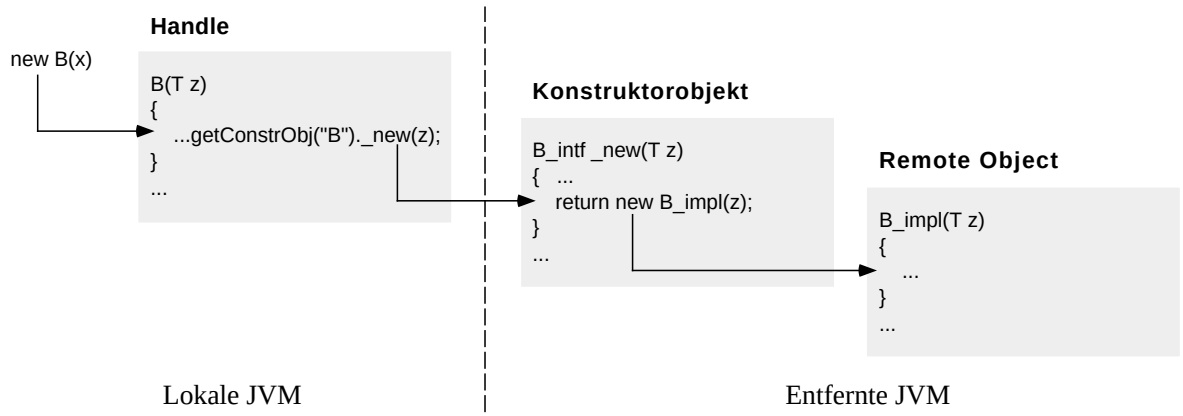


Abbildung 5: Instantiierung eines *remote objects*

Im folgenden Beispiel wird nun auf Quelltextebene verdeutlicht, wie Konstruktordeklarationen umgesetzt werden. Aus jedem Konstruktor in `B` entsteht in der Klasse `B_class_impl` eine Konstruktormethode `_new` mit gleicher Signatur. Ein Konstruktoraufruf bezüglich der ursprünglichen Klasse, wie z.B. „`new B()`“, bezieht sich nach der Transformation auf die Handle-Klasse. In jedem Handle-Konstruktor muss also über ein Konstruktorobjekt eine korrespondierende `_new`-Methode aufgerufen werden, die auf einer entfernten virtuellen Maschine ein *remote object* erzeugt. Auf dieses Objekt verweist dann das neu erzeugte Handle.

```
remote class B extends A
               implements C
{
   B() { P }
   B(T z) { Q }
}
```

```
class B_impl extends A_impl
             implements B_intf
{
   B_impl() throws RemoteException
   { toRemote(P) }
   B_impl(T z) throws RemoteException
   { toRemote(Q) }
}

final class B_class_impl extends RemoteClass implements B_class
{
   public B_intf _new() throws RemoteException
   {
      B_impl _x = new B_impl();
      synchronized (LocalJVM.lock) { _x._calledMethods--; }
      return _x;
   }
   public B_intf _new(T z) throws RemoteException
   {
      B_impl _x = new B_impl(z);
      synchronized (LocalJVM.lock) { _x._calledMethods--; }
      return _x;
   }
}

class B extends A implements C
{
   B()
   {
      try { ref = ((B_class)RuntimeEnvironment.
                          getConstrObj("B"))._new(); }
      catch (RemoteException _e)
            { _handleInstantException("B", _e); }
   }
   B(T z)
   {
      try { ref = ((B_class)RuntimeEnvironment.
                          getConstrObj("B"))._new(z); }
      catch (RemoteException _e)
            { _handleInstantException("B", _e); }
   }
}
```

Es bleibt noch zu klären, wie denn überhaupt Klassen- bzw. Konstruktorobjekte auf entfernten JVMs angelegt werden. Es sieht auf den ersten Blick so aus, als bräuchte man wiederum für jede Implementierung eines statischen Klassenanteils ein eigenes Konstruktorobjekt, usw. Glücklicherweise läßt sich aber die Instantiierung der Klassen- bzw. Konstruktorobjekte allgemein für beliebige *remote classes* formulieren, indem man die Methode `newInstance` aus der Klasse `java.lang.Class` verwendet. Diese erlaubt es, den parameterlosen Konstruktor für eine beliebige Klasse aufzurufen. Da es nur genau diesen einen Konstruktor in einer Realisierung eines statischen Klassenanteils gibt, ist es ohne weiteres möglich, Instanzen davon zu erzeugen.

3.4 Methodendeklarationen

3.4.1 Allgemeine Umsetzung statischer und nicht-statischer Methoden

Methoden einer lokalen Klasse lassen sich recht einfach in Methoden für *remote objects* umwandeln. Es muss lediglich die für RMI-Aufrufe nötige `RemoteException`, zu der Menge der möglicherweise auftretenden Exceptions hinzugefügt werden. Nicht-statische Methoden liefern, falls sie auf ein Objekt angewendet wurden, das umgezogen ist, eine `MovedException`. Diese muss ebenfalls zur Menge der auftretbaren Exceptions hinzugefügt werden. Außerdem ist es natürlich notwendig, dass bevor der ursprüngliche Methodenrumpf ausgeführt wird, überprüft wird, ob das Objekt überhaupt noch ansässig ist. Falls dies nicht der Fall ist, wird besagte Exception ausgelöst, wobei der aktuelle Stub als Information mitgegeben wird. Im Zusammenhang mit der Objekt-Migration sind beim Aufruf einer Methode auch noch einige Verwaltungsinformationen abzufragen.
Beispielsweise kann es bei nebenläufigen Programmen vorkommen, dass gleichzeitig von mehreren Threads aus auf ein Objekt zugegriffen wird. Ein Objekt, auf das aktuell eine Methode angewendet wird, kann natürlich nicht umziehen. Es ist deswegen notwendig, dass jedes *remote object* einen internen Zähler `_calledMethods` enthält, der angibt, wieviele Methoden augenblicklich auf das Objekt angewendet werden.
Umgekehrt ist der Vorgang der Objekt-Migration keine atomare Aktion. Eine interne Variable muss für jedes *remote object* anzeigen, ob das Objekt momentan gerade dabei ist umzuziehen. In diesem Fall wird das Objekt solange jeden Zugriff ablehnen, bis der Umzug abgeschlossen und ein Stub auf das neue Objekt verfügbar ist. Die zugreifende Instanz muss in diesem Fall in einer Schleife solange das Objekt abfragen, bis der Zugriff schließlich wieder möglich ist.
Alle genannten Variablen werden durch die Oberklasse `RemoteInstance` jedes *remote objects* zur Verfügung gestellt. Diese Klasse enthält eine Methode `_enter` welche alle genannten Überprüfungen zu Beginn eines Methodenaufrufs durchführt.

Wie aus dem folgenden Beispiel zu ersehen ist, sind die genannten Überprüfungen nur für die Instanzenklassen notwendig. Klassenobjekte können im vorgestellten System nicht umziehen, weshalb hier auch kein zusätzlicher Verwaltungsaufwand notwendig ist.

```
remote class B extends A
               implements C
{  ...
   T foo(V z) { P }
   static void foo2() { Q }
   ...
}
```

```
class B_impl extends A_impl
             implements B_intf
{  ...
   public T foo(V z) throws RemoteException, MovedException
   {
      _enter();
      try     { toRemote(P) }
      finally { synchronized (LocalJVM.lock)
                  { _calledMethods--; }}
   }
   ...
}

final class B_class_impl extends RemoteClass
                         implements B_class
{  ...
   public void foo2() throws RemoteException
   { toRemote(Q) }
   ...
}
```

Wie die entsprechenden Methoden in der Handle-Klasse aussehen, kann im Beispiel von Abschnitt 3.2 nachgeschlagen werden. Man sieht hier, dass zwischen den ursprünglichen Methodendefinitionen und den Deklarationen im Handle keine Unterschiede bestehen. Methodenaufrufe können also im Quelltext ohne Änderungen übernommen werden. Nur wird, im Gegensatz zur lokalen Ausführung, in der verteilten Umgebung jedesmal indirekt über eine Handle-Methode ein entfernter Methodenaufruf durchgeführt.

3.4.2 Sonderfälle

Wie bereits angesprochen, ergeben sich jedoch bei dem hier vorgestellten Schema für die Umsetzung von Methodendeklarationen und -aufrufen Probleme, wenn die ursprüngliche Methode eine Methode aus der einzigen, nicht-entfernten Oberklasse von `B`, `java.lang.Object`, überschreibt. Überschreibt eine Methode nämlich eine Methode einer Oberklasse, so darf sie keine zusätzlichen Exceptions liefern. Bei der Programmtransformation erhalten die Methoden jedoch zusätzlich noch die Exceptions `RemoteException` und `MovedException` zugewiesen. Dies würde beim Übersetzen der `_impl`-Klasse des Instanzenanteils zu einer Fehlermeldung des Übersetzers führen.

Die, auf den ersten Blick einfachste Möglichkeit, das Problem zu umgehen, indem z.B. die Methode `toString` einfach in der generierten Klasse in `_toString` umbenannt und alle Methodenaufrufe von `toString` bezüglich dieser Klasse konsistent in Aufrufe von `_toString` umgewandelt werden, ist unbrauchbar. Man kann nämlich zur Übersetzungszeit überhaupt nicht sicherstellen, ob eine Methodenanwendung bezüglich eines *remote objects* stattfindet oder nicht. Folgendes Beispiel zeigt einen solchen Fall:

```
remote class B extends A
{
    Object  obj;

    void print()
    {
        System.out.println(obj.toString());
    }
}
```

Da `obj` ein beliebiges Objekt, also auch ein *remote object* referenzieren kann, kann für den Aufruf „`obj.toString()`“ nicht entschieden werden, ob er durch „`obj._toString()`“ ersetzt werden muss.
Dieses Dilemma läßt sich nur durch eine trickreiche Sonderbehandlung der einzelnen Methoden der Klasse `java.lang.Object` beheben. Glücklicherweise sind die meisten Methoden dieser Klasse `final`, weshalb ein Überschreiben nur für die Methoden `toString`, `equals`, `hashCode`, `clone` und `finalize` möglich ist. Stellvertretend für die ersten vier Methoden, zeigt das folgende Beispiel, die nötige Sonderbehandlung am Beispiel der Methode `toString`:

```
remote class B extends A
{
   ...
   public String toString() { P }
   ...
}
```

```
class B_impl extends A_impl
             implements B_intf
{
   ...
   public String toString()
   {
       try { toRemote(P) }
       catch (MovedException _e) {}
       catch (RemoteException _e)
       { ObjectHandle._handleRemoteException
                          ("B.toString",_e); }
   }

   public String _toString() throws RemoteException,
                                    MovedException
   {
      _enter();
      try     { return toString(); }
      finally { synchronized (LocalJVM.lock)
                  { _calledMethods--; }}
   }
   ...
}

class B extends A  // Handle-Klasse
{
   ...
   public String toString()
   {
      while (true)
         try { return ((B_intf)ref)._toString(); }
         catch (MovedException _e)
               { _adaptRef(_e); }
         catch (RemoteException _e)
               { _handleRemoteException("B.toString",e); }
   }
   ...
}
```

Man sieht, dass der Bezeichner für die Methode `toString` in der Handle beibehalten wird – dort bereitet dies ja auch noch keine Probleme. In der `_impl`-Klasse jedoch gibt es zwei Versionen, die die Methode implementieren. Die entfernt aufrufbare Methode heißt `_toString` und wird von der Handle-Methode aus aufgerufen. Diese Version enthält lediglich einen Aufruf der Methode `toString`, die ihrerseits die entsprechende Methode in `java.lang.Object` überschreibt. In `toString` findet sich schließlich die ursprüngliche Implementierung. Da die Methode in dieser Version keine Exceptions liefern darf, müssen zusätzlich alle möglicherweise auftretenden Exceptions abgefangen werden.
Dass diese Umsetzung korrekt ist, solange keine Exceptions auftreten, müsste klar sein. Es bleibt lediglich zu diskutieren, ob sich das System richtig verhält, wenn eine Ausnahme auftritt. Die Untersuchung läßt sich auf das Auftreten einer `RemoteException` beschränken, da eine `MovedException`, auch wenn sie an der Stelle abgefangen werden muss, niemals in dieser Situation auftreten kann. Der Grund liegt darin, dass `toString` stets nur von einer Methode innerhalb der Klasse `B_impl` aufgerufen wird. Von außerhalb wird ja immer der „Umweg“ über die Handle genommen. Wenn nun bereits eine Methode auf das Objekt angewendet wird, so ist man sicher, dass das Objekt auch ansässig ist und zur Zeit auch nicht migrieren kann. Eine `MovedException` kann also gar nicht auftreten, es braucht demnach also auch keine Fehlerbehandlung angegeben zu werden.

Eine `RemoteException` kann jederzeit erfolgen, muss also auch geeignet behandelt werden. Es wird hierzu einfach die Fehlerbehandlung aus dem Handle verwendet, womit der Aufruf sich im Fehlerfall genauso verhält, als ob der Aufruf direkt über das Handle erfolgen würde – mit dem Unterschied, dass die Fehlerbehandlung eben schon innerhalb der Methode stattfindet und nicht erst außerhalb im Handle. Da sich Handle und *remote object* möglicherweise auf unterschiedlichen lokalen JVMs befinden, ist im schlimmsten Fall der Ort der Fehlerbehandlung unterschiedlich.

Bisher wurde nur diskutiert, wie man Aufrufe bezüglich von Methoden, die in der Oberklasse aller Klassen `java.lang.Object` definiert werden, in einer verteilten Umgebung korrekt durchführen kann. Es versteht sich jedoch von selbst, dass die Semantik der Methoden `equals`, `hashCode` und `clone` für das verteilte System ebenfalls abgeändert werden muss. Da die Klasse `RemoteInstance` Oberklasse aller *remote object*-Implementationsklassen ist, werden entsprechend angepasste Versionen dieser Methoden dort vereinbart.

Für die Methode `finalize` aus der Klasse `java.lang.Object` ist das erläuterte Übersetzungsschema allerdings, aufgrund der besonderen Semantik der Methode, nicht geeignet. Würde man eine Umsetzung wie beschrieben machen, so würde ein nicht mehr referenziertes Handle bei der Garbage-Collection eine Ausführung der `finalize`-Methode, des vom ihm referenzierten *remote objects*, auslösen. Um dies zu verhindern und ein korrektes Verhalten zu erzwingen, genügt es, wenn man die ursprüngliche Definition in die `_impl`-Klasse übernimmt und wie oben beschrieben die möglicherweise auftretenden Exceptions intern abfängt. Dann wird die `finalize`-Methode auch nur aufgerufen, wenn der DGC[8] feststellt, dass das *remote object* im gesamten verteilten System nicht mehr referenziert wird.

3.5 Variablendeklarationen

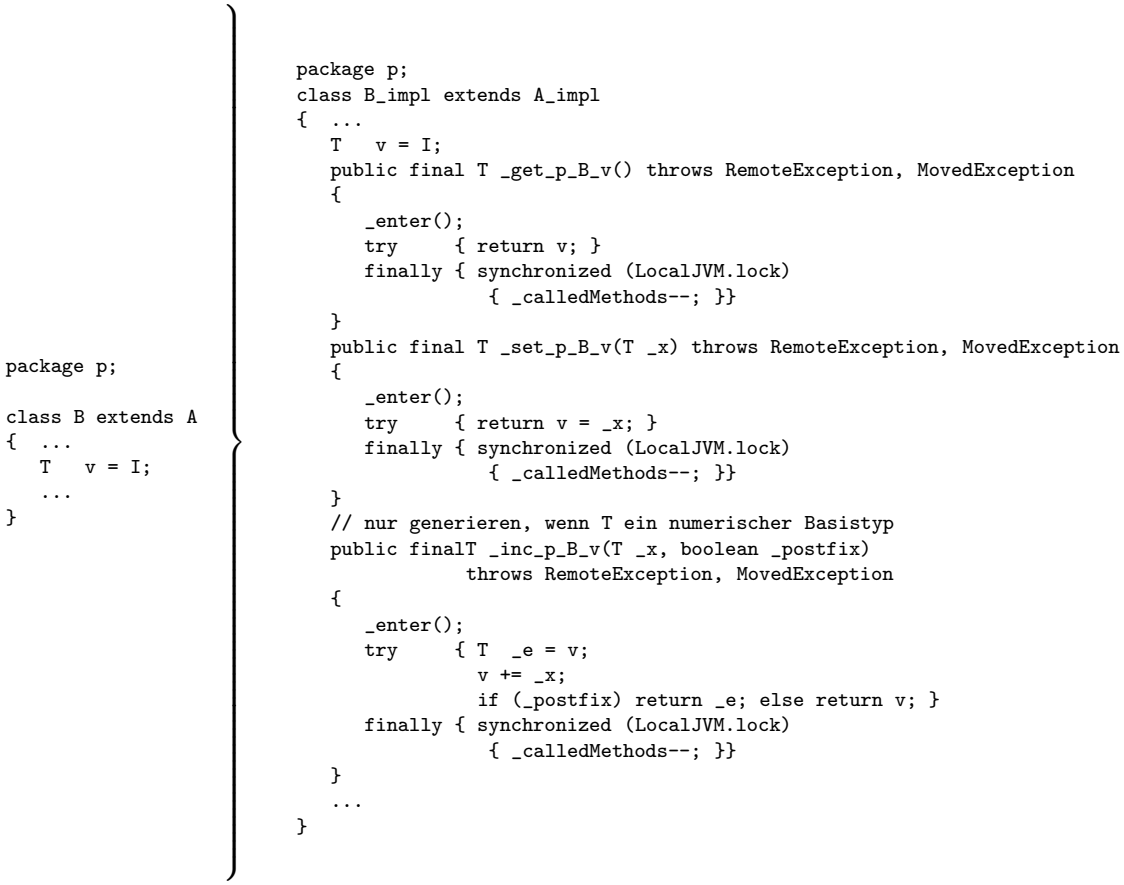
3.5.1 Allgemeine Umsetzung von Variablendeklarationen

RMI unterstützt lediglich einen entfernten Methodenaufruf. Zugriffe auf Variablen eines entfernten Objekts sind über RMI nicht möglich. Um diese Einschränkung zu umgehen, ist es notwendig, für alle Variablen Zugriffsmethoden zu generieren, die es erlauben, einen Variablenwert entfernt zu lesen bzw. neu zu setzen. Je nachdem ob die Variable statisch oder nicht-statisch ist, müssen die Zugriffsfunktionen entweder in der Realisierung des Instanzenanteils oder im statische Klassenanteil definiert werden.

Trifft man innerhalb eines Ausdrucks auf einen Zuweisungsoperator an eine Variable eines *remote objects* oder wird der Wert einer solchen Variable ausgelesen, so muss an Stelle dieses Ausdrucks ein geeigneter Aufruf der entsprechenden Zugriffsmethode erfolgen. In Abschnitt 3.6.5 wird näher untersucht, in welchen Fällen, welcher Zugriffsmethodenaufruf erfolgen muss.

Die Transformation, die für jede Variablendeklaration durchgeführt werden muss, verdeutlicht das folgende Beispiel. Auf die Angabe der entsprechenden Methoden im Handle wurden der Übersichtlichkeit wegen verzichtet. Es wird im Beispiel nur die Umsetzung einer nicht-statischen Variablendeklaration gezeigt. Statische Variablen lassen sich jedoch analog behandeln. Die Zugriffsmethoden werden in diesem Fall in der Realisierung des Klassenanteils vereinbart.

[8] *Distributed Garbage Collector*, Garbage-Collector des RMI-Systems, der mit einer verteilter Objekt-Semantik umgehen kann.



```
package p;

class B extends A
{  ...
   T   v = I;
   ...
}
```

```
package p;
class B_impl extends A_impl
{  ...
   T   v = I;
   public final T _get_p_B_v() throws RemoteException, MovedException
   {
      _enter();
      try     { return v; }
      finally { synchronized (LocalJVM.lock)
                 { _calledMethods--; }}
   }
   public final T _set_p_B_v(T _x) throws RemoteException, MovedException
   {
      _enter();
      try     { return v = _x; }
      finally { synchronized (LocalJVM.lock)
                 { _calledMethods--; }}
   }
   // nur generieren, wenn T ein numerischer Basistyp
   public finalT _inc_p_B_v(T _x, boolean _postfix)
               throws RemoteException, MovedException
   {
      _enter();
      try     { T  _e = v;
                v += _x;
                if (_postfix) return _e; else return v; }
      finally { synchronized (LocalJVM.lock)
                 { _calledMethods--; }}
   }
   ...
}
```

Die generierten Zugriffsfunktionen müssen auch dazu hergenommen werden, die für die numerischen Basistypen definierten Zuweisungsoperatoren `+=`, `-=`, `*=`, usw. sowie die Inkrement- und Dekrementoperatoren `--` und `++` nachzubilden. Für die binären Zuweisungsoperatoren ist dies auch nicht weiter problematisch. So kann man beispielsweise „`b.v += 5`“ einfach durch den Ausdruck „`b._set_p_B_v(b._get_p_B_v() + 5)`“ ersetzen.

Für die unären Postfixoperatoren ist aber eine Umsetzung unter Verwendung der `_set`- und `_get`-Methoden äußerst aufwendig und sehr ineffizient. Deswegen wird genau für diesen, doch ziemlich oft vorkommenden Fall, eine zusätzliche Zugriffsmethode, im Beispiel `_inc_p_B_v`, generiert, mit deren Hilfe der Inkrement- und Dekrement-Operator relativ effizient nachgebildet werden kann. Die Erzeugung dieser Methode ist nur für numerische Basistypen (`byte`, `short`, `char`, `int`, `long`, `float`, `double`) notwendig. Die `_inc`-Methode erlaubt es, zu dem Variablenwert einen bestimmten Wert hinzuzuaddieren und liefert, ja nachdem, ob ein Post- oder Präfixoperator erwartet wird, den aktuellen Wert, oder den alten Wert vor der Addition.

3.5.2 Variablen eines Array-Typs

Problematisch sind Variablenzugriffe, die sich auf Arrays beziehen. Zur Veranschaulichung soll als Beispiel die Variablendeklaration „`long[][] a;`“ betrachtet werden. Die nach obigem Schema generierten Zugriffsfunktionen, lassen nur zu, dass das gesamte Array auf einmal gelesen oder geschrieben wird. Es ist hier nicht möglich, einzelne Elemente neu zu setzen. In der Tat kann auf die Variable `a` äußerst vielseitig zugegriffen werden, je nachdem, wieviele Dimensionen indiziert werden: z.B. „`a[1][2] = 7`“, „`a[3] = new int[8]`“, usw. Man beachte, dass in allen

Fällen, der Zuweisungsausdruck auch einen unterschiedlichen Typ besitzt. Deswegen werden für Arrays mehrere Schreibfunktionen angeboten. Für das obige Beispielfeld `a` müssen insgesamt drei `_set`-Methoden generiert werden:

```
long[][] _set_P_B_a(long[][] _x) { ... }              für a = _x
long[] _set_P_B_a(long[] _x, int _x$0) { ... }        für a[_x$0] = _x
long _set_P_B_a(long _x, int _x$0, int _x$1) { ... }  für a[_x$0][_x$1] = _x
```

Allerdings nur bei voll indiziertem Zugriff auf ein Array kann es nötig werden, auch noch eine `_inc`-Methode für das Array zu generieren. Nmlich genau dann, wenn es sich um ein Array eines numerischen Basistyps handelt.
Auch die Erzeugung von mehreren Lesefunktionen bringt Vorteile, da diese im Durchschnitt einen wesentlich schnelleren Zugriff auf einzelne Feldelemente erlauben. Statt des gesamten Arrays muss hier nämlich nur noch der indizierte Wert per RMI verschickt werden.

3.5.3 Variablen vom Typ `java.lang.Object`

Im Zusammenhang mit Arrays treten auch für Variablen des Typs `java.lang.Object` Konflikte auf. In Java können nämlich Arrays auch `java.lang.Object`-Variablen zugewiesen werden. Für Variablen dieses Typs wird allerdings nur eine Schreibmethode generiert – mehrere Zugriffsmethoden mit unterschiedlicher Indizierung zu erzeugen, gäbe auch gar keinen Sinn, da man Arrays beliebiger Dimension in `java.lang.Object`-Variablen ablegen kann. Für solche Variablen benötigt man also noch eine Schreibfunktion, die die Variable als beliebiges Array interpretiert und eine unbegrenzte Indizierung erlaubt. Das folgende Beispiel zeigt, wie eine solche Spezialbehandlung aussieht:

```
package p;

class B extends A
{  ...
   Object v;
   ...
}
```

```
package p;
class B_impl extends A_impl
{  ...
   Object v;
   public final Object _get_p_B_v() throws RemoteException, MovedException
   {
      _enter();
      try     { return v; }
      finally { synchronized (LocalJVM.lock)
                  { _calledMethods--; }}
   }
   public final Object _set_p_B_v(Object _x)
                        throws RemoteException, MovedException
   {
      _enter();
      try     { return v = _x; }
      finally { synchronized (LocalJVM.lock)
                  { _calledMethods--; }}
   }
   public final Object _set_p_B_v(Object _x, ArrayIndex _y)
                        throws RemoteException, MovedException
   {
      _enter();
      try     { return RemoteTools.setArrayElem(v,_x,_y); }
      finally { synchronized (LocalJVM.lock)
                  { _calledMethods--; }}
   }
   ...
}
```

Unter Zuhilfenahme der zweiten Schreibefunktion lässt sich nun beispielsweise die Zuweisung „`((int[][])b.v)[0][1] = 2`“ (es wird angenommen, dass die Variable `b.v` bereits ein erzeugtes Array vom Typ `int[][]` enthält) folgendermaßen übersetzen: „`b._set_p_B_v(RemoteTools.wrap(2),new ArrayIndex(0,1))`“. Wie man sieht, kann es notwendig werden, dass man einen Wert eines Basistyps erst in ein Objekt einer geeigneten *Wrapper*-Klasse konvertieren muss, bevor man es als neuen Wert übergeben kann. Außerdem müssen die Indizes in Form einer Liste übergeben werden; diese wird durch die Klasse `ArrayIndex` realisiert. Da Java keine Einschränkung bezüglich einer maximalen Array-Dimensionalität kennt, sind mit der Klasse `ArrayIndex` beliebig lange Indexlisten generierbar.
Der oben angegebene erzeugte Zugriffsmethodenaufruf ist allerdings in der bisherigen Form noch nicht ganz korrekt, da eine Zuweisung in Java ein Ausdruck mit einem bestimmten Typ darstellt. Eine richtige Transformation müsste einen Ausdruck erzeugen, der den neuen Wert des Feldelements „`((int[][])b.v)[0][1]`“ (also im Beispiel `2`) als Ergebnis liefert. Die Zugriffsmethode macht dies zwar, allerdings erhält man die `2` in Form eines `Integer`-Objekts und nicht als `int`-Grundtyp. Bei Basistypen muss also das Ergebnis noch aus dem entsprechenden Wrapper-Objekt wieder ausgepackt werden, wohingegen bei normalen Klassen- oder Interfacetypen eine geeignete Typkonversion genügt. Der korrekt transformierte Ausdruck sieht nun folgendermaßen aus: „`((Integer)b._set_p_B_v(RemoteTools.wrap(2),new ArrayIndex(0,1))).intValue()`“.
Diese spezielle, zusätzliche Schreibmethode muss aber nicht nur für `java.lang.Object`-Variablen generiert werden. Auch für Arrays mit Elementtyp `java.lang.Object` wird diese Zugriffsfunktion benötigt, da ja die Elemente selbst wiederum beliebig Arrays und hier sogar Arrays verschiedenen Typs enthalten können. Man kann sich vorstellen, dass das Transformieren solcher Ausdrücke dann äußerst kompliziert werden kann.

Als Ergebnis dieser Diskussion über die Sonderbehandlung von Arrayzugriffen kann man zusammenfassend feststellen, dass Zugriffe auf Arrays durch spezielle Methoden beschleunigt und teilweise erst uneingeschränkt ermöglicht werden. Man muss sich aber auch bewusst sein, dass dem Effizienzgewinnen bei Zugriffen über spezialisierte Zugriffsmethoden, eine drastische Zunahme der Anzahl der Methoden gegenübersteht. Bereits für ein zweidimensionales Array vom Typ `java.lang.Object[][]` werden vier Schreibfunktionen und drei Lesefunktionen – also insgesamt sieben Zugriffsmethoden – erzeugt.

3.5.4 Behandlung überschatteter Variablendefinitionen

Wie dem obigen transformierten Codefragment zu entnehmen ist, beginnen generierte Bezeichner stets mit einem Unterstrich „`_`“. Wird ein Bezeichner aus mehreren Bezeichnern des Originals zusammengesetzt, so wird auch in diesem Fall zur Verdeutlichung der Konkatenation ein Unterstrich verwendet. Ein Beispiel hierfür sind die Namen der Zugriffsfunktionen aus dem obigen transformierten Codefragment. Gefolgt vom Präfix `_get`, `_set` bzw. `_inc` wird der voll qualifizierte Bezeichner der Variable als Name der Zugriffsfunktion gewählt.
Würde man statt des voll qualifizierten Variablennamens lediglich den einfachen Namen im Zugriffsmethodenbezeichner verwenden und sich auf Javas Überschreibungsmöglichkeit bei Methoden verlassen, gäbe es Probleme bei überschatteten Variablen. Folgendes Beispiel demonstriert dies:

```
class Point
{
    int x = 1;
```

```
    int get_x() { return x; }
}

class Test extends Point
{
    int x = 2;
    int get_x() { return x; }
    public static void main(String[] args)
    {
        Point   p = new Test();
        System.out.println(p.x + " " + p.get_x());
    }
}
```

Führt man Klasse `Test` aus, erhält man die Ausgabe „1 2“. Der Grund liegt darin, dass Verweise auf Variablen statisch zur Übersetzungszeit bestimmt, nicht-statische Methodenaufrufe – entfernt aufrufbare Methoden, insbesondere Zugriffsmethoden sind niemals statisch – jedoch zur Laufzeit aufgelöst werden. Da `p` vom Typ `Point` ist (Übersetzungszeittyp), bezieht sich `p.x` auf die Variable `x` aus Klasse `Point`. Für `p.get_x()` wird erst zur Laufzeit bestimmt, welche Methode aufgerufen wird. Da `p` zur Laufzeit ein `Test`-Objekt referenziert, greift man in diesem Fall über die Zugriffsmethode auf die Variable `x` aus Klasse `Test` zu.

Verwendet man bei Variablenzugriffen Zugriffsmethoden, so muss man sicherstellen, dass dabei kein Überschreiben stattfindet. Sonst würde man niemals mehr auf überschattete Variablen zugreifen können. Bei voll qualifizierten Variablennamen im Bezeichner der Zugriffsmethode hat man keine Probleme zu erwarten, eine Zugriffsmethode wird niemals überschrieben. Aus diesem Grund können Zugriffsmethoden stets als `final` deklariert werden. Bei einem Variablenzugriff muss der Übersetzer dann zur Übersetzungszeit bestimmen, welche Methode konkret, anstelle des Variablenzugriffs, aufzurufen ist.

3.6 Anweisungen und Ausdrücke

3.6.1 Grundlagen

Bisher wurde im Wesentlichen nur auf die Übersetzungskonzepte von Deklarationskonstrukten, wie z.B. Klassen-, Methoden-, Variablen- oder Konstruktordefinitionen der Sprache eingegangen. In den Übersetzungsbeispielen wurde an den Stellen, an denen Programmtransformationen betreffend Anweisungen und Ausdrücken nötig waren, dies stets nur durch Anwendung der Umsetzungsfunktion `toRemote` auf den betreffenden Code, angezeigt. Damit aber mit den transformierten Deklarationen auch adäquat umgegangen wird, sind gerade diese Umsetzungen äußerst wichtig.

Für die meisten Anweisungen und Ausdrücke ist die Umsetzung trivial: ein Sprachkonstrukt wird einfach wieder auf das gleiche Konstrukt abgebildet. Die Transformationsregel sorgt in diesem Fall nur für eine rekursive Anwendung der Umsetzungsfunktion auf Teilanweisungen und -ausdrücke. Dies wird im folgenden Beispiel exemplarisch für eine `if-then-else`-Anweisung und den korrespondierenden `?:`-Operator gezeigt:

```
if (B) P else Q        }  if (toRemote(B)) toRemote(P) else toRemote(Q)
B ? A1 : A2            }  toRemote(B) ? toRemote(A1) : toRemote(A2)
```

Es gibt jedoch auch einige Anweisungen und Ausdrücke, bei denen die Umsetzung nicht mehr so einfach abläuft, wie in den beiden zuvor gezeigten Beispielen. Dies betrifft vor allem die

Anwendung von Methoden und besonders der Zugriff auf Variablen (das müsste ja schon ansatzweise in 3.5 erkennbar gewesen sein). Auf solche nicht-trivialen Fällen, wird nun in den folgenden Unterabschnitten eingegangen.

3.6.2 Verwendung von `this`

Wird innerhalb einer nicht-statischen Methode das Schlüsselwort `this` verwendet, dann wird damit das Objekt bezeichnet, auf welches gerade die Methode angewendet wird. Betrachtet man nun eine lokale Klassendefinition für eine Klasse `B`, dann besitzt `this` den Typ `B`. Wird diese Klassendefinition transformiert, dann finden sich alle nicht-statischen Methoden in der Klasse `B_impl` wieder. `this` hat nun aber den Typ `B_impl` und nicht den erforderlichen Handle-Typ `B`. Benutzt man `this` als Qualifikation, dann spielt dies keine Rolle, da hiermit eben direkt auf eine Methode oder Variable aus `B_impl` zugegriffen wird, ohne den Umweg über ein Handle zu nehmen. Tritt `this` allerdings alleine auf, dann tritt ein ernsthafter Typkonflikt auf. `this` muss an dieser Stelle also geeignet in ein Handle auf `this` umgewandelt werden. Eine Transformationslösung wäre also, alleinstehende `this` Schlüsselwörter in „`new B(this)`“ (neues `B`-Handle anlegen) zu übersetzen. Dies wäre semantisch durchaus korrekt, würde aber, bei überwiegend lokaler Nutzung eines Objekts, unter Umständen sehr viele, nur einmal benötigte Handle-Objekte entstehen lassen.

Das hier vorgestellte System verwendet eine alternative Lösung: jedes *remote object* besitzt implizit eine Variable `_this`, die ein aktuelles Handle auf das Objekt selbst enthält. Beim Erzeugen eines *remote objects* oder unmittelbar nach einer Objekt-Migration wird das Handle automatisch angelegt. Es genügt damit, alleinestehende `this`-Schlüsselwörter einfach durch die `_this` Variable zu ersetzen.
Das folgende Programmfragment verdeutlicht nochmals, in welchen Fällen eine Transformation nötig ist, und wie diese aussieht:

```
{   ...
    B   obj;
    ...
    obj.method(this);
    this.method(obj);
    ...
}
```

```
{   ...
    B   obj;
    ...
    obj.method(_this);  // hier ist eine Transformation notwendig
    this.method(obj);   // hier nicht
    ...
}
```

3.6.3 Vergleich zweier Objekte

Ein weiteres Problem, das sich in Zusammenhang mit Handles ergibt, besteht beim Vergleich zweier Objekte über den `==`-Operator. Der Ausdruck „`obj1 == obj2`“ wertet nur dann zu `true` aus, wenn `obj1` und `obj2` das gleiche Objekt bezeichnen. Auf *remote objects* wird aber üblicherweise über Handles zugegriffen. Hier ist es fast der Normalfall, dass man unterschiedliche Handle-Objekte hat, die auf ein und dasselbe *remote object* verweisen. Vergleicht man nun zwei Handles über `==`, dann liefert der Ausdruck auch dann den Wert `false`, wenn beide Handles das gleiche *remote object* referenzieren. Man muss also anstelle eines Vergleichs zweier *remote objects* immer einen geeigneten anderen Ausdruck generieren, der dies berücksichtigt.
Nun ist es leider zur Übersetzungszeit im Allgemeinen nicht möglich festzustellen, ob eine Variable ein *remote object* oder ein lokales Objekt enthält. Dieses Problem tritt genau dann auf,

wenn die Variable vom Typ `java.lang.Object` ist. Deswegen müssen alle Vergleiche, bei denen auch nur die Möglichkeit besteht, dass eines der beteiligten Objekte ein *remote object* ist, folgendermaßen übersetzt werden:

```
/* obj1 oder obj2 bezeichnet ein remote
object oder ist vom Typ java.lang.Object */
obj1 == obj2
```

} `RemoteTools.equals(obj1,obj2)`

Die Methode `RemoteTools.equals` enthält eine Routine, die zur Laufzeit feststellt, um welche Objekt-Typen – remote oder lokal – es sich bei den beiden Argumenten handelt und ausgehend davon entweder einen lokalen Objektvergleich oder einen Referenzenvergleich der Handle-Referenzen vornimmt.

3.6.4 Methodenaufrufe

Motivation

Nach Abschnitt 3.4 zu urteilen, müsste ein Methodenaufruf im lokalen Fall genauso aussehen, wie bezüglich eines *remote objects* – dies ermöglicht schon, abgesehen von zusätzlich abzufangenden Exceptions, das Java RMI-System. Um zu zeigen, dass es dennoch Anlass gibt, sich über die Umsetzung von Methodenaufrufen Gedanken zu machen, soll folgendes Beispielprogramm demonstrieren:

```
class Alpha
{
    int i = 80273;
    Alpha id()
    {
        return this;
    }
}
remote class Beta
{
    Alpha  y;
    Alpha gety()
    {
        return y;
    }
    Alpha id(Alpha x)
    {
        return x;
    }
    void doSomething()
    {
        Alpha  a = new Alpha();
        Beta   b = this;
        y = a;
        if (a == a.id())  System.out.print(1);
        if (a == id(a))   System.out.print(2);
        if (a == b.id(a)) System.out.print(3);
        if (a == gety())  System.out.print(4);
        if (a == y)       System.out.print(5);
    }
}
```

Läßt man einmal das `remote`-Schlüsselwort der Klasse `Beta` außer Acht und ruft man die Methode `doSomething` auf, so kommt man leicht zu dem Ergebnis, dass als Ausgabe die Zahlenfolge „12345“ zu erwarten ist. Betrachtet man nun `Beta` als *remote class* und führt man

alle Methodenaufrufe bezüglich der Klasse `Beta` als entfernte Methodenaufrufe durch, dann erhält man, sicherlich wie erwartet, ein abweichendes Ergebnis. Beim ersten Methodenaufruf läuft alles lokal ab, weshalb hier noch keine Abweichungen auftreten. Bereits der zweite Aufruf erfolgt über RMI, was zur Folge hat, dass nicht `a`, sondern eine Kopie von `a` als Argument übergeben wird. Der anschließende Vergleich kann also nur zu `false` auswerten. Analog verhalten sich der dritte und der vierte Methodenaufruf. Die letzte Anweisung fällt erstaunlicherweise ebenso in diesen Diskussionsbereich, obwohl hier überhaupt keine Methode aufgerufen wird. Tatsächlich hängt nämlich das Ergebnis des Vergleichs davon ab, ob auf die Instanzvariable `y` über die Zugriffsfunktion und dann entfernt, oder direkt und lokale zugegriffen wird. Nimmt man einmal an, es wird auf `y` entfernt über die Zugriffsfunktion zugegriffen, dann ergibt sich als Gesamtausgabe „1“ – also nur der erste Fall ist äquivalent zur lokalen Ausführung.

Strikte Verwendung entfernter Methodenaufrufe bei *remote objects*

Wie man an dem Beispiel sieht, hat ein so einfaches Grundprinzip wie „Methodenaufrufe bezüglich *remote objects* erfolgen stets als entfernte Aufrufe – und damit auch mit deren Eigenheiten“ zur Folge, dass die entfernte Objekt-Semantik, mit der lokalen praktisch kaum mehr etwas gemeinsam hat. Die Idee, die man hinter einem solchen Prinzip sehen kann, zielt auf eine uniforme Behandlung von Methodenaufrufen ab und hat folglich natürlich auch zur Konsequenz, dass Variablenzugriffe einheitlich gehandhabt werden. Dies bedeutet konkret, dass Variablenzugriffe ebenfalls über entfernte Aufrufe von Zugriffsfunktionen abgewickelt werden. Legt man eine solch einfache Semantik einer verteilten Java Umgebung zu Grunde, dann muss man allerdings auch folgende Nachteile in Kauf nehmen:

- Entfernte Methodenaufrufe sind prinzipiell teuer, insbesonders, wenn komplizierte Datenstrukturen als Argumente übergeben werden.
- Entfernte Methodenaufrufe werden, bedingt durch das Java RMI-System, stets in einem anderen Thread ausgeführt als der aufrufende Thread. Eine sinnvolle Synchronisation über die `synchronized`-Anweisung oder `synchronized`-Modifikatoren ist damit komplett ausgeschlossen (näheres hierzu, siehe übernächsten Unterabschnitt).
- Entfernte Aufrufe benutzen *Object Serialization* zur Übergabe der Argumente und zur Rückgabe von Ergebnissen. Es sind also nur serialisierbare Objekte als Argumente zugelassen. Insbesondere ist es nicht mehr sinnvoll möglich, mit systemnahen und damit nicht serialisierbaren Objekte auch nur Objekt-intern umzugehen.
- Weitaus restriktivere Konsequenzen ergeben sich für die Verwendung von Instanzvariablen:
  - Es können nur serialisierbare Objekte Instanzvariablen zugewiesen werden. Es ist damit nicht mehr möglich, einen großen Teil der Java Klassenbibliothek zu verwenden.
  - Bei der Zuweisung eines lokalen Objekts an eine Instanzvariable, wird stets eine Kopie und nicht das ursprüngliche Objekt zugewiesen.

Differenzierter Umgang mit entfernten Methodenaufrufen

Besonders der letzte Punkt macht das Prinzip, Methodenaufrufe bezüglich entfernter Objekte stets als entfernten Methodenaufruf durchzuführen, für die Praxis unakzeptabel. Der Zugriff auf lokale Objekte – der im verteilten System sowieso schon auf die lokale virtuelle Maschine

beschränkt ist – wird noch weiter auf den Gültigkeitsbereich eines einzigen Methodenrumpfes eingeschränkt (nämlich den Rumpf, in dem das Objekt erzeugt wird). Lokale Objekte werden von diesem Block nach Außen nur als Kopien weitergegeben. Ein ernstzunehmender Einsatz lokaler Objekte wird damit endgültig zunichte gemacht. Aus diesen Gründen wird im hier vorgestellten System ein anderer Ansatz verfolgt:

1. Nicht-statische Methodenaufrufe finden bezüglich entfernter Objekte genau dann lokal statt, wenn der Aufruf entweder unqualifiziert oder bezüglich `this` bzw. `super` qualifiziert erfolgt; d.h. wenn zur Übersetzungszeit erkennbar ist, dass das den Aufruf initiierende Objekt und das gerufene Objekt identisch sind.
2. Alle übrigen qualifizierten (nicht-statischen) Methodenaufrufe bezüglich eines *remote objects* werden als entfernte Methodenaufrufe durchgeführt.
3. Statische Methodenaufrufe, die aus einer nicht-statischen Methode heraus erfolgen, werden stets als entfernte Aufrufe behandelt.
4. Statische Methodenaufrufe, die in einer anderen statischen Methode gemacht werden, welche nicht zur gleichen Klasse gehört, finden immer entfernt statt. Methoden einer Oberklasse zählen in diesem Zusammenhang nicht zur gleichen Klasse.
5. Genau dann, wenn der Aufruf einer statischen Methode in einer anderen statischen Methode der gleichen Klasse erfolgt, wird ein lokaler Methodenaufruf durchgeführt.

Diese Regelung scheint zwar auf den ersten Blick relativ kompliziert zu sein, in der Praxis ergeben sich aber im Umgang mit Methoden diesbezüglich wenig Probleme. Das Konzept orientiert sich nämlich an der intuitiven Vorstellung eines Programmierers, dass Methodenaufrufe die das gleiche Objekt betreffen, wie im lokalen Fall abgewickelt werden (da ein solcher Aufruf ja nur das gleiche Objekt und nicht möglicherweise entfernt liegende Objekte betrifft) und dass für Aufrufe bezüglich eines anderen *remote objects* stets das RMI Verwendung finden muss, da sich das andere Objekt ja möglicherweise auf einer anderen virtuellen Maschine befinden kann (aber natürlich nicht befinden muss). Dass man an der Syntax eines Methodenaufrufs erkennen kann, ob ein entfernter oder ein lokaler Aufruf stattfindet, ist hier sicherlich auch sehr dienlich. Für statische Methoden ist dieser Vorteil, an der Syntax die Form des Aufrufs erkennen zu können, leider nicht gegeben. Hat man aber eine Vorstellung vom Umgang mit Klassen in der verteilten Java Umgebung, dann wird man als Benutzer auch die Semantik eines statischen Methodenaufrufs intuitiv richtig verstehen.

Der entscheidende Vorzug dieses Konzepts besteht nun aber darin, dass alle oben genannten Nachteile, des ursprünglichen Prinzips für Methodenaufrufe, hier nun nicht mehr bestehen. Man kann insbesonders entfernte Klassen, die eine bestimmte Datenstruktur implementieren und von anderen Klassen unabhängig sind, fast analog zu lokalen Klassen realisieren.

3.6.5 Zugriffe auf Variablen

Direkter Zugriff vs. Zugriff über Zugriffsmethoden

Im vorhergehenden Abschnitt wurde die Zugriffsstrategie für Variablen bereits implizit mit abgehandelt. Sie orientiert sich am Konzept für die Durchführung von Methodenaufrufen, mit dem Unterschied, dass hier bei der Umsetzung zwischen dem Zugriff über Zugriffsmethoden

oder einem direkten Variablenzugriff entschieden werden muss. Übertragen auf Variablen, sehen die Prinzipien folgendermaßen aus:

1. Wird von einem Objekt (aus einer nicht-statischen Methode heraus) unqualifiziert, oder über `this` bzw. `super` qualifiziert, auf eine nicht-statische Variable zugegriffen, so erfolgt dies direkt unter Umgehung von Zugriffsmethoden.
2. Bei allen weiteren qualifizierten Zugriffen auf nicht-statische Variablen werden Zugriffsmethoden eingesetzt.
3. Wird von einer nicht-statischen Methode aus, eine statische Variable angesprochen, dann werden hierfür prinzipiell Zugriffsmethoden eingesetzt.
4. Wird von einer statischen Methode auf eine statische Variable einer anderen Klasse (hierzu zählen auch mögliche Oberklassen) zugegriffen, dann geschieht dies über eine Zugriffsmethode.
5. Auf statische Variablen, die von einer statischen Methode der gleichen Klasse angesprochen werden, wird immer direkt zugegriffen.

Es stellt sich hier möglicherweise die Frage, warum man diese Diskussion für Variablenzugriffe überhaupt führen muss. Man muss sich nämlich auch beim Lesen oder Schreiben von Variablen stets im Klaren sein, ob eine Zugriffsfunktion eingesetzt wird oder nicht. Im ersten Fall sind lokale Objekte wieder mit einer Werte-Semantik behaftet: man erhält beim lesenden Zugriff eine Kopie der Variable bzw. bei schreibendem Zugriff wird der Wert der Variable auf eine Kopie des Objekts gesetzt. Nur bei Basistypen wie `boolean`, `byte`, `char`, `int` usw. muss man sich diesbezüglich keine Gedanken machen.

Sonderfälle

Neben der Frage nach der Zugriffsart, ergeben sich für schreibende als auch lesende Variablenzugriffe noch eine Reihe von weiteren Problemen, die meist in Zusammenhang mit der Vielzahl an Zuweisungsoperatoren oder allgemein mit Arrays oder schlimmstenfalls in einer Kombination aus beidem, stehen. In Abschnitt 3.5 wurde diese Problematik ausführlich diskutiert. Zusammenfassend soll an dieser Stelle lediglich ein Beispiel die damit zusammenhängenden Fragestellungen beim Umsetzen eines Variablenzugriffs deutlich machen. Eine Diskussion, in welchem Fall, wie mit einer Variable umgegangen wird, würde den Rahmen dieser Arbeit sprengen, da die Entscheidungskriterien teilweise äußerst kompliziert sind. Für Details sei daher auf den Quelltext des entwickelten Übersetzers verwiesen (siehe dazu 5).
Aus Gründen der Übersichtlichkeit, folgt nun zuerst das ursprüngliche Codefragment des betrachteten Beispielprogramms:

```
remote class ArrayDemo
{
    static Object[] objArr;
    int[][]         intArr;

    public void nonsense()
    {
        int[]       i = {11,22,33,44};
        ArrayDemo   a = new ArrayDemo();

        i[3] -= 4;
        intArr = new int[2][];
```

```
        intArr[0] = i;
        a.intArr = intArr;
        a.intArr[1] = a.intArr[0];

        objArr = intArr;
        ((int[][])objArr)[0][0]++;
        objArr[1] = objArr[0];
    }
}
```

Von den transformierten Klassen ist eigentlich nur die Klasse `ArrayDemo_impl` von Interesse. Sie enthält die Implementation der Methode `nonsense`. Wie man am Quellcode der Klasse unschwer erkennen kann, wird der Übersetzer bei solchen Spielereien mit `java.lang.Object`- und Array-Typen sehr gefordert. Wie eine Zuweisung oder ein lesender Zugriff in den einzelnen Fällen ausfallen muss, kann, bei Verständnisproblemen nochmals in 3.5 nachgelesen werden.

```
public class ArrayDemo_impl extends RemoteInstance implements ArrayDemo_intf
{
    /* objArr ist statisch und in der Klasse ArrayDemo_class_impl deklariert */
    int[][]         intArr;
    ...
    public void nonsense() throws RemoteException, MovedException
    {
        _enter();
        try
        {
            int[]            i = {11, 22, 33, 44};
            ArrayDemo        a = new ArrayDemo();

        /* es folgen lokale, direkte Zugriffe */
            i[3] -= 4;
            intArr = new int[2][];
            intArr[0] = i;

        /* hier wird auf Instanzvariablen eines anderen Objekts zugegriffen */
            a._set_ArrayDemo_intArr(intArr);
            a._set_ArrayDemo_intArr(a._get_ArrayDemo_intArr(0),1);

        /* jetzt folgen Zugriffe auf die statische Variable, wobei hier erschwerend noch
           der Typ Object[] hinzukommt. In diese Variable wird ein int[][]-Array kopiert! */
            ArrayDemo._set_ArrayDemo_objArr(intArr);
            ((Integer)ArrayDemo._set_ArrayDemo_objArr(
                        RemoteTools.wrap(((int[][])ArrayDemo._get_ArrayDemo_objArr())[0][0] + 1),
                        new ArrayIndex(0,0))).intValue();
            (java.lang.Object)ArrayDemo._set_ArrayDemo_objArr(
                        ArrayDemo._get_ArrayDemo_objArr()[0],
                        new ArrayIndex(1));
        }
        finally
        {
            synchronized (LocalJVM.lock) { _calledMethods--; }
        }
    }
    ...
}
```

3.6.6 Synchronisationsanweisungen

Grundlagen zur Synchronisation in Java

Wie bereits zu Beginn dieses Kapitels angesprochen wurde, bereiten die lokalen Synchronisationsmechanismen Javas in einer verteilten Umgebung große Schwierigkeiten. Da man

zum Verständnis der Problematik das Synchronisationskonzept Javas gut kennen muss, folgt zunächst eine kurze Erläuterung hierzu.

Java unterstützt durch sein Thread-Konzept die Entwicklung nebenläufiger Anwendungen. Um ein deterministisches Verhalten zu ermöglichen und Inkonsistenzen zu vermeiden, bietet Java ein Prinzip zur *Synchronisation* der Aktivitäten nebenläufiger Threads. Hierzu werden in Java *Monitore* eingesetzt. Monitore stellen einen Mechanismus dar, der verhindert, dass sich mehrere Threads gleichzeitig in einem kritischen Bereich aufhalten. Sie werden in Java durch *Locks* realisiert. Jedes Objekt ist dabei mit genau einem Lock assoziiert. Ein kritischer Bereich lässt sich auf zwei verschiedene Arten schützen:

- Man versieht eine Methode mit dem `synchronized`-Modifikator. Damit wird der gesamte Methodenrumpf als kritischer Bereich geschützt. Bevor ein Thread den Rumpf der Methode betreten darf, muss dieser erst das Lock des betreffenden Objekts bzw. des Klassenobjekts, wenn es sich um eine statische Methode handelt, erwerben. Dies ist möglich, wenn zur Zeit kein anderer Thread das Lock besitzt. Andernfalls muss der aktuelle Thread so lange warten, bis das Lock wieder verfügbar ist. Am Ende des Methodenrumpfes wird das Lock wieder freigegeben.

- Man fordert das Lock eines ganz bestimmten Objektes explizit an, indem man einen kritischen Bereich in eine `synchronized`-Anweisung einbettet. Hier erwirbt der aktuelle Thread vor der Ausführung des Rumpfes zunächst das Lock für das angegebene Objekt (was wiederum mit einem Warten auf die Freigabe des Locks durch einen anderen Thread verbunden sein kann). Nach der Ausführung des `synchronized`-Rumpfes wird das Lock vom ausführenden Thread wieder freigegeben.

Die Methoden `wait`, `notify` und `notifyAll` der Klasse `java.lang.Object` unterstützen zur Thread-Koordination die Übertragung eines Kontrollflusses von einem Thread zu einem anderen. Ein Thread kann durch einen Aufruf von `wait` sich selbst suspendieren, wobei implizit das Lock des Objekts, bezüglich dem die `wait`-Methode aufgerufen wurde, freigegeben und der Thread in das *Wait-Set* dieses Objekts eingetragen wird. Sobald ein anderer Thread die Methode `notify` bzw. `notifyAll` bezüglich dieses Objekts aufruft, wird ein Thread bzw. werden alle Threads des Wait-Sets wieder zum Leben erweckt; d.h. sie bewerben sich erneut um die Zuteilung des Locks dieses Objekts.

Synchronisation in einem verteilten System

In einem verteilten System hat man nun aber Threads auf verschiedenen virtuellen Maschinen laufen. Threads sind dabei stets bezüglich einer Maschine lokal; d.h. sie können niemals einen Adressraum verlassen. Mit den *remote threads* aus Abschnitt 4.2.5 wird lediglich ein Mechanismus zur Verfügung gestellt, welcher es erlaubt, von einer lokalen virtuellen Maschine aus, auf die Threads anderer Maschinen zuzugreifen bzw. auf einer anderen Maschine einen Thread zu starten. Es wird sich noch herausstellen, dass die Hauptursache des Synchronisationsproblems innerhalb einer verteilten Java Umgebung genau darin besteht, dass es keine maschinenübergreifenden *Aktivitätsstränge* gibt. Nur diese würden ein analoges Synchronisationskonzept für ein verteiltes Java zulassen.

Zunächst wird für die zuvor angeführten Synchronisationsmöglichkeiten geklärt, warum diese in einer verteilten Umgebung im Allgemeinen versagen:

- `synchronized`-Anweisungen beziehen sich auf das Handle und nicht, wie gewünscht, auf das *remote object* selbst. Da es selbst innerhalb einer virtuellen Maschine gewöhnlicherweise pro *remote object* mehrere Handle-Objekte gibt, ist eine solche Synchronisation zwecklos.

- Der `synchronized`-Modifikator einer Methode führt korrekterweise zu einer Synchronisation eines *remote objects* – nennen wir es einmal A. Sobald das *Lock* von A verfügbar ist, wird es dem Thread zugewiesen, der die `synchronized`-Methode ausführt. Dieser Thread sei nun mit T bezeichnet. Im lokalen Fall werden alle, innerhalb dieser Methode direkt aufgerufenen Methoden in genau diesem Thread T, der ja das Lock von A besitzt, ausgeführt. Es kommt bei anderen `synchronized`-Methoden oder -Anweisungen bezüglich von A, die infolgedessen ausgeführt werden, also niemals zu der Situation, dass sie warten müssen.
  Im verteilten System sorgt das RMI aber dafür, dass entfernte Methoden stets in einem neuen Thread, möglicherweise sogar auf einer anderen Maschine, abgearbeitet werden. Findet nun infolgedessen ein entfernter Methodenaufruf einer `synchronized`-Methode von A statt, so erhält der von RMI neu gestartete, ausführende Thread aber nicht das Lock, da dieses ja der ansässige Thread T besitzt. Es kommt zu einem Dead-Lock, das es im lokalen Java in dieser Situation niemals gegeben hätte, da hier eben alle Methodenaufrufe innerhalb eines Threads, der im Besitz des nötigen Locks ist, ausgeführt worden wären. Hier ist ein Beispiel für eine ähnliche Situation:

  ```
  remote class Gamma
  {
      synchronized void method1(boolean b)
      {
          if (b)
              (new Gamma()).method2(this);
      }
      synchronized void method2(Gamma x)
      {
          x.method1(false);
      }
  }
  ```

  Führt man nun den Methodenaufruf „`(new Gamma()).method1(true)`“ aus, dann führt dies, bei dem Versuch der Durchführung des Methodenaufrufs „`x.method1(false)`“ aus der Methode `method2` zu einer Verklemmung. Der ursprüngliche Thread, in dem der initiierende Aufruf „`(new Gamma()).method1(true)`“ stattfand, besitzt nämlich das Lock, das auch von dem Thread benötigt wird, welcher den Aufruf „`x.method1(false)`“ bearbeitet.

- Die Methoden `wait`, `notify` und `notifyAll` beziehen sich, falls sie auf ein *remote object* angewendet werden, auf dessen Handle und nicht wie gewünscht auf das *remote object* selbst.

Trotz dieser Problematik kann der Einsatz von `synchronized`-Methoden durchaus zweckmäßig sein,[9] wenn man bei dem Entwurf einer solchen Klasse sich bewußt ist, in welchen Situationen es zu Konflikten kommen kann. Folgende Regeln charakterisieren die Fälle, in denen keine Probleme für eine `synchronized`-Methode zu erwarten sind:

[9] Ein Beispiel hierfür sind die Signalobjekt- und die Monitor-Klasse der Klassenbibliothek des in 4 beschriebenen Laufzeitsystems.

- Unproblematisch ist jeglicher Umgang mit lokalen Objekten im Methodenrumpf.
- Zugriffe auf Variablen von beliebigen *remote objects* ist uneingeschränkt möglich.
- Werden nur Methoden bezüglich des eigenen Objekts (`this`) aufgerufen, so sind keine Konflikte zu erwarten.
- Enfernte Methodenaufrufe bezüglich eines anderen Objekts sind unkritisch, solange niemals die Situation auftritt, dass im Laufe deren Auswertung eine `synchronized`-Methode des ursprünglichen Objekts aufgerufen wird. Dann kommt es nämlich mit Sicherheit zu einer Verklemmung. Das vorige Beispielprogramm zeigt einen Fall, bei dem genau diese Regel nicht eingehalten wird.

Ein sinnvoller Einsatz der Synchronisationsmechanismen Javas ist oftmals nur mit den Methoden `wait`, `notify` und `notifyAll` möglich. Diese sind jedoch so, wie sie in `java.lang.Object` definiert wurden, für *remote objects* unbrauchbar (siehe obige Aufzählung). Da die Methoden als `final` deklariert werden, ist ein Überschreiben, zur Anpassung an *remote objects* nicht möglich. Deshalb muss der Übersetzer Aufrufe dieser Methoden bezüglich eines anderen *remote objects* (unqualifizierte Aufrufe von `wait`, `notify` und `notifyAll` arbeiten korrekt) durch eigenen Code ersetzen. Die Aufrufe werden konkret folgendermaßen transformiert:

```
obj.wait()              }   RemoteTools.doWait(obj)
obj.wait(1000)          }   RemoteTools.doWait(obj,1000)
obj.wait(1000,80273)    }   RemoteTools.doWait(obj,1000,80273)
obj.notify()            }   RemoteTools.doNotify(obj)
obj.notifyAll()         }   RemoteTools.doNotifyAll(obj)
```

Für einfache Methoden sind die obigen Regeln zur Verwendung des `synchronized`-Modifikators leicht nachzuprüfen. Bei komplizierteren Synchronisationsanforderungen empfielt es sich innerhalb eines verteilten Systems explizit auf entfernte Synchronisationsobjekte zurückzugreifen. Das hier vorgestellte Laufzeitsystem stellt eine Klasse für Signalobjekte und eine Monitor-Klasse zur Verfügung. Mit der ersten Klassen lassen sich vor allem Threads einfach synchronisieren, wohingegen die zweite Klasse hauptsächlich zur Realisierung von gegenseitigen Ausschlüssen gedacht ist.

Transparenter Synchronisationsmechanismus für ein verteiltes Java

Ein zum Synchronisationskonzept des lokalen Java analoges verteiltes Synchronisationskonzept kann nur dann realisiert werden, wenn die Aktivitätsstränge bei einer lokalen Ausführung eines Java-Programms durch eine verteilte Laufzeitumgebung simuliert werden. Hierzu müsste bei jedem entfernten Methodenaufruf eine Kennung als Parameter mitgegeben werden, welche eindeutig den aufrufenden Thread kennzeichnet. Über diese Kennung könnte dann lokal entschieden werden, ob der aufrufende Thread das (globale) Lock für ein synchronisiertes *remote object* besitzt. Dieser Idee liegt konzeptionell die Vorstellung von maschinenübergreifenden Threads zugrunde, die auf verschiedene lokale Threads abgebildet werden, deren „Aktivitätszeiten“ sich jedoch niemals überlappen. Eine Realisierung dieses Konzepts ist technisch sicherlich möglich, wenngleich dies auch äußerst aufwendig werden würde.

3.7 Modifikatoren

3.7.1 Modifikatoren von Variablen und Methoden

In den beiden letzten Abschnitten über die Umsetzung von Methoden- und Variablendeklarationen wurde noch kein Wort über deren Modifikatoren verloren. Prinzipiell muss man bei Modifikatoren zwischen Modifikatoren, die den Zugriff auf ein Attribut regeln (`private`, `protected`, `public`) und Modifikatoren, die ein bestimmtes Verhalten beschreiben, unterscheiden.

`static` dient, sowohl bei Variablen als auch Methoden, als Unterscheidungskriterium dafür, ob eine Deklaration in den Instanzenanteil oder statischen Anteil einer Klasse gehört. Abgesehen vom Handle, taucht `static` in keiner der weiteren vier generierten Klassen mehr auf.

Bei Variablendeklarationen ist klar, dass alle Modifikatoren außer `static` aus der ursprünglichen Definition, für die Deklaration in der Realisierung des Instanzen- oder statischen Klassenanteils übernommen werden können. Womit nur noch zu überlegen wäre, wie mit Modifikatoren von Methoden und Zugriffsfunktionen umgegangen werden muss. Tabelle 1 zeigt, wie ursprüngliche Modifikatoren von Methoden innerhalb der Handle und den beiden `_impl`-Klassen eingesetzt werden. Ist in der Tabelle an einer Stelle nichts eingetragen, so bedeutet dies, dass dieser Modifikator nicht übernommen wird, ein Strich bedeutet, dass diese Methode in der betreffenden Klasse überhaupt nicht vorhanden ist.

Modifikator	**Handle**	**Remote Impl.**
`public`	`public`	`public`
`protected`		`public`
`private`		`public`
`abstract`	`abstract`	—
`static`	`static`	
`final`	`final`	`final`
`synchronized`		`synchronized`
`native`	`native`	—

Tabelle 1: Umsetzung von Methoden-Modifikatoren

Am erstaunlichsten an der Tabelle sind sicherlich die Umsetzungsregeln für die Zugriffsmodifikatoren. In den beiden `_impl`-Klassen sind alle Methoden `public`. Dies liegt daran, dass alle diese Methoden auch in den dazugehörigen Interfaces definiert werden. Ist eine Methode in einem Interface aufgeführt, so ist sie automatisch `public`, mit der Konsequenz, dass die Implementation dieser Methode ebenfalls `public` sein muss. Setzt man das Java RMI ein, so gehen deswegen alle Zugriffsmodifikatoren von Methoden verloren.
Dass nun auch im Handle, das ja kein entferntes Objekt im Sinne von RMI darstellt, `private`- und `protected`-Modifikatoren wegfallen, hat einen anderen Grund: die Aufsplittung einer einzigen Klasse in einen Instanzen- und einen statischen Anteil. Ist beispielsweise eine statische Methode `private` deklariert, so muss sie auch in der Instanzenklasse aufgerufen werden können. Die Methoden im statischen Klassenanteil können also auch in der Handle nicht `private` deklariert werden. Umgekehrt dürfen ursprünglich `private` oder `protected` deklarierte nicht-statische Methoden in der Handle ebenfalls nicht mit den Modifikatoren `private` bzw. `protected` versehen werden, da sie ja auch möglicherweise aus einer statischen Methode, und damit aus einer Methode außerhalb der, den Instanzenanteil implementierenden Klasse, heraus aufgerufen werden könnten.
Diese Problematik ist auch der Grund dafür, dass für Zugriffsmethoden auf Variablen die Modifikatoren `protected` und `private` nicht zulässig sind.

Der Grund für das Wegfallen des `synchronized`-Modifikators bei Handle-Methoden, hängt mit dem in 3.6.6 beschriebenen Synchronisationsproblem zusammen. Die eigentliche Ursache für das Wegfallen des `synchronized`-Modifikators liegt in der, an das verteilte System speziell angepassten `wait`-Methode. Deren Aufruf in einer `synchronized` Methode würde das Lock für das Implementationsobjekt zwar wieder temporär freigeben, aber das Lock für das Handle-Objekt würde weiter belegt bleiben. Es bestünde dann die Gefahr eines Dead-Locks. Da der `synchronized`-Modifikator sowieso keine unmittelbare Bedeutung für das Handle-Objekt besitzt, kann er auch ohne Gefahr einfach weggelassen werden.

Zusammenfassend kann man feststellen, dass bei der Übersetzung einer als `remote` deklarierten Klasse, auf die meisten der Modifikatoren aus technischen Gründen und aus Architekturgründen, verzichtet werden muss. Der Übersetzer überprüft jedoch selbstverständlich, dass innerhalb der ursprünglichen Klasse auch alle Zugriffsgrundsätze Javas korrekt eingehalten werden. Dennoch muss man sich klar sein, dass die generierten Klassen, schon alleine durch das RMI, nicht mehr sicher sind und nicht verhindert werden kann, dass auch `private`-Variablen, ohne weiteres über Zugriffsfunktionen ausgelesen werden können.

3.7.2 Klassenmodifikatoren

Die Vergabe der Klassenmodifikatoren geschieht analog zur ursprünglichen Klasse; d.h. das Handle einer *remote class* bekommt genau die gleichen Modifikatoren, wie die zu übersetzende `remote`-deklarierte Klasse. Für alle restlichen generierten Klassen und Interfaces muss es möglich sein, dass von beliebigen Packages aus auf sie zugegriffen wird – so z.B. auch von dem JDR[10]-Package, das die hier beschriebene verteilte Laufzeitumgebung realisiert. Dies wird man im nächsten Kapitel erkennen. Diese Klassen müssen deshalb mit dem `public`-Modifikator versehen werden.

[10] *Java Distributed Runtime*

4 Verteilte Laufzeitumgebung

4.1 Anforderungen

Die Laufzeitumgebung muss dafür sorgen, dass Vorgänge im verteilten System, das aus mehreren lokalen virtuellen Maschinen mit getrennten Adressräumen besteht, genau so ablaufen, wie im nicht-verteilten Fall auf einer einzigen JVM. Wie in Abschnitt 2 bereits erläutert wurde, war eine Grundvoraussetzung hierfür, dass Klassen anders übersetzt werden, um wenigstens annähernd eine äquivalente Objekt-Semantik von *remote objects* im Vergleich zu normalen lokalen Java Objekten zu ermöglichen.

Basierend auf dieser Grundlage, müssen zur Laufzeit folgende Aufgaben erledigt werden:

- Entfernte Methodenaufrufe müssen durchgeführt und bei Fehlern entsprechende Fehlerbehandlungsmaßnahmen ergriffen werden.
- Ein verteilter Gabage-Collector muss an die Stelle der lokalen Garbage-Collectoren treten. Ein *remote object* darf erst gelöscht werden, wenn es im gesamten verteilten System nicht mehr referenziert wird.
- *remote classes* müssen bei erstmaligem Zugriff auf genau einer lokalen JVM initialisiert werden. Alle zum System gehörigen JVMs müssen bei statischen Methodenaufrufen bzw. Zugriffen auf statische Variablen bezüglich dieser Klasse dann auf das dort angelegte Klassenobjekt zugreifen.
- Bei einer Objekt-Instantiierung eines *remote objects* muss eine lokale JVM ausgewählt und dort das neue Objekt angelegt werden. Hierfür muss auf dieser Maschine ein Konstruktorobjekt für die betreffende Klasse verfügbar sein oder, falls noch nicht vorhanden, ein neues Konstruktorobjekt angelegt werden.
- Ein *remote object* muss von seinem aktuellen Adressraum in den Adressraum einer anderen virtuellen Maschine umziehen können.
- Um die echte Nebenläufigkeit, die ein verteiltes System anbietet, nutzen zu können, müssen Threads zur Verfügung gestellt werden, die von einer beliebigen Maschine aus auf einer anderen beliebigen lokalen JVM eingerichtet werden können. Zugriffe auf diese sogenannten *remote threads* muss von allen lokalen virtuellen Maschinen aus jederzeit möglich sein.

Die ersten beiden Punkte werden bereits durch das Java RMI-System abgedeckt. Bestenfalls die Fehlerbehandlung bei fehlgeschlagenen entfernten Aufrufen ist zu realisieren. Der DGC-Algorithmus von RMI [RMI_96] ist genau so implementiert, wie er in dem hier vorgestellten System benötigt wird.

Neben der Erledigung der grundlegenden Aufgaben, werden an die Laufzeitumgebung auch noch folgende Leistungsanforderungen gestellt:

- Die verteilte Umgebung muss frei skalierbar sein. Es müssen beliebig viele lokale JVMs zum System hinzugefügt werden können. Dies sollte auch noch möglich sein, während ein Programm gerade abgearbeitet wird. Die lokalen JVMs des Systems können entweder alle auf einem Rechner oder verteilt auf mehreren Rechnern laufen. Das Entfernen von

virtuellen Maschinen aus dem System dagegen ist problematisch, da dies nur dann möglich ist, wenn keine Objekte dort von außerhalb referenziert werden. Dies ist jedoch praktisch niemals der Fall.

- Die Rate an entfernten Methodenaufrufen, die der Synchronisation aller lokalen JVMs oder anderer administrativer Zwecke dienen, soll möglichst gering gehalten werden. Demzufolge ist es notwendig, Verwaltungsinformationen redundant auf mehreren virtuellen Maschinen zu halten.

Wie man sehen wird, sind es diese beiden zuletzt angeführten Anforderungen an eine verteilte Java Laufzeitumgebung, die wesentlich die Architektur der Umgebung bestimmen.

4.2 Architektur der Laufzeitumgebung

4.2.1 Prinzipielle Struktur

Die hier vorgestellte Java Laufzeitumgebung JDR (*java distributed runtime*) besitzt eine zentrale Komponente, als *Runtime Manager* bezeichnet, die alle für das verteilte System notwendigen Informationen verwaltet. Dieser Runtime Manager muss beim Start des Systems als erstes auf einem vordefinierten Rechner gestartet werden. Er ist als Kern von JDR die einzige Komponente deren Ausführungsort vordefiniert ist.[11] Lokale JVMs können anschließend auf beliebigen Rechnern gestartet werden. Sie registrieren sich bei ihrem Start automatisch beim Runtime Manager. Dieser führt eine Liste aller lokalen JVMs des Systems. Lokale JVMs haben nur die Aufgabe, auf ihrer Maschine Klassen- und Konstruktorobjekte anzulegen, sowie *remote objects* auf ihre Maschine umziehen zu lassen. Sie besitzen keine administrativen Aufgaben. Zugriffe auf die verteilte Umgebung sind in jeder virtuellen Maschine über eine Klasse `RuntimeEnvironment` möglich. Diese Klasse enthält eine statische Schnittstelle zur verteilten Umgebung. Es gibt insbesonders keine Objekte dieser Klasse, womit diese auch nicht von außen, z.B. über RMI, angesprochen werden kann. Wird das erste Mal von einer virtuellen Maschine auf die verteilte Umgebung zugegriffen, so besorgt sich das lokale `RuntimeEnvironment` automatisch vom Runtime Manager eine Kopie aller Verwaltungsdaten. Ab sofort laufen alle Zugriffe auf die Umgebung ohne Kontaktierung des Runtime Managers ab. Erst wenn einmal festgestellt wird, dass Daten fehlen, so werden diese vom Runtime Manager besorgt. Die Synchronisation zwischen Runtime Manager und den lokalen `RuntimeEnvironment`-Klassen läuft also lazy ab. Kommt es vor, dass der Runtime Manager Daten erhält, die für alle lokalen RuntimeEnvironments von Interesse sind – z.B. wenn eine neue lokale JVM registriert wurde – so kontaktiert er alle lokalen JVMs, die ihrerseits ihren RuntimeEnvironments mitteilen, dass ihre Verwaltungsdaten veraltet sind. Beim nächsten Zugriff von einer virtuellen Maschine auf die verteilte Umgebung, werden dann vom Runtime Manager automatisch die aktuellen Daten kopiert.

Abbildung 6 gibt einen Überblick über die Abhängigkeiten der einzelnen Komponenten des JDR-Systems. Ausserdem werden bereits die Aufgabenbereiche der unterschiedlichen Systemkomponenten näher umrissen. Diese werden in den folgenden Abschnitten noch ausführlich diskutiert.

Mit dieser Struktur lassen sich die Leistungsanforderungen aus 4.1 erfüllen. Die Aufgaben, die noch nicht von RMI übernommen werden, müssen vom Runtime Manager in Zusammenarbeit

[11] In 4.3 wird erläutert, wie man den Ausführungsort des Runtime Managers im JDR-System festlegt, damit er auch lokalen JVMs bekannt ist, die sich bei dem Runtime Manager registrieren lassen möchten.

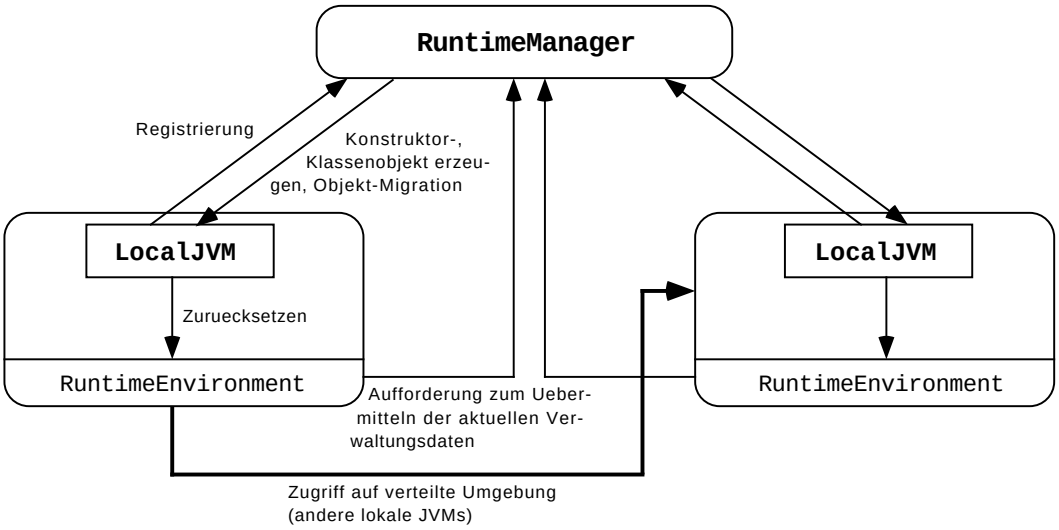


Abbildung 6: Interaktion der JDR-Komponenten

mit den lokalen JVMs erledigt werden. Hierbei handelt es sich lediglich um die Instantiierung von *remote objects* auf den verschiedenen lokalen virtuellen Maschinen, den Zugriff auf statische Methoden und Variablen, die gesamte Objekt-Migration, sowie der Umgang mit *remote threads*.

4.2.2 Runtime Manager

Verwaltung der Laufzeitumgebung

Bei dem Runtime Manager handelt es sich um ein Objekt, mit welchem über RMI kommuniziert werden kann. Dieses Objekt wird auf einer festgelegten Maschine instantiiert und ist dort über einen vordefinierten Port erreichbar. In der Terminologie von RMI handelt es sich beim Runtime Manager um eine Registrierung. Allerdings hat diese Registrierung nichts mehr mit der von RMI zur Verfügung gestellten Registrierung zu tun. Diese bietet nur die Möglichkeit, entfernte Objekte, die sich auf dem gleichen Rechner wie die Registrierung befinden, unter einem Namen abzulegen bzw. über einen Namen zu suchen oder zu löschen. Dies ist jedoch lange nicht für die Erfüllung der geforderten Funktionalität des Runtime Managers ausreichend. Bei diesem müssen sich ja möglicherweise lokale JVMs von anderen Rechnern registrieren lassen, neben virtuellen Maschinen müssen auch Konstruktor- und Klassenobjekte registriert werden und er muss außerdem aktiv mit lokalen JVMs kommunizieren können.
Intern müssen im Runtime Manager also folgende Datenstrukturen gehalten werden:

- eine Liste aller zur verteilten Umgebung gehörigen lokalen JVMs. Diese werden wiederum durch ein entferntes Objekt repräsentiert. Somit ist eine aktive Kommunikation zwischen beiden Komponenten sowohl vom Runtime Manager als auch von den lokalen JVMs aus möglich,
- eine Klassenobjekttabelle, die zu jeder initialisierten Klasse einen Stub auf das dazugehörige Klassenobjekt gespeichert hat und
- eine Konstruktorobjekttabelle, die für bekannte Klassen ein Feld von Konstruktorobjekt-Stubs enthält. Die Felder sind jeweils so groß, dass es für jede lokale JVM einen Eintrag gibt.

Redundante Datenspeicherung

Alle Tabellen enthalten anfangs keine Einträge. Diese werden erst bei konkretem Bedarf zur Laufzeit eingefügt. Da diese drei Datenstrukturen auch aus Effizienzgründen lokal innerhalb der jeweiligen `RuntimeEnvironment`-Klassen gehalten werden und auch nur von diesen aus selbst Einträge initiiert werden, muss dafür gesorgt werden, dass es zwischen den einzelnen Tabellen nicht zu Inkonsistenzen kommt. Folgende Grundsätze beschreiben ein geeignetes Synchronisationsprotokoll:

1. Die Tabellen des Runtime Managers enthalten zu jedem Zeitpunkt stets vollständig alle aktuellen Einträge.

2. Die Tabellen der lokalen RuntimeEnvironments enthalten bestenfalls Einträge, die auch in den Tabellen des Runtime Managers zu finden sind. Es gibt keine lokalen Einträge, die in den globalen Tabellen des Runtime Managers nicht vorzufinden sind.

3. Es tritt niemals die Notwendigkeit ein, ein einmal in einer Tabelle eingetragenes Objekt wieder daraus zu entfernen. Nur beim Zurücksetzen des gesamten Systems werden alle Tabellen gemeinsam geleert.

4. Soll über ein lokales RuntimeEnvironment ein Klassen- oder Konstruktorobjekt beschafft werden, so wird zunächst in der betreffenden lokalen Tabelle nachgeschlagen. Findet sich dort das Objekt, so ist die Suche beendet. Andernfalls wird der Auftrag an den Runtime Manager weitergeleitet. Dieser sucht das gewünschte Objekt seinerseits in seiner eigenen Tabelle. Findet es sich dort, wird es an das lokale RuntimeEnvironment weitergegeben, welches dieses Objekt jetzt in seine eigene lokale Tabelle eintragen kann. Ist jedoch auch die Suche im Runtime Manager erfolglos, so ist man sicher, dass dieses Objekt im gesamten System noch nicht existiert. Bei einem Klassenobjekt veranlasst der Runtime Manager, dass es auf einer, gemäß einer Auswahlstrategie, bestimmten lokalen JVM neu erzeugt wird. Bei Konstruktorobjekten erhält der Runtime Manager die gewünschte Zielmaschine von dem auftraggebenden lokalen RuntimeEnvironment explizit. Den Stub auf das neu erzeugte Klassen- oder Konstruktorobjekt trägt der Runtime Manager schließlich in seinen Tabellen ein und liefert ihn an das lokale RuntimeEnvironment weiter.

Nur die Liste aller registrierten lokalen JVMs erfordert eine etwas abgeänderte Vorgehensweise. Hier ist es nicht eines der lokalen `RuntimeEnvironments`, das für einen neuen Eintrag in der Liste zuständig ist, sondern der Runtime Manager verändert selbst ohne direkten Auftrag eine seiner Listen, wenn er über den Start einer neuen lokalen JVM erfährt. Damit sofort alle lokalen `RuntimeEnvironments` über den Neuzugang einer JVM informiert werden, kontaktiert der Runtime Manager hierzu alle bereits existierenden lokalen JVMs und veranlaßt diese, die jeweils lokalen Laufzeitverwaltungsdaten neu vom Runtime Manager zu laden. Dieses Prinzip ist bereits in Abbildung 6 zu erkennen. Es ist der einzige Interaktionsvorgang zwischen dem `LocalJVM`-Objekt und der lokalen `RuntimeEnvironment`-Klasse.
Der Grund, warum die Liste der existierenden lokalen Java Virtual Machines in den lokalen `RuntimeEnvironment`-Klassen nicht ebenso lazy verwaltet wird, wird erst mit Abschnitt 4.2.4 richtig verständlich. Es muss nämlich lokal jederzeit zumindest die Anzahl der verfügbaren virtuellen Maschinen bekannt sein (für lokale Auswahlstrategien von JVMs), weshalb auch bei einem Neuzugang einer JVM alle lokalen `RuntimeEnvironments` sofort informiert werden müssen. Diese Benachrichtigung wird im vorgestellten System als eine Synchronisation der Laufzeitdaten aller lokalen JVMs realisiert.

Funktionalität

Der Runtime Manager verwaltet als zentrale Instanz lediglich globale, auf die ganze verteilte Umgebung bezogene Daten. Er hat nichts mit der Instantiierung von *remote objects* oder der Objekt-Migration zu tun. Diese wird unmittelbar von den lokalen JVMs durchgeführt. Zusammenfassend lässt sich folgender Aufgabenkatalog für den Runtime Manager zusammenstellen:

- Registrierung neuer lokaler JVMs
- Beschaffung von Klassenobjekten. Ist ein gewünschtes Objekt noch nicht im System vorhanden, so muss eine lokale JVM ausgewählt und dort ein neues Klassenobjekt angelegt werden.
- Beschaffung eines Konstruktorobjekts auf einer bestimmten lokalen JVM. Ist auf dieser Maschine noch kein solches Konstruktorobjekt existent, so wird dort ein solches angelegt.
- Übermittlung der internen Verwaltungsdaten an lokale `RuntimeEnvironments`.
- Neustart bzw. Beenden der gesamten verteilten Umgebung. Unter einem Neustart wird verstanden, dass alle angelegten Klassen- und Konstruktorobjekte der verteilten Umgebung gelöscht werden.

Erzeugung von Klassenobjekten

Gemäß §12.4 in [GJS_96] wird eine Java Klasse bei ihrer ersten *aktiven Nutzung* initialisiert. Unter einer aktiven Nutzung wird folgendes verstanden:

- Es wird eine statische Methode aufgerufen, die direkt in der Klasse deklariert ist (und nicht von einer Oberklasse geerbt wird).
- Ein Konstruktor der Klasse wird aufgerufen.
- Auf eine nicht-konstante Variable, die direkt in der Klasse deklariert ist, wird lesend oder schreibend zugegriffen. Unter einer „konstanten Variablen“ wird eine statische Variable verstanden, die mit dem Modifikator `final` versehen ist. Werte solcher Variablen werden bereits zur Übersetzungszeit bestimmt, weshalb Zugriffe auf die Variable niemals als aktive Nutzung angesehen werden.

Bei der Initialisierung einer Klasse werden alle statischen Initialisierungsblöcke und alle Initialisierungsausdrücke statischer Variablen in der textuellen Reihenfolge der Klassendefinition ausgewertet. Bevor jedoch eine Klasse initialisiert wird, müssen alle ihre Oberklassen initialisiert sein. Für Interfaces die durch die Klasse implementiert werden, braucht dies nicht zu gelten. Mit dieser Vorgehensweise soll garantiert werden, dass eine Klasse in einen definierten Zustand versetzt wird, bevor eine andere Klasse auf sie zugreifen kann.[12]

Wie bereits in 3.1 angesprochen wurde, entspricht die Klasseninitialisierung innerhalb einer Java Virtual Machine, der Erzeugung eines Klassenobjekts in der verteilten Java Laufzeitumgebung. Das Laufzeitsystem ist hier dafür verantwortlich, dass das Anlegen eines Klassenobjekts nach den gleichen Regeln abläuft, wie im lokalen Fall. Da Java Nebenläufigkeiten zulässt,

[12] In §12.4.1 und §8.5 von [GJS_96] werden einige Beispiele aufgeführt, die zeigen, dass es in Java dennoch möglich ist, auf statische Variablen einer Klasse zuzugreifen, bevor sie initialisiert werden.

stellt diese Aufgabe hohe Anforderungen an einen geeigneten Synchronisationsmechanismus. Es ist beispielsweise vorstellbar, dass zwei Threads versuchen gleichzeitig die gleiche Klasse zu initialisieren. Es ist durchaus auch möglich, daßs die Initialisierung einer Klasse rekursiv, also ausgehend vom eigenen Initialisierungscode, initiiert wird. Das später folgende Beispielprogramm zeigt einen solchen Fall.
Zur Erzeugung von Klassenobjekten ist im JDR-System innerhalb des Runtime Managers folgende Vorgehensweise implementiert (man vergleiche hierzu auch den in [GJS_96] unter §12.4.2 angegebenen Algorithmus für die lokale Java Virtual Machine).

1. Nachdem der Runtime Manager synchronisiert wurde, wird in der Klassenobjekttabelle nach dem Klassenobjekt der zu initialisierenden Klasse gesucht. Findet sich das Objekt, so wurde die Klasse bereits initialisiert und der Initialisierungsversuch wird abgebrochen.

2. Das Lock auf den Runtime Manager wird freigegeben und es wird versucht, die betreffende Oberklasse zu initialisieren (durch eine rekursive Abarbeitung dieser gesamten Prozedur).

3. Nun ist man sicher, dass für alle Oberklassen Klassenobjekte vorliegen. Nach einer erneuten Synchronisation des Runtime Managers muss jetzt nochmals in der Klassenobjekttabelle gesucht werden. Es wäre nämlich mittlerweile möglich gewesen, dass ein anderer Thread die Klasse bereits initialisiert hätte (unmittelbar nach Freigabe des Locks).

4. Findet sich in der Klassenobjekttabelle dagegen kein Eintrag, so wird in der Tabelle anstelle eines Klassenobjekts, ein Stellvertreter (`inProgress`) eingetragen, der angibt, dass die Klasse zur Zeit gerade initialisiert wird.
   Wird andererseits in der Tabelle bereits ein solcher `inProgress`-Vertreter vorgefunden, wird mittels `wait` gewartet (was implizit das Lock auf den Runtime Manager temporär freigibt) bis ein fertig initialisiertes Klassenobjekt vorliegt. Damit endet in diesem Fall die Prozedur.

5. Nachdem nun klar ist, dass kein anderer Thread zuvor die Klasse initialisiert hat und der `inProgress`-Stellvertreter sichert, dass auch kein anderer Thread zukünftig versuchen wird selbst die Klasse zu initialisieren, kann das Lock auf den Runtime Manager wieder freigegeben werden.

6. Es wird eine lokale JVM ausgewählt, auf der das Klassenobjekt angelegt werden soll.

7. Die ausgewählte lokale JVM wird beauftragt, die Klasse in ihrem Adressraum zu initialisieren.

8. Sobald das, von der lokalen JVM instantiierte und initialisierte Klassenobjekt vorliegt, wird das Lock des Runtime Manager erneut belegt. Das Klassenobjekt kann nun in die Klassenobjekttabelle eingetragen werden und alle wartenden Threads können benachrichtigt werden.

Bei dem hier vorgestellten Verfahren wird ein Klassenobjekt erst dann in die Klassenobjekttabelle des Runtime Managers eingetragen, wenn es vollständig initialisiert wurde. Zuvor wird an dessen Stelle nur ein Stellvertreterobjekt abgelegt. Dass diese Vorgehensweise problematisch ist, soll an folgendem Programm verdeutlicht werden.

```
class Alpha
{
    static int x = Beta.a;
}

class Beta
{
    static int a = 1;
    static int b = Alpha.x + 2;

    public static void main(String[] args)
    {
        System.out.println(a + " " + b);   // Ausgabe: 1 3
    }
}
```

Da es sich beim Ausführen der statischen Methode `Beta.main` um eine aktive Nutzung der Klasse `Beta` handelt, muss zuvor die Klasse erst initialisiert werden. Hierfür wird `Beta.a` der Wert `1` zugewiesen. Anschließend wird der Ausdruck `Alpha.x + 2` ausgewertet. Dies bedeutet aber eine aktive Nutzung der Klasse `Alpha`, weshalb zuvor also auch noch die Klasse `Alpha` initialisiert werden muss. Dabei wird nun aber wieder auf die, noch nicht vollständig initialisierte Klasse `Beta` zugegriffen. An dieser Stelle kommt es dann beim obigen Algorithmus zu einer Verklemmung, da beim Zugriff auf das Klassenobjekt von `Beta` in dem Initialisierungsausdruck für die statische Variable `Alpha.x` festgestellt wird, dass das Klassenobjekt für `Beta` noch nicht verfügbar ist (siehe 4. im obigen Algorithmus). Andererseits kann das Klassenobjekt aber auch nicht vollständig initialisiert werden, wenn der Wert von `Alpha.x` nicht bekannt ist.

Dieses Problem tritt innerhalb einer lokalen Java Virtual Machine nicht auf, da hier das, die Klasse repräsentierende `Class`-Objekt zu Beginn der Initialisierung nur mit einem Lock versehen wird, sodass andere Threads erst dann Zugriff auf die statischen Variablen und Methoden bekommen, wenn die Initialisierung der Klasse abgeschlossen ist (und damit auch ein konsistenter Zustand der Klasse vorliegt). Vom dem Thread aus, der die Initialisierung der Klasse durchführt, kann jedoch von Anfang an unbegrenzt auf das komplette, teilweise noch uninitialisierte `Class`-Objekt, zugegriffen werden. Da im obigen Beispiel alle Berechnungen und natürlich auch beide Klasseninitialisierungen nur in dem einzigen aktuellen Thread abgewickelt werden, kann es an keiner Stelle zu einer Verklemmung kommen.

Um ein ähnliches Verhalten innerhalb der verteilten Laufzeitumgebung zu erhalten, müsste also das Klassenobjekt bereits vor dessen Initialisierung in die Klassenobjekttabelle des Runtime Managers eingetragen werden. Das obige Programm würde sich damit auch innerhalb der verteilten Java Umgebung korrekt verhalten, da man jetzt auch auf eine teilweise initialisierte Klasse im Rahmen einer rekursiven Klasseninitialisierung zugreifen könnte. Allerdings würde diese Lösung es auch zulassen, dass bei gleichzeitigem Zugriff zweier Threads auf das Klassenobjekt, einer der Threads ein, nur teilweise initialisiertes Klassenobjekt erhalten würde. Auch hier macht sich nämlich wieder das aus 3.6 bekannte Synchronisationsproblem bemerkbar. Es kann hier auf der Ebene des Runtime Managers nämlich nicht entschieden werden, ob die Nachfrage nach einem Klassenobjekt `Beta` lokal gesehen vom gleichen Thread aus geschieht, wie eine zur Zeit laufende Klasseninitialisierung von `Beta` (dann darf ein nur halb initialisiertes Klassenobjekt herausgegeben werden), oder ob die Nachfrage von einem völlig unabhängigen Thread ausgeht (dann muss gewartet werden, bis das Klassenobjekt vollständig initialisiert ist).

Die zuletzt diskutierte Lösung birkt zwar nicht die Gefahr von Verklemmungen in sich, hat aber den Nachteil, dass nicht sichergestellt ist, dass ein vom Runtime Manager herausgegebe-

nes Klassenobjekt auch wirklich vollständig initialisiert ist. Dieses, völlig durch den Zufall bestimmte Konsistenzproblem, macht die Lösung für die Praxis unbrauchbar. Bei der ausführlich dargestellten ersten Variante besteht zwar die Einschränkung, dass verteilte Java Programme keine „Zyklen“ bezüglich ihrer statischen Initialisierungen enthalten dürfen – wird dies aber strikt befolgt, verhält sich die Klasse im verteilten System stets so wie erwartet.

Alternative Klasseninitialisierungsstrategien

Die Implementierung der verteilten Java Laufzeitumgebung hat unter anderem auch gezeigt, dass die für Java definierte Klasseninitialisierungsstrategie zumindest für die verteilte Umgebung auch weniger streng formuliert werden kann. Dies bezieht sich vor allem auf folgende zwei Aspekte:

- Bei einem Konstruktoraufruf wird auf den statischen Klassenanteil eigentlich nicht direkt zugegegriffen. Es ist also nicht nötig, einen Konstruktoraufruf als eine aktive Klassennutzung anzusehen. Konsequenterweise braucht eine Klasse demnach nicht initialisiert zu sein, wenn nur eine Instanz davon angelegt wird.
- Es gibt für das verteilte System keinen Grund, warum vor einer Klasseninitialisierung erst einmal alle noch nicht initialisierten Oberklassen initialisiert werden müssen. Wird innerhalb einer Klasseninitialisierung nirgendwo die Oberklasse aktiv genutzt, so braucht diese auch nicht initialisiert zu werden.

Der Grund, warum das JDR-System diese wesentlich weniger strengen Klasseninitialisierungsprinzipien ermöglicht, ist in der Architektur des Systems zu finden: sämtliche Instanzen des Laufzeitsystems arbeiten konsequent lazy. Systeminterne Objekte, wie z.B. Klassen- oder Konstruktorobjekte werden wirklich nur dann angefordert, wenn auf sie unbedingt zugegriffen werden muss. Da Zugriffe auf entfernte Objekte stets teuer sind, war dies eine der zentralen Entwurfsentscheidungen.

Aus Kompatibilitätsgründen zum lokalen Java arbeitet der Runtime Manager, trotz der geringeren Effizienz, normalerweise nach den ausführlich beschriebenen klassischen Klasseninitialisierungsregeln. Optional kann aber auch eine bedarfsgesteuerte, aber nicht mehr Java-konforme Klasseninitialisierung eingeschaltet werden.

Erzeugung von Konstruktorobjekten

Die Erzeugung von Konstruktorobjekten durch den Runtime Manager ist im Allgemeinen wesentlich einfacher als die Erstellung von Klassenobjekten. Ein neues Konstruktorobjekt wird stets für eine bestimmte lokale JVM angefordert, wenn ein neues *remote object* auf dieser Maschine instantiiert werden soll, dort aber noch kein Konstruktorobjekt verfügbar ist. Der Runtime Manager beantwortet Anfragen nach Konstruktorobjekten nach folgendem Schema:

1. Nachdem der Runtime Manager synchronisiert wurde, wird in der Konstruktortabelle nach dem betreffenden Konstruktorobjekt gesucht. Unmittelbar anschließend wird das Lock des Runtime Managers wieder freigegeben. Fand sich das Objekt in der Tabelle, so wird es an die auftraggebende lokale JVM weitergegeben und die Prozedur ist beendet.
2. Gibt es noch auf keiner lokalen JVM ein Konstruktorobjekt für die betreffende Klasse, so muss jetzt explizit sichergestellt werden, dass die Klasse bereits initialisiert ist (Konstruktorobjekte werden Zwecks einer aktiven Klassennutzung herausgegeben, die Klasse muss also zuvor initialisiert sein; beim Einsatz der bedarfsgesteuerten Klasseninitialisierung entfällt dieser Punkt).

3. Da für eine möglicherweise stattfindende Klasseninitialisierung das Runtime Manager Lock wieder aufgegeben wurde, muss nun eine erneute Synchronisation durchgeführt werden. Das Konstruktorobjekt wird jetzt nochmals in der Tabelle gesucht (ein anderer Thread hätte ja mittlerweile das Anlegen dieses Konstruktorobjekts bewirken können).

4. Ist der Eintrag für das Konstruktorobjekt in der Tabelle immer noch unbesetzt, so wird eine lokale JVM beauftragt, ein Konstruktorobjekt bei sich neu zu generieren. Unmittelbar nachdem der Runtime Manager einen Stub auf dieses Objekt erhalten hat, wird das Lock wieder freigegeben.

4.2.3 Lokale Virtuelle Maschinen

Aufgaben

Eine verteilte Java Umgebung besteht im Allgemeinen aus mehreren lokalen JVMs und dem zentralen Runtime Manager. Jede dieser Instanzen läuft als eigenständiger Prozess. Die Kommunikation erfolgt ausschließlich per RMI.

Da man mit einer normalen Java Virtual Machine nicht direkt kommunizieren kann, braucht man ein Objekt auf dieser Maschine als Ansprechpartner. Das Objekt, das in einer verteilten Umgebung eine lokale JVM repräsentiert, ist eine Instanz der Klasse `LocalJVM`. Dieses entfernte Objekt wird beim Start einer lokalen JVM beim Runtime Manager registriert. Folgende Dienste werden von dem `LocalJVM`-Objekt angeboten:

- Anlegen eines neuen Konstruktorobjekts für eine bestimmte *remote class* auf der lokalen virtuellen Maschine.
- Anlegen eines Klassenobjekts für eine bestimmte *remote class* auf der lokalen JVM.
- Umzug eines *remote objects* auf die lokale JVM.
- Zurücksetzen der lokalen `RuntimeEnvironment`-Klasse. Beim nächsten Zugriff von der lokalen Maschine auf die verteilte Umgebung wird dadurch ein Neuladen der Laufzeitverwaltungsdaten vom Runtime Manager erzwungen.

Da man im Quelltext einer Klasse möglicherweise explizit Kommandos einfügen möchte, die Aussagen über den Aufenthaltsort eines *remote objects* machen, ist es notwendig, dass man einzelne Maschinen identifizieren kann. Die Vergabe von Identifikationskennungen muss dynamisch geschehen, da ja die Konfiguration eines verteilten Systems nicht festgelegt ist und jedesmal neu zusammengestellt werden kann. Deswegen vergibt der Runtime Manager bei der Registrierung einer lokalen JVM, ausgehend von 0, Identifikations-Nummern an alle virtuellen Maschinen. Über eine solche Nummer kann dann eine JVM des Systems eindeutig adressiert werden.

Objekt-Migration

Um ein Objekt von der momentanen Lage auf eine andere Maschine umziehen zu lassen, bietet das Handle verschiedene Methoden an, über welche ein solcher Umzug initiiert werden kann. Es ist entweder möglich, ein Objekt auf die gleiche Maschine wie das betrachtete Handle zu schicken oder auf eine, durch eine Identifikations-Nummer adressierte, lokale JVM umziehen zu lassen. Ein Objekt-Umzug läuft dann nach folgendem Schema ab:

1. Das Handle beschafft sich vom lokalen RuntimeEnvironment das `LocalJVM`-Objekt, das die Maschine repräsentiert, auf welche das *remote object* umziehen soll.

2. Das `LocalJVM`-Objekt wird durch einen entfernten Methodenaufruf beauftragt, das *remote object* auf seine Heimatmaschine zu holen.

3. Der eigentliche Objekt-Umzug wird jetzt in einer Art Handshake-Verfahren zwischen dem `LocalJVM`-Objekt und dem umzuziehenden *remote object* (eine Instanz der Oberklasse `RemoteInstance`) durchgeführt:

   3.1. Die `LocalJVM` signalisiert dem *remote object*, dass der Umzug beginnen soll

   3.2. Das *remote object* prüft, ob ein Umzug zur Zeit möglich ist: dies ist der Fall, wenn auf das Objekt zur Zeit keine Methoden angewendet werden, kein anderweitiger Objekt-Umzug bezüglich dieses Objekts im Gange ist und das `_resident`-Flag[13] nicht gesetzt ist. Trifft dies zu, so legt es von sich selbst, mittels Object Serialization eine serialisierte Repräsentation, in Form eines `byte`-Arrays an. Dieses Array wird schließlich an die `LocalJVM` geschickt. Bereits ab dem Zeitpunkt, an dem festgestellt wird, dass ein Umzug zulässig ist, wird das Objekt für weitere Zugriffe (Methodenaufrufe, Objekt-Migrationen, usw.) gesperrt. Konkret bedeutet dies, dass bei jedem folgenden Zugriff eine Exception ausgelöst wird, die angibt, dass eine Objekt-Migration zur Zeit im Gange ist.
   Ist ein Umzug aus den genannten Gründen nicht zulässig, so lehnt das *remote object* die geforderte Objekt-Migration ab und das bisherige Objekt bleibt weiterhin gültig.

   3.3. Die `LocalJVM` erzeugt nun aus der erhaltenen `byte`-Array Repräsentation wieder ein neues, lokales *remote object* (also eine Kopie von dem ursprünglichen Objekt auf einer anderen JVM). Ein Stub für dieses neue Objekt wird anschließend an das noch aktuelle *remote object* geschickt, mit der Aufforderung, den Umzug zu beenden.

   3.4. Das alte *remote object* setzt daraufhin seinen Verweis, dass es umgezogen ist, auf den neuen Stub und beendet damit die Objekt-Migration. Alle weiteren Zugriffe auf das Objekt werden ab sofort auf die umgezogene Kopie umgelenkt. Da sich diese Umlenkung nach mehreren Migrationen über mehrere Stationen erstrecken kann, werden bei jedem Zugriff auf ein Objekt ebenfalls die Stubverweise, die die neue Objektlage beschreiben, aktualisiert, um indirekte Referenzierungen zu vermeiden.

4. Der interne Stub-Verweis des Handles wird auf einen neuen Stub für das umgezogene Objekt gesetzt.

Da man für ein *remote object* jederzeit die Identifikations-Nummer der lokalen JVM, auf der es beheimatet ist, ermitteln kann, ist es mit den vorgestellten Methoden auch möglich, ein *remote object* auf die virtuelle Maschine umziehen zu lassen, auf der ein anderes *remote object* sich befindet. Dies wird nämlich in der Praxis, der wohl am meisten vorkommende Fall sein.

[13] Durch das Setzen des `_resident`-Flags kann explizit verhindert werden, dass ein *remote object* umziehen darf. Für bestimmte Objekte kann es durchaus wünschenswert, ja teilweise sogar erforderlich sein, dass Objekt-Migrationen ausgeschlossen sind. `RemoteThread`-Objekte dürfen beispielsweise nach ihrer Erzeugung nicht mehr umziehen, weil sie bereits mit ihrer Instantiierung an eine virtuelle Maschine, durch die Erzeugung eines dort lokalen Threads, gebunden sind. Generell sollten alle Objekte, deren Instanzvariablen möglicherweise nicht serialisierbare Objekte referenzieren, vom Benutzer als nicht migrationsfähig gekennzeichnet werden. Im JDR-System gelten Objekte von Klassen, die das `Resident`-Interface implementieren, als nicht migrationsfähig.

4.2.4 Lokale Schnittstelle zur Laufzeitumgebung

Die Verwaltungsdaten der verteilten Java Umgebung werden aus Effizienzgründen, beim ersten Zugriff einer Java Virtual Machine auf das verteilte System, automatisch geladen und lokal gespeichert. Da auch Java Virtual Machines, die nicht als lokale JVM im verteilten System registriert sind, die Möglichkeit erhalten sollen, auf Objekte der verteilten Umgebung zugreifen zu können,[14] musste die Schnittstelle zur Umgebung von den Stellvertreterobjekten einer lokalen JVM (`LocalJVM`-Objekte) getrennt werden. Außerdem musste sichergestellt sein, dass auf einer ganz normalen JVM, die nicht zum verteilten System gehört, also dem Runtime Manager auch nicht bekannt ist, diese Schnittstelle mit ihren lokalen Verwaltungsinformationen automatisch initialisiert wird.

Eine Schnittstelle, die solches leistet, wird durch die Klasse `RuntimeEnvironment` realisiert. Zugriffe von *remote objects* oder von lokalen Objekten auf die verteilte Umgebung werden ausschließlich über diese Klasse abgewickelt. Nur diese Klasse greift auf den Runtime Manager zu (siehe dazu auch Abbildung 6). Sie besitzt nur statische Methoden und Variablen und initialisiert sich beim Laden innerhalb einer virtuellen Maschine selbstständig.

Folgende Funktionalität der Klasse `RuntimeEnvironment` genügt für die Interaktion im gesamten System:

- Beschaffung eines bestimmten Klassenobjekts. Ist ein solches lokal nicht in den Tabellen registriert, so wird der Auftrag an den Runtime Manager weitergeleitet.
- Beschaffung eines bestimmten Konstruktorobjekts. Hierzu wird, gemäß einer Auswahlstrategie, zunächst eine lokale JVM bestimmt, auf welcher das Konstruktorobjekt beheimatet sein soll. Findet sich in der Tabelle aller Konstruktorobjekte für diese Maschine kein solches Objekt, so wird der Auftrag wieder an den Runtime Manager weitergegeben.
- Festlegen der Auswahlstrategie für Konstruktorobjekte, womit effektiv natürlich der Ort eines neuen *remote objects* festgelegt wird. Zur Zeit kann man entweder vor jeder Objektinstantiierung explizit festlegen, wo das Objekt erzeugt werden soll, oder es kann eine implizite interne Strategie verwendet werden.
- Erzwingung einer Neuinitialisierung der Verwaltungsdaten der Schnittstellen-Klasse. Bei dem ersten nachfolgenden direkten Zugriff auf das verteilte Laufzeitsystem wird automatisch der Runtime Manager kontaktiert und die neuesten Verwaltungsdaten werden kopiert.

4.2.5 Remote Threads

Die durch die Java Virtual Machine zur Verfügung gestellten Threads sind in einer verteilten Umgebung schlecht einsetzbar, da sie jeweils nur innerhalb der lokalen Java Virtual Machine erzeugt werden können. Es ist weder möglich, auf einer entfernten virtuellen Maschine einen

[14] Eine lokale Klasse `B` mit einer `main`-Methode lässt sich durch den Aufruf „`java B`“ ausführen. Ein transparenter Umgang mit *remote classes* bzw. *remote objects* erfordert, dass sich die entsprechende, als `remote` deklarierte Klasse `B` ebenfalls mit diesem Aufruf ausführen lässt. Da die hiermit neu gestartete JVM nicht Teil einer verteilten Laufzeitumgebung ist (also dem Runtime Manager auch nicht bekannt ist), aber auf das verteilte System zugreifen muss, muss auch für diesen Fall sichergestellt sein, dass für diese JVM ein lokales `RuntimeEnvironment` angelegt wird, ohne das ein `LocalJVM`-Objekt existiert.

neuen Thread zu starten, noch auf einen entfernten Thread in irgendeiner Weise zuzugreifen. Es ist noch nicht einmal zulässig, ein lokales `Thread`-Objekt als Argument eines entfernten Methodenaufrufs zu übergeben, da `Thread`-Objekte, als „systemnahe“ Objekte nicht serialisierbar sind.[15]

Die verteilte Java Laufzeitumgebung muss also ein Konzept zur Verfügung stellen, mit dem es möglich wird, neue Threads auf anderen virtuellen Maschinen zu erzeugen bzw. Threads anderer Maschinen steuern zu können. Erst damit kann die volle Leistungsfähigkeit eines verteilten Systems ausgenutzt werden, da nun auf verschiedenen Rechnern nebenläufig ein verteiltes Programm abgearbeitet werden kann. Diese Forderungen werden im JDR-System durch die Klasse `jdr.lang.RemoteThread` erfüllt. Folgende Eigenschaften charakterisieren die *remote threads* des JDR-Systems:

- Ein *remote thread* kann auf verschiedene Weisen angelegt werden:
  - Ein `Runnable`-Objekt wird dem `RemoteThread`-Konstruktor als Argument übergeben. Handelt es sich bei dem `Runnable`-Objekt um ein lokales Objekt, so kann entweder noch die lokale JVM angegeben werden, auf der der Thread gestartet wird oder das Laufzeitsystem erzeugt den Thread auf einer selbstgewählten Maschine. Ist das `Runnable`-Objekt dagegen ein *remote object*, so wird der *remote thread* auf der gleichen Maschine angelegt, auf der sich das `Runnable`-*remote object* befindet.
  - Es wird eine entfernte Unterklasse von `RemoteThread` angelegt, die ihre eigene `run`-Methode definiert. In diesem Fall wird der *remote thread* auf der Maschine erzeugt, auf der das `RemoteThread`-Unterklassenobjekt durch das Laufzeitsystem instantiiert wurde.
- `RemoteThread`-Objekte sind *remote objects* die das `Resident`-Interface implementieren, also nicht migrationsfähig sind. Deswegen kann auf einen *remote thread* von jeder beliebigen Java Virtual Machine aus zugegriffen werden.
- Handelt es sich bei dem `Runnable`-Objekt, das die `run`-Methode des neuen *remote threads* bereitstellt, um ein *remote object*, so wird auch dieses Objekt ab dem Zeitpunkt der Instantiierung des *remote threads* zu einem nicht mehr migrationsfähigen Objekt.[16]
- Bei der Instantiierung eines *remote threads* wird auf der betreffenden lokalen JVM ein korrespondierendes `Thread`-Objekt angelegt. Entfernte Methodenaufrufe bezüglich des *remote threads* werden an dieses lokale `Thread`-Objekt weitergeleitet. Die Methoden eines `RemoteThread`-Objekts bewirken also genau das gleiche, wie die entsprechenden Methoden der `Thread` Klasse.

Wie im letzten Punkt zuvor bereits angedeutet wurde, werden *remote threads* des verteilten Systems immer auf lokale Threads einer bestimmten lokalen virtuellen Maschine abgebildet.

[15] Lässt man ein *remote object* das `Runnable`-Interface implementieren und legt man einen neuen lokalen Thread bezüglich dieses `Runnable`-Objekts an, dann sieht es zwar auf den ersten Blick so aus, als könnte man damit auch einen Thread auf einer entfernten Maschine starten, da das Skeleton ja die `run`-Methode des `remote objects` in einem eigenen neuen Thread ausführt. Allerdings hat man über diesen „entfernten“ Thread überhaupt keine Kontrolle, da sich sämtliche `Thread`-Methoden auf den lokalen Thread beziehen, der lediglich auf die Terminierung des „entfernten“ Threads wartet. Außerdem ist der neu angelegte Thread natürlich lokal und ein entfernter Zugriff somit ausgeschlossen.

[16] Es ist natürlich jederzeit möglich, das `_resident`-Flag für ein solches `Runnable`-Objekt per Hand wieder zurückzusetzen, damit eine Objekt-Migration nach dem Terminieren des Threads wieder zulässig wird.

Insofern sind `RemoteThread`-Objekte nichts anderes als Stellvertreter für `Thread`-Instanzen, mit einer besonderen entfernten Objekt-Semantik. Dieser Bezug zu einem lokalen Thread ist auch die Ursache dafür, dass `RemoteThread`-Objekte nicht auf eine andere lokale JVM umziehen können.

In einer Java Virtual Machine werden Threads in sogenannten *Thread Groups* gegliedert, welche ihrerseits wiederum übergeordneten Thread Groups angehören. Obwohl entfernte Thread Groups sicherlich auch für eine entfernte Laufzeitumgebung einen Sinn machen würden, gibt es zur Zeit noch keine korrespondierende Lösung für das JDR-System. Deswegen wurden auch alle Methoden, die Thread Groups betreffen, aus der momentan aktuellen Realisierung der Klasse `RemoteThread` herausgenommen.

4.2.6 Objektverteilung

Klassen- und Objektdistributoren

Die Verteilung der Klassenobjekte und die Auswahl der Konstruktorobjekte zur Instantiierung eines *remote objects* auf einer bestimmten Maschine, werden innerhalb von JDR von zwei verschiedenen Komponenten durchgeführt, wie bereits in den Abschnitten 4.2.2 und 4.2.4 angedeutet wurde.

Der Runtime Manager, der als solcher die vollständige Sicht auf das gesamte System besitzt, wählt für eine noch nicht initialisierte Klasse eine lokale JVM aus, auf der dann das Klassenobjekt angelegt wird. Die lokalen `RuntimeEnvironments`, die typischerweise nur eine lokale Sicht auf das verteilte System besitzen, sind für die Auswahl der Heimatmaschine eines neuen *remote objects* verwantwortlich. Diese Arbeitsteilung läßt sich auch logisch motivieren, wenn man bedenkt, dass der Runtime Manager eigentlich nur für die Verwaltung der *remote classes* zuständig ist und die Objekt-Instantiierung von *remote objects* völlig in den Händen des lokalen `RuntimeEnvironments` liegt.

Die Auswahlstrategien für den Ort einer Objektinstantiierung oder einer Initialisierung eines Klassenobjekts sind nicht unmittelbar in den Runtime Manager oder die einzelnen lokalen `RuntimeEnvironments` integriert, sondern werden durch sogenannte *Distributoren* realisiert. Dem Runtime Manager und jedem lokalen `RuntimeEnvironment` ist jeweils genau ein Distributor-Objekt zugeordnet, welches von der jeweiligen Komponente stets bei Auswahlentscheidungen herangezogen wird. Standardmäßig arbeitet der Runtime Manager mit einem `ClassDistributor`-Objekt und jedes `RuntimeEnvironment` verwendet jeweils ein eigenes `ObjectDistributor`-Objekt. Durch die Abtrennung der Strategie von der Realisierung der einzelnen Laufzeitsystemkomponenten, kann jederzeit für eine Komponente die Auswahlstrategie durch eine neue Strategie bzw. ein neues Distributor-Objekt ersetzt werden.

In Abbildung 7 ist das Zusammenwirken der `RuntimeEnvironments` mit dem Runtime Manager dargestellt. Es soll hier insbesondere deutlich werden, dass die jeweiligen Distributor-Objekte für ihre Auswahlentscheidungen stets auf die lokalen Laufzeitdaten der zugeordneten Komponente zugreifen können. Da die Laufzeitdaten des Runtime Managers jederzeit vollständig den aktuellen Systemzustand wiederspiegeln, kann bei einer Strategie für einen `ClassDistributor` vorausgesetzt werden, dass stets sämtliche Laufzeitverwaltungsdaten komplett für den Auswahlalgorithmus zur Verfügung stehen.
Weiterhin muß bezüglich Abbildung 7 noch angemerkt werden, dass die Auswahl einer Zielmaschine für ein Konstruktorobjekt zwar lokal durch den `ObjectDistributor` erfolgt, im Falle

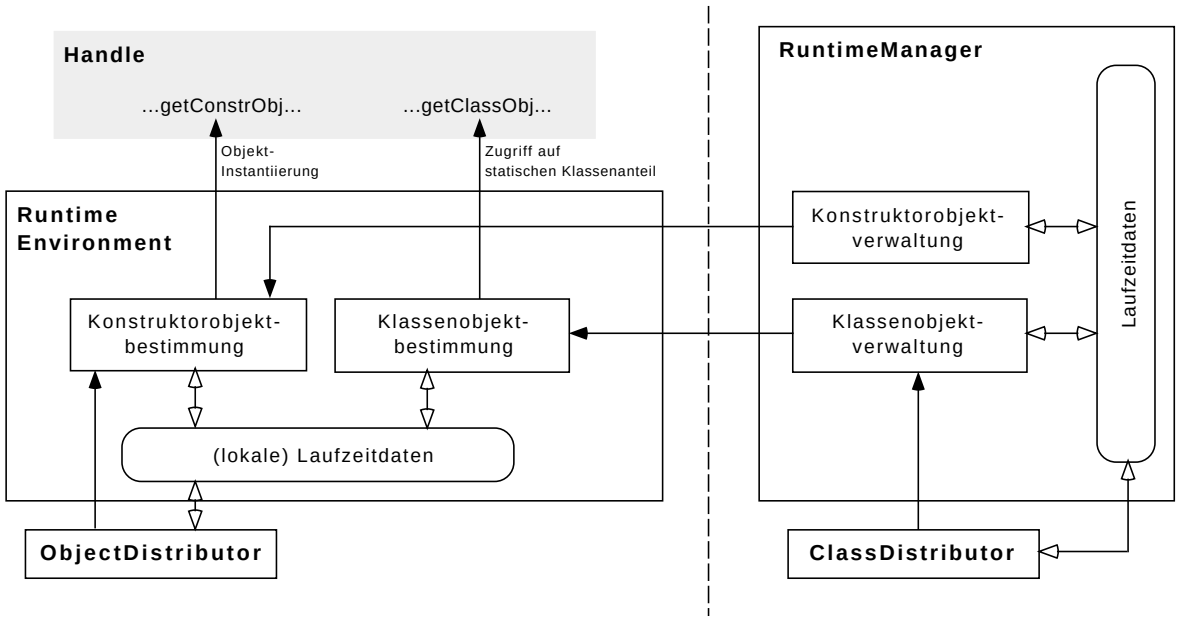


Abbildung 7: Klassen- und Objektverteilung im JDR-System

eines noch nicht vorhandenen bzw. bekannten Konstruktorobjekts aber der Runtime Manager beauftragt wird, ein Konstruktorobjekt für die geforderte Maschine zu besorgen.

Ausblick

Zur Zeit gibt es noch kein Konzept für das JDR-System, mehrere zusammengehörige Objekte zu einem *Objekt-Cluster* zu gruppieren [CBS_93]. In einem objektbasierten System haben Cluster vorwiegend den Zweck, den Kommunikationsaufwand zu minimieren, nebenläufige Prozesse zu strukturieren und durch Objektlokalität Speicherzugriffe zu optimieren [BCL_94]. Erst auf dieser Ebene lassen sich also Verteilungs- und Migrationsstrategien sinnvoll und effizient einsetzen. Statt mit Objekten hat das Laufzeitsystem dann vorwiegend mit Clustern zu tun. Die Distributoren aus dem vorhergehenden Abschnitt haben dann nicht die Aufgabe eine Maschine für ein neues Objekt auszuwählen, sondern einen geeigneten Cluster (damit wird dann implizit natürlich auch eine Maschine ausgewählt).
Für diese Auswahl ist es sinnvoll neben den reinen dynamischen Laufzeitdaten, die sich beispielsweise aus der Beobachtung der Kommunikation innerhalb eines Clusters ergeben, auch auf die Resultate statischer Code-Analysen zurückzugreifen. Die Einführung von Objekt-Clustern bedingt also auch den Übersetzungsvorgang von entfernten Klassen, da das Ergebnis einer statischen Cluster-Analyse typischerweise direkt im Programm bzw. – im vorliegenden Fall – in einer Klassendatei codiert werden würde [CBS_93].

4.2.7 Fehler während der Programmausführung

Wird während einer Ausführung eines normalen Java-Programms eine `RuntimeException` ausgelöst, welche nicht explizit in einer `try-catch`-Anweisung abgefangen wird, so führt dies in der Regel nach einer Fehlermeldung zum sofortigen Programmabbruch. `RuntimeExceptions` können in einer verteilten Umgebung jedoch auf verschiedenen lokalen virtuellen Maschinen auftreten. Eine solche Exception darf hier nicht zur Beendigung der lokalen Maschine führen,

da sonst durch die entstehenden Inkonsistenzen das gesamte verteilte System lahmgelegt werden würde. Deswegen werden selbst `RuntimeExceptions` vom RMI-System „eingepackt“ und an den Aufrufer weitergeleitet. Letztendlich kommt es also zu einem Abbruch der Java Virtual Machine, über welche ursprünglich ein verteiltes Java Programm gestartet wurde. An dieser Stelle wird nun erst richtig klar, wieso eine wichtige Forderung an eine verteilte Laufzeitumgebung darin besteht, dass das Ausführen von verteilten Java-Programmen über eine virtuelle Maschine durchgeführt werden muss, die nicht zu der Laufzeitumgebung gehört.[17]

In einer verteilten Umgebung bringt lediglich die Methode `exit` der Klasse `java.lang.System` Probleme mit sich. Diese beendet augenblicklich die lokale JVM, auf der die Methode ausgeführt wird. Damit gerät aber die verteilte Umgebung in einen undefinierten Zustand: sie ist im folgenden nicht mehr verwendbar. An dieser Stelle ist der Programmierer eines verteilten Java-Programms gefragt. Die Klassenbibliothek des JDR-Systems bietet entsprechende Methoden an, die es erlauben, ein verteiltes Programm wie im lokalen Fall über `exit` zu beenden.[18]

4.3 Ausführung verteilter Java Programme

Das verteilte Java Laufzeitsystem wurde so aufgebaut, dass nach seinem Start, verteilte Java Programme genau so ausgeführt werden können, wie im lokalen Fall. Eine als `remote` deklarierte Klasse `B`, muss sich also durch den Aufruf „`java B`“ in der verteilten Umgebung starten lassen. In diesem Abschnitt wird erläutert, wie eine verteilte Laufzeitumgebung eingerichtet wird und wie man diese geeignet konfigurieren kann, um verteilte Programme darauf ausführen zu können.

Dem Runtime Manager kommt innerhalb des verteilten Systems eine zentrale Rolle zu. Er kann als Repräsentant der gesamten Umgebung angesehen werden, da er, wie in 4.2.2 gefordert wurde, stets vollständig alle aktuellen Verwaltungsdaten des Systems kennt. Kennt eine lokale JVM den Rechner auf dem ein Runtime Manager gestartet wurde und den Port, über den er dort erreichbar ist, so kann sie sich bei ihm registrieren lassen und wird damit automatisch in das System integriert. Rechner und Port können also zur Identifikation einer verteilten Java Laufzeitumgebung herangezogen werden.

Im JDR-System werden diese Identifikationsdaten, bestehend aus Rechner des Runtime Managers und dessen Port, üblicherweise in einer sogenannten *Konfigurationsdatei* festgelegt. Momentan können in einer Konfigurationsdatei lediglich zwei verschiedene Properties definiert werden: `runtime.manager.host` gibt den Rechner an, auf dem der Runtime Manager läuft, `runtime.manager.port` bezeichnet den betreffenden Port, über den man einen Stub für das `RuntimeManager`-Objekt bekommt. Konfigurationsdateien werden alle in einem bestimmten Verzeichnis abgespeichert, in welchem bei Bedarf nach einer bestimmten Konfiguration gesucht wird.[19]

Einen Runtime Manager für eine Konfigurationsdatei `djava.config` startet man auf dem entsprechenden Rechner stets unter Angabe des Konfigurationsdateinamens. In UNIX lautet ein

[17] Damit sollten die Beweggründe nun endgültig verständlich sein, warum in 4.2.4 gefordert wurde, dass auch eine, nicht zur Laufzeitumgebung gehörende JVM, auf eine verteilte Umgebung zugreifen kann.

[18] Für ein verteiltes System besteht hier ein Freiheitsgrad, den es im Falle einer lokalen JVM nicht gibt: entweder kann ein verteiltes Java-Programm beendet werden und die gesamte Umgebung läuft weiter, oder die gesamte verteilte Laufzeitumgebung wird beendet. Mit der statischen Methode `exit` der Klasse `DistributedRuntime` wird die erste Variante realisiert, mit der Methode `DistributedRuntime.kill` ist ein Gesamtabbruch des Systems möglich.

[19] Das Basisverzeichnis des JDR-Systems wird durch das System-Property `jdr.home` bestimmt. Alle Konfigurationsdateien werden im Verzeichnis `configs` abgelegt, welches sich direkt im JDR-Basisverzeichnis befindet.

solcher Aufruf beispielsweise „`jdrm djava.config`“. Eine neue lokale JVM kann man jetzt zu dem System von einem beliebigen Rechner innerhalb der gleichen Domain und des gleichen Dateisystems[20] mittels des Aufrufs „`jdrj djava.config`“ hinzufügen. Wird auf die Angabe einer Konfigurationsdatei verzichtet, so wird die, durch das System-Property `jdr.config` bestimmte Standardkonfiguration verwendet.

Führt man eine *remote class* `B` mit einer `main`-Methode nun mittels des Kommandos „`java B`“ aus, so wird auch hier die durch das Property `jdr.config` bestimmte Laufzeitumgebung als Ausführungssystem gewählt. Durch den Aufruf „`java -Djdr.config=djava.config B`“ lässt sich aber auch eine beliebige andere Umgebung `djava.config` als Ausführungsumgebung festlegen.

Trotz des, zur lokalen Ausführung von Java-Programmen, analogen Aufrufens von verteilten Java Programmen, sei nochmals darauf hingewiesen, dass durch das Kommando „`java B`“ (`B` sei hier natürlich eine *remote class*) keineswegs die `main`-Methode in der gestarteten virtuellen Maschine ausgeführt wird. Bei der Klasse `B` handelt es sich nämlich um ein Handle, in dessen `main`-Methode versucht wird über das lokale `RuntimeEnvironment` das Klassenobjekt für `B` im verteilten System zu lokalisieren und über dieses Objekt, die Methode `main` auszuführen. Dabei wird automatisch die Klasse `RuntimeEnvironment` initialisiert, d.h. der Runtime Manager wird lokalisiert und die Verwaltungsdaten werden kopiert (siehe 4.2.4). Andere Aktionen werden auf der gestarteten virtuellen Maschine nicht durchgeführt. Sie dient also lediglich als Initiator für eine Programmausführung in einer verteilten Java Laufzeitumgebung und ermöglicht, wie in 4.2.7 erläutert, ein korrektes Terminieren eines Java-Programms.

Wie man sieht, lässt die Architektur des Systems es durchaus zu, dass mehrere verteilte Java-Programme innerhalb einer Umgebung gleichzeitig ausgeführt werden. Andererseits erlaubt das hier vorgestellte Konfigurationsprinzip es ebenfalls, dass nebeneinander mehrere Laufzeitumgebungen eingerichtet werden können.

4.4 Laufzeitsystem-Zugriffe auf Quelltextebene

4.4.1 Umgang mit Objekt-Migration

Die auf den ersten Blick einzige „Spracherweiterung“, die für das verteilte Java System vorgenommen werden musste, besteht in dem neuen Klassenmodifikator `remote`. Der Übersetzer sorgt dafür, dass Klassendefinitionen die mit dem `remote`-Modifikator verstehen werden und für die keine explizite Oberklasse angegeben wird, automatisch Unterklassen der Klasse `RemoteObject` sind. Damit kann innerhalb des Quelltextes mittels des Ausdrucks „`obj instanceof RemoteObject`“ überprüft werden, ob es sich bei einem Objekt um ein lokales oder um ein entferntes Objekt handelt.

Nun besitzen *remote objects* einige zusätzliche Eigenschaften, auf die man als Programmierer Einfluss nehmen möchte. Diese Eigenschaften betreffen vor allem die Lage des Objekts im System und die Möglichkeit der Objekt-Migration. Es sollte für den Programmierer explizit möglich sein, die Lage eines Objekts abzufragen und eine Objekt-Migration durchzuführen[21]. Solche Aktionen müssen also durch entsprechende Anweisungen in den Quelltext eingestreut

[20] JDR unterstützt momentan noch kein dynamisches Stub-Laden.

[21] Zur Zeit ist dies die einzige Möglichkeit eine Objekt-Migration zu initiieren. Weder das Laufzeitsystem noch der Übersetzer führen momentan Analysen durch, die Aussagen über eine geeignete Lage eines Objekts machen. Solange das Laufzeitsystem keine *Objekt-Cluster* unterstützt, versprechen dynamische Objektverteilungsstrategien wenige Effizienz-Gewinne.

werden können. Damit man hierfür aber nicht die Sprache Java um weitere Anweisungen ergänzen muss, werden diese, für *remote objects* spezifischen Aktionen durch Methoden der gemeinsamen Oberklasse `RemoteObject` realisiert. Die Schnittstelle der Klasse `RemoteObject` sieht folgendermaßen aus:

```
public remote class RemoteObject implements Serializable
{
    public int _location();
    public boolean _fetch();
    public boolean _moveTo(int target);
    public boolean _moveTo(RemoteObject obj);
    public void _disableMigration();
    public void _enableMigration();
    public void _deref();
    public void _deref(int n);
}
```

Die Methode `_location` liefert die Identifikationsnummer der lokalen JVM, auf der ein *remote object* beheimatet ist.
Mit Hilfe der Methode `_fetch` lässt sich ein Objekt auf die Maschine umziehen, auf der die `_fetch`-Methode ausgeführt wird. Mittels `_moveTo` kann ebenfalls eine Objekt-Migration durchgeführt werden, nur lässt sich hier das Ziel des Umzugs entweder durch eine Identifikationsnummer einer lokalen JVM oder durch Angabe eines weiteren *remote objects* festlegen. Im letzten Fall wird das Objekt auf die gleiche Maschine transferiert, auf der sich dieses weitere *remote object* augenblicklich befindet.
`_disableMigration` und `_enableMigration` können sinnvoll nur auf *remote objects* angewendet werden, deren Klassen nicht das `Resident`-Interface implementieren. Diese Objekte sind im Allgemeinen migrationsfähig. Allerdings lässt sich diese Migrationsfähigkeit explizit mittels der beiden Methoden ein- und ausschalten.[22]
Die `_deref`-Methoden führen die in Abschnitt 3.2 beschriebenen Dereferenzierungsmethoden aus. Das Argument `levels` gibt dabei die maximale Rekursionstiefe des Dereferenzierungsvorgangs an. Wird die Methode ohne Argument aufgerufen, so findet eine vollständige Auflösung indirekter Referenzierungen statt[23].

4.4.2 Dynamische Konfiguration des Laufzeitsystems

Über die vorgestellten Methoden der Oberklasse `RemoteObject` lässt sich der Umgang mit *remote objects* beeinflussen. Es ist jedoch nicht möglich Parameter abzufragen oder zu verändern, die das gesamte verteilte Laufzeitsystem betreffen. Hierzu zählen vor allem die Objektinstantiierungsstrategie und die Strategie für das Anlegen von Klassenobjekten. Die Klasse `DistributedRuntime` stellt für den Programmierer eine lokale Schnittstelle für das verteilte Laufzeitsystem dar. Die Methoden dieser Klasse bieten zugleich auch die einzige Möglichkeit für den Benutzer, auf Programmebene direkt auf das Laufzeitsystem als Ganzes einzuwirken.

[22] Zur Zeit kann von jeder Instanz aus die Migrationsfähigkeit eines beliebigen *remote objects* ein- und ausgeschaltet werden. Ein Verfahren welches über eine entsprechende Rechtevergabe regelt, wer die Migrationsfähigkeit eines Objekts verändern darf, wäre sicherlich eine sinnvolle Erweiterung des hier vorgestellten Migrationskonzepts.

[23] Bei zyklischen Verweisen zwischen Objekten terminiert die Methode zur Zeit nicht. Deswegen ist es momentan empfehlenswert, eine maximale Rekursionstiefe anzugeben.

```
public final class DistributedRuntime
{
    public static final int machines();
    public static final int thisLocation();

    public static final void setClassDistributor(Distributor distr);
    public static final void setObjectDistributor(Distributor distr);

    public static final void setTarget();
    public static final void setTarget(int t);
    public static final void setTarget(RemoteObject obj);
    public static final void resetTarget();

    public static final void exit(int status);
    public static final void kill(int status);
}
```

Über die Methode `machines` kann man die aktuelle Anzahl, der zum verteilten System gehörenden lokalen JVMs bestimmen. Es ist wohlgemerkt jederzeit möglich, zum System neue JVMs hinzuzufügen, weswegen der von Methode `machines` gelieferte Wert nur augenblicklich Gültigkeit besitzt.[24].
Auskunft über die lokale Java Virtual Machine auf der eine Methode ausgeführt wird, gibt die Methode `thisLocation`. Sie liefert die betreffende Identifikationsnummer.

Mit der Methode `setClassDistributor` lässt sich zur Laufzeit dynamisch das Distributor-Objekt des Runtime Managers festlegen. Dieses realisiert eine Auswahlstrategie für die Lage von Klassenobjekten. Analog hierzu erlaubt es die Methode `setObjectDistributor` das Distributor-Objekt des lokalen `RuntimeEnvironments` neu zu definieren.

Mit den `setTarget`-Methoden ist es möglich, die lokale Ortsauswahlstrategie für neu zu Instantiierende *remote objects* zu beeinflussen. Die erste Variante von `setTarget` sorgt dafür, dass neue Objekte stets in der lokalen virtuellen Maschine angelegt werden. Wird bei der zweiten Methode als Argument ein positiver `int`-Wert angegeben, so bezeichnet die Angabe eine lokale JVM, auf der im folgenden stets die neuen Objekte instantiiert werden. Die letzte überladene Version von `setTarget` setzt die Zielmaschine für neue Objekte auf die lokale JVM, auf der sich das übergebene *remote object* befindet. Mit der Methode `resetTarget` kann schließlich wieder die übliche Ortsauswahlstrategie (ohne vom Benutzer explizit vorgeschlagene Bevorzugung einer lokalen JVM) eingeschaltet werden. Es muss an dieser Stelle noch erwähnt werden, dass die `setTarget`-Methoden lediglich Empfehlungen an das Laufzeitsystem darstellen. Eine Implementierung eines Distributors muss sich nicht zwangsweise an die Anweisungen halten. Der Standard-Distributor `ObjectDistributor` hält sich jedoch strikt an die, vom Programmierer vorgegebenen Richtlinien.

Die Methoden `exit` und `kill` realisieren, wie bereits in 4.2.7 erwähnt, die beiden Möglichkeiten, ein verteiltes Java-Programm „gewaltmäßig“ abzubrechen. `exit` beendet unmittelkbar die Ausführung eines verteilten Java-Programms. Die Laufzeitumgebung bleibt hiervon unbetroffen. Werden parallel zum abgebrochenen Programm weitere Programme auf der gleichen Laufzeitumgebung abgearbeitet, so sind diese vom Abbruch nicht betroffen.

[24]Man kann aber stets voraussetzen, dass die Anzahl der lokalen JVMs einer JDR-Konfiguration zunimmt, d.h. der von der Methode `machines` als Ergebnis zurückgegebene Wert, eine Mindestanzahl an virtuellen Maschinen darstellt.

`kill` dagegen beendet nicht nur das eigentliche Programm bzw. die virtuelle Maschine, über welche das Programm gestartet wurde, sondern sorgt zugleich auch dafür, dass die gesamte Laufzeitumgebung auf welcher das Programm abgearbeitet wird, ebenso auf der Stelle beendet wird.

5 Übersetzung von Java Quellcode

Als Basis für den Bau eines Laufzeitsystems für ein verteiltes Java, wurde in Abschnitt 3 bereits diskutiert, wie normale Java Klassen auf Quelltextebene transformiert werden müssen, damit Instanzen dieser Klasse in einer verteilten Umgebung interagieren können. Ein Übersetzer, der eine als `remote` deklarierte Klasse übersetzen soll, muss intern prinzipiell genau diese Transformationen durchführen und für die daraus erhaltenen Klassen Bytecode erzeugen. Der speziell für das JDR-System entwickelte Übersetzer *jdrc* [25] ist eine Erweiterung des Java Übersetzers *EspressoGrinder* [OdP_96]. *EspressoGrinder* wurde am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe entwickelt. Er ist selbst in Java geschrieben und kann anstelle von *javac* zusammen mit dem JDK[26] eingesetzt werden.

Es folgt zunächst eine kurze Beschreibung der Architektur von *EspressoGrinder*, um die Erweiterungen, die am Übersetzer vorgenommen wurden besser einordnen zu können. Details zum Aufbau und zur grundlegenden Arbeitsweise können [Zen_96] entnommen werden.

5.1 Architektur des Java Übersetzers *EspressoGrinder*

Die Arbeitsweise von *EspressoGrinder* entspricht der üblichen Vorgehensweise von Übersetzern. Die Übersetzung von Java Quelltexten wird prinzipiell in vier aufeinanderfolgenden Übersetzungsphasen durchgeführt:

1. Zunächst werden nacheinander alle zu übersetzenden Quelldateien eingelesen und zerteilt. Bei diesem Vorgang wird für jede Datei ein abstrakter Syntaxbaum (*abstract syntax tree*, AST) generiert, der die Übersetzungseinheit repräsentiert.
2. Basierend auf den Strukturbäumen findet anschließend die semantische Analyse statt. Dabei werden die ASTs vollständig attributiert; d.h. für alle Knoten werden zugehörige Attributwerte durch Auswertung von semantischen Regeln bestimmt. Hierzu sind insgesamt drei getrennte Durchläufe durch die abstrakten Syntaxbäume nötig:
   2.1. Aufsammeln aller deklarierten Klassen
   2.2. Für jede Klassendefinition Ermittlung aller Methoden- und Variablendeklarationen
   2.3. Folgende Aufgaben werden nun gleichzeitig, in einem Durchlauf, erledigt:
      - Synthetisierung des Typs aller Konstrukte im Strukturbaum
      - Typverifizierung aufbauend auf dem jeweils ermittelten Typ
      - Namensauflösung und damit zusammenhängende Überprüfungen
      - Sprungzielermittlung und Sammlung von Kontexinformationen bei Sprunganweisungen
3. Für alle in den abstrakten Syntaxbäumen enthaltenen Methoden, Konstruktoren und Initialisierungsroutinen wird direkt Bytecode erzeugt.
4. Die zu übersetzenden Klassen werden jeweils in eine eigene Klassendatei geschrieben.

Abbildung 8 zeigt schematisch die Abfolge eines Übersetzerlaufs für normalen Java Quellcode.

[25] *Java Distributed Runtime Compiler*
[26] *Java Development Kit* von Sun Microsystems, Inc.

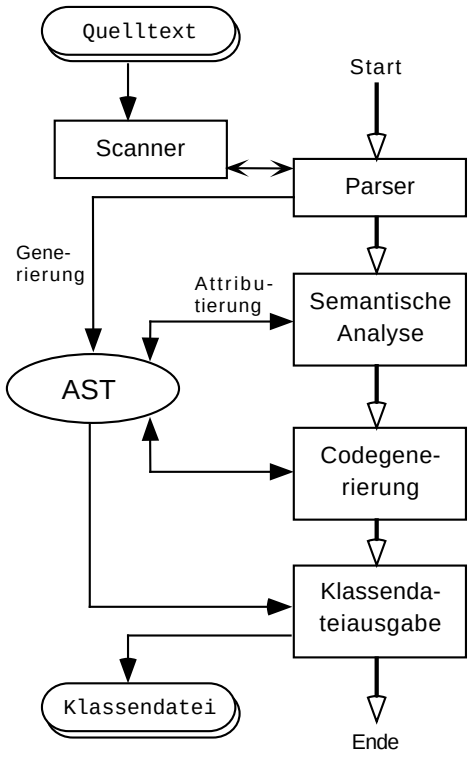


Abbildung 8: Ablauf eines Übersetzungsvorgangs in *EspressoGrinder*

Wenn man nun *EspressoGrinder* als Grundlage für eine Übersetzer hernimmt, der *remote classes* übersetzen kann, dann muss diese Abfolge an geeigneter Stelle unterbrochen werden, um die in 3 erläuterten Transformationen durchzuführen. Anschließend müssen dann die transformierten Klassen anstelle der ursprünglichen übersetzt werden. Im folgenden Abschnitt wird auf diese Methodik näher eingegangen.

5.2 Übersetzung von Klassen zur Verwendung im verteilten Laufzeitsystem

Das hier beschriebene *verteilte Java* kennt zwei verschiedene Typen von Klassen: lokale Klassen und *remote classes*. Eine Klasse wird nur dann als *remote class* übersetzt, wenn der Modifikator `remote` in der Klassendeklaration angegeben wird. Das heisst jedoch nicht, dass lokale Klassen wie sonst üblich übersetzt werden können. Lokale Klassen sollen ohne weiteres auch mit entfernten Klassen umgehen können. Der Übersetzer *jdrc* muss deswegen auch lokale Klassen anders übersetzen, als das z.B. *javac* machen würde. Wie man später noch sehen wird, ist eine modifizierte semantische Analyse notwendig und Zugriffe auf Variablen eines *remote objects* müssen über Zugriffsfunktionen abgewickelt werden.

Für die Durchführung der in 3 beschriebenen Transformationen ist relativ viel Kontextwissen notwendig. Es macht also keinen Sinn diese Umsetzungen auf Quelltextebene bzw. auf der Ebene des vom Zerteiler generierten Strukturbaums durchzuführen. Nach der Phase der semantischen Analyse im Übersetzerlauf ist jedoch der gesamte abstrakte Syntaxbaum attributiert und alle semantischen Informationen, wie z.B. Typen von Ausdrücken, oder die Zuordnung von Bezeichnern zu Variablen, Methoden bzw. Klassen, sind verfügbar. Mit dem attributierten AST lassen sich also die Transformationen im Anschluss an die semantische Analyse durchführen.

Abbildung 9 verdeutlicht die Einordnung der zusätzlichen Transformationsphase in das Übersetzungsmodell von *EspressoGrinder* (siehe auch Abbildung 8). Man beachte, dass Abbildung 9 nur einen ersten Ansatz für einen Übersetzerlauf skizziert und noch nicht das endgültige Verhalten von *jdrc* beschreibt.

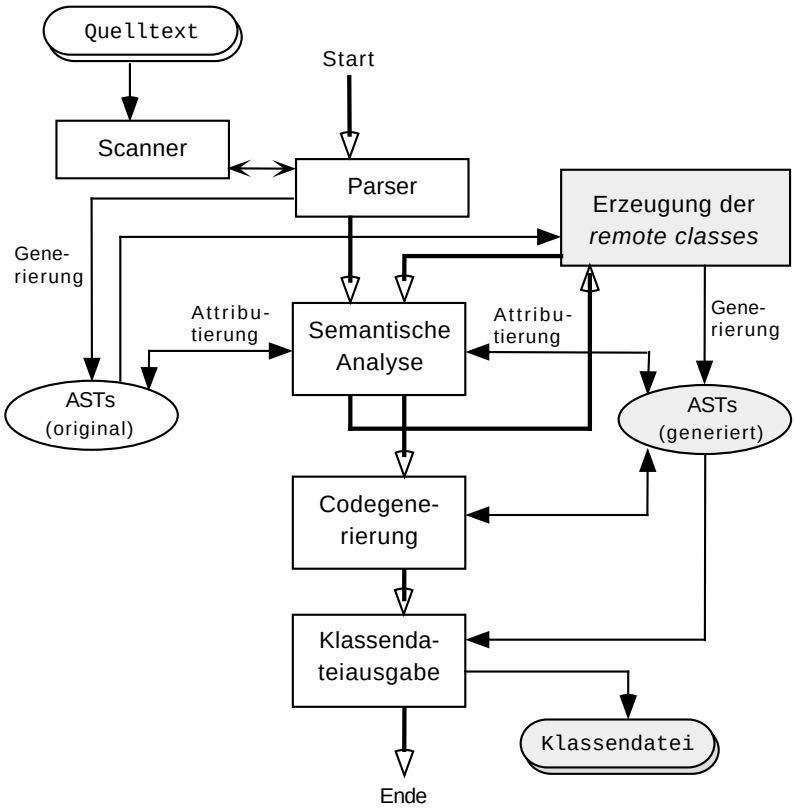


Abbildung 9: Ablauf eines Übersetzungsvorgangs in *jdrc* (vorläufig)

Gemäß Abbildung 9 werden also zunächst die originalen Quelltexte zerteilt und semantisch analysiert, ohne eine Unterscheidung zwischen lokalen Klassen und *remote classes* zu machen. Damit wird sichergestellt, dass der Übersetzer in beiden Fällen korrekt und vollständig die Klassendefinitionen auf Fehler hin untersuchen kann. Anschließend werden aus den attributierten Syntaxbäumen gemäß den Umformungsregeln neue, noch nicht attributierte Strukturbäume erzeugt. Diese Transformationsphase ist also auf der Ebene des Zerteilens anzusiedeln, da hier, unter Ausnutzung der vorhandenen semantischen Informationen, lediglich eine syntaktische Struktur erzeugt wird. Deswegen ist nun auch eine erneute semantische Analysephase notwendig, um die neuen Strukturbäume zu attributieren. Sind die AST-Umsetzungsregeln korrekt implementiert, dann können in dieser Phase diesmal keine Fehler auftreten. Nach dieser Analyse kann nun für die endgültigen Klassen Bytecode erzeugt werden, welcher anschließend in Klassendateien geschrieben wird.

Bei dieser Vorgehensweise geht die ursprüngliche Klassendefinition verloren, da nur für die umgesetzten, generierten Klassen Klassendateien erzeugt werden. Verwendet man nun aber in einem Java Quelltext eine bereits übersetzte *remote class* und übersetzt man diesen Quelltext mit *jdrc*, dann wird der Übersetzer bei der ersten semantische Analyse Probleme bekommen, da der Versuch, die Klassendatei der *remote class* zu laden, im Laden der zugehörigen Handle-Klasse, die ja den Namen der ursprünglichen Klassendefinition besitzt, resultieren würde. Da

die Signatur der Handle-Klasse nur in Bezug auf die Methoden, mit der ursprünglichen Klasse übereinstimmt, und der Übersetzer auch aus den restlichen zur *remote class* gehörenden Klassendateien, die ursprüngliche Signatur nicht wieder herstellen kann,[27] ist eine korrekte semantische Analyse in diesem Fall nicht möglich. Es wird hierzu die ursprüngliche Klassendatei bzw. zumindest die ursprüngliche Signatur der *remote class* benötigt.
Deswegen ist es erforderlich, dass *jdrc* für *remote classes* neben den generierten Klassen auch noch für die ursprüngliche Klassendeklaration eine Klassendatei erzeugt. Diese Klassendatei erfüllt einzig und allein den Zweck, dass bei der Verwendung der *remote class* in einem anderen Java Quelltext, der Übersetzer bei der ersten semantischen Analyse des Quelltextes, die ja auf den lokalen Klassendefinitionen basiert, diese ursprüngliche Klassendatei laden kann.
Für lokale Klassen ist das Erzeugen einer solchen zusätzlichen Klassendatei, die nur für den Übersetzer benötigt und niemals ausgeführt wird, nicht erforderlich. Hier werden nämlich schlimmstenfalls Methodenrümpfe abgeändert, die Signatur bleibt auf jeden Fall erhalten.

Der in Abbildung 9 skizzierte Übersetzerlauf muss also noch so abgeändert werden, dass auch zusätzlich aus den originalen attributierten Strukturbäumen Klassendateien erzeugt werden. Um Namenskonflike mit der Handle-Klasse zu vermeiden, erhalten die hierbei ausgegebenen Dateien anstelle der Namensendung `.class` die Endung `.local`. Bei der ersten semantischen Analyse versucht dann *jdrc*, sobald festgestellt wird, dass eine Klassendatei nachgeladen werden muss, zunächst eine `.local`-Datei ausfindig zu machen. Gelingt dies nicht, so wird die normale `.class`-Datei geladen, da man dann davon ausgehen kann, dass es sich bei der Klasse nicht um eine *remote class* handeln kann.
In der zweiten semantischen Analyse liegen dann alle, als `remote` deklarierten Klassendefinitionen als einen Satz von normalen Java Klassen vor, welche wie üblich übersetzt werden können. Die `.local`-Klassen werden für diesen restlichen Teil des Übersetzerlaufs nicht mehr benötigt und stören auch durch ihre Namensgleichheit mit den Handle-Klassen nur unnötig, weshalb sie nach der semantischen Analyse einfach wieder aus dem Speicher des Übersetzers entfernt werden können.

In Abbildung 10 ist jetzt der komplette Ablauf eines Übersetzungsvorgangs von *jdrc* dargestellt. Die linke Hälfte des Diagramms, also der erste Teil des Übersetzerlaufs, zeigt den Teil von *jdrc*, der dafür verantwortlich ist, dass *remote classes* ohne Einschränkungen genauso syntaktisch und semantisch überprüft werden können, wie lokale Klassen. Selbst die Tatsache, dass für die Klassendateien der *remote classes* die Modifikatoren verloren gehen, spielt für die Überprüfungen in dieser Phase keine Rolle. Die Übersetzungszeit-Zugriffsrechte können durch die Verwendung der `.local`-Klassen korrekt nachgeprüft werden.
Die rechte Hälfte des Diagramms 10 zeigt einen normalen Übersetzerlauf, wie er auch in *EspressoGrinder* abläuft. Statt der vom Parser erzeugten Strukturbäume werden hier eben die generierten abstrakten Syntaxbäume verwendet. Fehler können in diesem Teil nicht mehr auftreten.
Im letzten Übersetzungsabschnitt werden schließlich noch die Stub- und Skeleton-Klassendateien erzeugt. Hierzu ruft *jdrc* den zum RMI-Paket gehörenden Stub-/Skeleton-Übersetzer *rmic* mit allen generierten Implementierungen von *remote classes* auf. Abbildung 11 gibt abschließend einen Überblick über alle aus einer `remote` deklarierten Klassendeklaration von *jdrc* erzeugten Klassendateien.

[27] Bei der Transformation gehen teilweise Informationen verloren. Bereits an Tabelle 1 für die Umsetzung von Modifikatoren sieht man, dass die Zugriffmodifikatoren vollständig verschwinden.

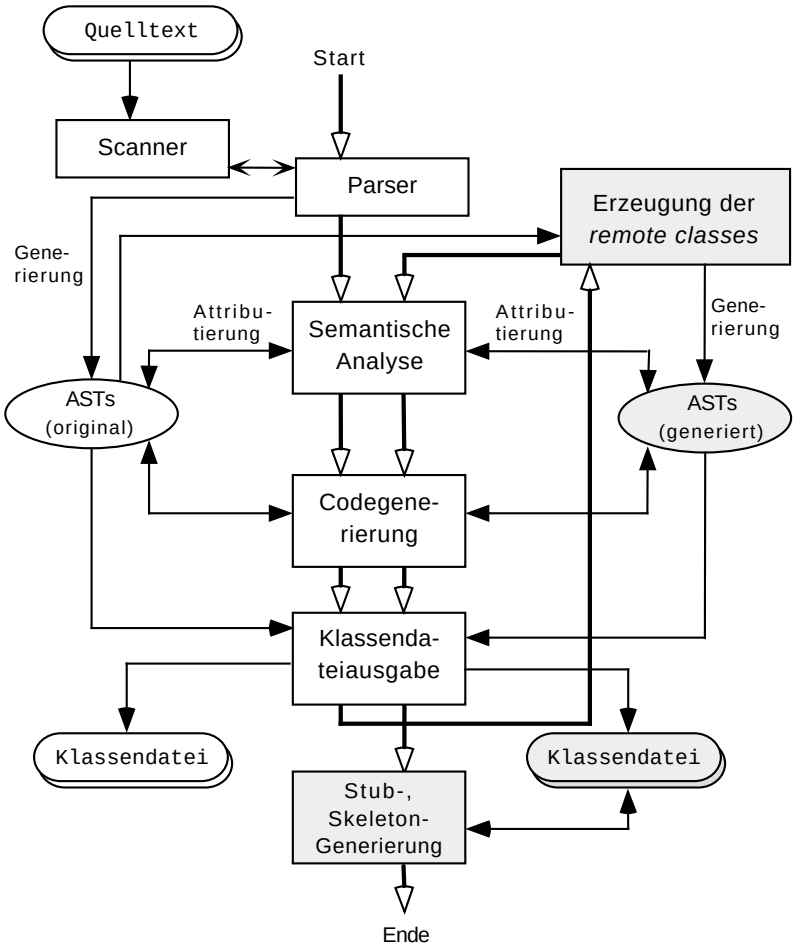


Abbildung 10: Vollständiger Übersetzungslauf in *jdrc*

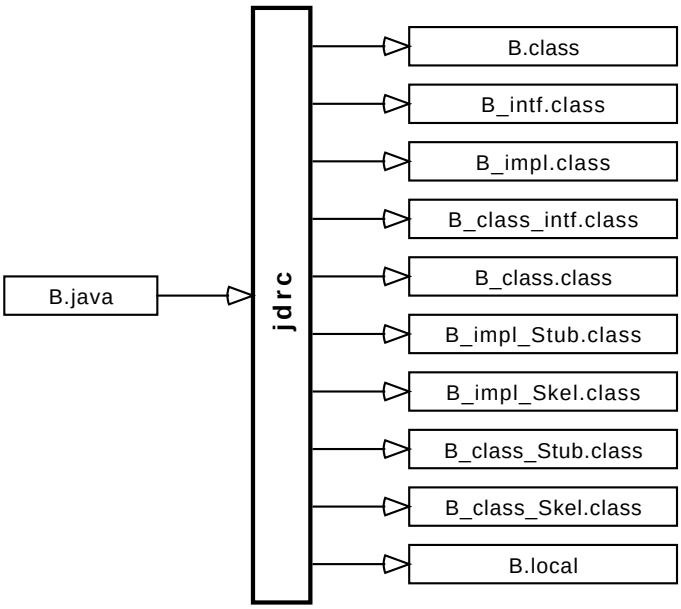


Abbildung 11: Übersicht aller, aus einer `remote` deklarierten Klasse `B` generierten Klassendateien

Anhang

A Alternative Ansätze für ein verteiltes Java

A.1 *Remote Objects In Java*

Der in dieser Arbeit vorgestellte Prototyp eines verteilten Java ist nicht der erste Versuch auf diesem Gebiet. Nataraj Nagaratnam und Arvind Srinivasan implementierten bereits ein verteiltes Java-System, welches sie *Remote Objects In Java* nennen [NSL_96]. *Remote objects* werden in ihrem System durch eine entfernte Objektinstantiierung und entfernte Methodenaufrufe unterstützt. Im Gegensatz zu JDR erweiterten Nagaratnam und Srinivasan die Sprache um ein neues Schlüsselwort `remotenew`. Dieses Schlüsselwort muß anstelle von `new` verwendet werden um ein Objekt auf einer entfernten virtuellen Maschine zu erzeugen. Wurde ein *remote object* auf dieser Weise auf einem Server erzeugt, können Methoden mit der üblichen Java-Syntax aufgerufen werden. Ausgeführt werden die Methoden wie beim Java RMI auf der entfernten Maschine.[28]

Ein *remote object* ist an die Maschine gebunden, auf der es instantiiert wurde; es gibt keine Objekt-Migration. Zustandsveränderungen des Objekts werden also vollständig auf dieser virtuellen Maschine vorgenommen. *Remote objects* besitzen eine Objekt-ID auf der lokalen Maschine, eine weitere ID auf der Heimatmaschine. Die lokale Maschine verwaltet eine Tabelle, die die Abbildung der lokalen IDs auf die entfernten Objekt-IDs beschreibt. Über diese Tabelle kann auch entschieden werden, ob ein Objekt ein *remote object* oder ein lokales Objekt ist. Bei *Remote Objects In Java* ist es nämlich ohne weiteres möglich, dass es zu einer Klasse sowohl entfernte also auch lokale Objekte gibt. Es gibt also keine Unterscheidung zwischen lokalen Klassen und *remote classes* wie im JDR-System, sondern lediglich eine Unterscheidung zwischen lokalen Objekten und *remote objects*.

Charakteristisch für *Remote Objects In Java* ist auch das Prinzip, dass der Bytecode sämtlicher Klassen anfangs lediglich auf der lokalen virtuellen Maschine bekannt ist. Eine entfernte virtuelle Maschine wartet nach der Initialisierung lediglich auf Anfragen anderer Maschinen. Zur entfernten Ausführung eines Konstruktors bzw. einer Methode muss dann stets der betreffende Bytecode mitgeschickt werden.

Ein typisches Szenario für eine enfernte Objektinstantiierung muss man sich folgendermaßen vorstellen:

1. Der Bytecode für einen Konstruktoraufruf einer Klasse[29] wird von der lokalen Maschine zu einer entfernten Maschine transferiert.

2. Die entfernte Maschine führt den Konstruktor aus, erzeugt eine (entfernte) Objekt-ID für das neu generierte Objekt und sendet dieses an den Aufrufer zurück.

[28] Das RMI-Package von Sun Microsystems wird bei *Remote Objects In Java* nicht eingesetzt.

[29] Aus [NSL_96] geht leider nicht eindeutig hervor, ob nun der Bytecode für den Konstruktorufruf oder die Konstruktorimplementierung verschickt wird. Beide Möglichkeiten sind problematisch. Wird der komplette Aufruf verschickt, so erfolgt die Auswertung der Argumente enfernt, also nicht auf der lokalen, aufrufenden Seite. Im Aufruf dürfen demnach keine lokalen Variablen u.ä. vorkommen. Das Verschicken einer einzigen Konstruktorimplementierung ist andererseits ebenfalls sinnlos, da möglicherweise andere Methoden der Klasse vom Konstruktor aus aufgerufen werden. Deren Bytecode muss hierfür ebenso bekannt sein. Es muss also im Allgemeinen sowieso der Bytecode für eine gesamte Klasse auf der Server-Seite verfügbar sein.

3. Die lokale Maschine trägt das neue Objekt bzw. dessen ID in ihren Tabellen ein.

4. Die lokale virtuelle Maschine fährt mit der nächsten Instruktion nach dem Konstruktoraufruf fort.

Entfernte Methodenaufrufe werden nach dem gleichen Prinzip abgewickelt. Hier entfallen jedoch Schritt 2 und 3. In Schritt 1 wird statt des Bytecodes für den Konstruktoraufruf, der Code für den Aufruf der Methode verschickt.

Remote Objects In Java unterstützt keine *Object Serialization*. Es können also keine Objekte bei entfernten Methodenaufrufen übergeben werden. Lediglich Basistypen sind als Argumente zulässig.[30] Durch diese Beschränkung entfallen natürlich auch die Probleme des RMI, die mit der Wertübergabe lokaler Objekte bei entfernten Methodenaufrufen zusammenhängen. Allerdings ergibt sich an dieser Stelle auch ein großes Problem: dadurch dass erst zur Laufzeit bekannt ist, ob ein Objekt ein *remote object* oder ein lokales Objekt ist,[31] kann auch erst zur Laufzeit entschieden werden, ob ein Methodenaufruf (bei dem auch Argumente eines Nicht-Basistyps beteiligt sind) bezüglich eines Objekts überhaupt möglich ist. Faktisch gehen hiermit die Vorteile, die sich aus der starken Typisierbarkeit Javas ergeben, vollständig verloren. Dies ist aber gerade eine der hervorstechenden Eigenschaften Javas.

In diesem Zusammenhang erweist sich das Konzept, dass von einer einzigen Klasse sowohl lokale als auch entfernte Instanzen angelegt werden können, für die Praxis als relativ ungeeignet. Selbst wenn eine Object Serialization unterstützt werden würde, wüsste der Anwender niemals, ob er es mit einem entfernten Objekt (und damit mit einer anderen Objekt-Semantik) oder mit einem lokalen Objekt zu tun hätte.[32]

Der Zugriff auf Variablen entfernter Objekt, die Einbettung von Exceptions sowie die Notwendigkeit einer Anpassung der Klassen-Semantik an ein verteiltes System, insbesonders der Umgang statischen Variablen und Methoden, werden in [NSL_96] nicht diskutiert.

A.2 Implementation von *Remote Objects In Java*

Wie beim JDR-System sind zur Realisierung des verteilten Objektmodells ein speziell angepasster Übersetzer und ein eigenes Laufzeitsystem notwendig. Der Übersetzer projeziert hier jedoch nicht die Spracherweiterungen auf normalen Java-Code, sondern generiert beispielsweise für einen `remotenew`-Aufruf einen eigenen Bytecode-Opcode. Da folglich die erzeugten Klassen nicht auf einem üblichen Java Bytecode-Interpreter abgearbeitet werden können, musste der bestehende Interpreter erweitert werden, um mit dem zusätzlichen Sprachschatz umgehen zu können.
Das Laufzeitsystem für *Remote Objects In Java* konnte damit gleich direkt in die virtuelle Maschine integriert werden. Bereits die Idee, Bytecode zu verschicken, der dann auf einer anderen Maschine ausgeführt wird, hätte eine Modifikation der Java Virtual Machine erzwungen.

[30] *Remote Objects In Java* kann insbesondere deswegen auch nicht mit Exceptions umgehen.

[31] Durch die Unterscheidung zwischen lokalen und entfernten Klassen, kann beim JDR-System bereits zur Übersetzungszeit über den Typ einer Variable festgestellt werden, ob ein lokales oder entferntes Objekt referenziert wird. Dies verleiht auch dem Programmierer die nötige Sicherheit, jederzeit zu wissen, ob ein Objekt mit einer lokalen oder entfernten Objekt-Semantik behaftet ist.

[32] Das JDR-System wurde so entworfen, dass statisch – ausgehend vom Programmcode – entschieden werden kann, ob ein Methodenaufruf mit einer entfernten oder einer lokalen Semantik versehen ist. In Abschnitt 3.6.4 werden die Kriterien ausführlich erläutert.

Durch die Integration der *remote object*-Unterstützung in die virtuelle Maschine entfällt natürlich auch die Notwendigkeit von Stubs und Skeletons (sowie viele weitere umständliche Konstruktionen im JDR-System, die nur deswegen notwendig sind, weil zur Übersetzungszeit nicht alle notwendigen Informationen bekannt sind). Deren Aufgaben werden von der Maschine direkt übernommen. Damit wird implizit die Aufgabenlösung, beim Vergleich mit dem JDR-System, von der Übersetzungszeit in die Laufzeit verschoben. Neben der Flexibilität besitzt dieser Ansatz einen weiteren Vorteil: pro Klasse muss vom Übersetzer nur eine einzige Klassendatei erzeugt werden.

B Implementation

B.1 Übersicht

Mit der Implementation des JDR-Systems sollte gezeigt werden, dass die in dieser Arbeit vorgestellten Ideen durchaus realisierbar sind. Das entwickelte Packet baut auf dem Java Development Kit 1.0.2 und der Java Remote Method Invocation für Java 1.0.2 auf. Sowohl das Laufzeitsystem als auch der Übersetzer wurden vollständig in Java geschrieben. Zur Übersicht über die Strukturierung der Implementation folgt eine hierarchische Auflistung aller Packages (fettgedruckt) und Klassen die zu dem System gehören:

jdr
- **compiler**
  - . . . *EspressoGrinder*-Klassen. . .
  - Remote
  - Transform
  - PrettyPrinter
- **env**
  - ClassDistributor
  - DistributedJVM
  - Distributor
  - JVM
  - LocalJVM
  - LocateRuntimeManager
  - ObjectCopyStream
  - ObjectDistributor
  - RemoteExitException
  - RTManager
  - RuntimeManager
  - RuntimeEnvironment
  - RuntimeExplorer
- **lang**
  - ArrayIndex
  - Closure
  - JDRConstants
  - MessageNotUnderstood
  - MovedException
  - ObjectHandle
  - RemoteTools
  - RemoteClass
  - RemoteClassIntf
  - RemoteInstance
  - RemoteIntf
  - RemoteThread
  - RemoteThread_intf
  - RemoteThread_impl
  - RemoteThread_class
  - RemoteThread_class_impl
  - RuntimeRemoteException
- **util**
  - RemoteDictionary
  - RemoteHashtable
  - RemoteMonitor
  - RemoteSignal
  - RemoteStack
  - RemoteStringTokenizer
  - RemoteVector

java
- **lang**
  - DistributedRuntime
  - RemoteObject
  - Resident

Es folgt nun eine kurze Erläuterung der Bedeutung aller wichtigen Klassen des JDR-Systems. Details können den dokumentierten Quelltexten entnommen werden.

B.2 `jdr.compiler` Package

Diese Package enthält die Klassen des Übersetzers *jdrc*, welcher Klassendefinitionen für das verteilte Java Laufzeitsystem übersetzt. Da *jdrc* ausgehend von dem Java-Übersetzer *EspressoGrinder* entwickelt wurde, entspricht die interne Struktur weitgehend der von *EspressoGrinder* [Zen_96].

Remote: Umsetzung von Klassen-, Methoden- und Variablendefinitionen. Insbesondere wird hier die Struktur von *remote classes* erzeugt.

Transform: Realisierung der in Kapitel 3 beschriebenen Transformationsfunktion *toRemote*. Für jeden möglichen AST-Knoten wird eine geeignete Umsetzungsmethode zur Verfügung gestellt.

PrettyPrinter: Ausgaberoutinen, die einen Java Quelltext in möglichst lesbarer Form in eine Datei schreiben.

B.3 `jdr.env` Package

Mit der Package `jdr.env` werden alle Komponenten der in Kapitel 4 beschriebenen verteilten Laufzeitumgebung realisiert.

ClassDistributor: Distributor-Klasse mit der die voreingestellte Strategie zur Klassenobjektverteilung realisiert wird.

DistributedJVM: Klasse, die speziell zur Ausführung verteilter Java-Programme verwendet werden kann (*djava*). Im Gegensatz zum Aufruf über eine normale lokale virtuelle Maschine, überprüft diese Klasse zunächst eine JDR-Laufzeitumgebung, bevor mit der eigentlichen Ausführung einer *remote class* begonnen wird.

Distributor: Oberklasse aller Distributoren (sowohl zur *remote object*- als auch zur Klassenobjektverteilung).

LocalJVM: Implementation einer lokalen JVM einer JDR-Laufzeitumgebung.

LocateRuntimeManager: Verwaltung verschiedener Java-Laufzeitumgebungen. Es wird eine JDR-Konfigurationsdatei eingelesen und ausgehend von den Daten wird ein Stub für den entsprechenden Runtime Manager ermittelt.

ObjectDistributor: Distributor-Klasse, die standardmäßig in den `RuntimeEnvironments` zur Verteilung der *remote objects* eingesetzt wird.

RuntimeManager: Implementation des Runtime Managers.

RuntimeEnvironment: Schnittstellen-Klasse, die von einer lokalen virtuellen Maschine aus Zugriffe auf eine verteilte Java-Laufzeitumgebung ermöglicht.

RuntimeExplorer: Interaktives Werkzeug zur Untersuchung einer JDR-Laufzeitumgebung. Ausserdem kann über diese Klasse eine Laufzeitumgebung beendet oder zurückgesetzt werden.

B.4 `jdr.lang` Package

Die Package `jdr.lang` enthält sämtliche Klassen, die zum Aufbau von entfernten Klassenhierarchien benötigt werden. Insbesonders sind hier, als „Gegenstücke“ zu den lokalen Basisklassen `java.lang.Object` und `java.lang.Class` die analogen entfernten Klassen `RemoteInstance` und `RemoteClas` für das verteilte Java-System zu finden.

ArrayIndex: Diese Klasse erlaubt die Konstruktion einer unbegrenzten Index-Liste. Sie wird bei Arrayzugriffen über `java.lang.Object`-Variablen vom Übersetzer eingesetzt (näheres siehe Abschnitt 3.5.3).

JDRConstants: Definition globaler Konstanten des JDR-Systems.

MovedException: Exception die automatisch von einem „veralteten“ *remote object* bei einem entfernten Methodenaufruf ausgelöst wird. Ein Objekt ist dann veraltet, wenn es nach einer Objekt-Migration lediglich die Funktion eines Verweises auf das aktuelle *remote object* hat. Einer `MovedException` wird stets ein Stub auf das umgezogene *remote object* mitgeliefert.

ObjectHandle: Oberklasse aller Handles. Hier wird die Semantik aller überschreibbaren Methoden aus der Klasse `java.lang.Object` bzw. `java.lang.RemoteObject` an das verteilte Laufzeitsystem angepaßt.

RemoteTools: Statische Hilfsmethoden, die im, vom Übersetzer *jdrc* generierten Code, Verwendung finden.

RemoteClass: Oberklasse aller *remote class*-Implementationen.

RemoteInstance: Oberklasse aller *remote object*-Implementationen.

RemoteThread: Handle-Klasse für *remote threads*.

RuntimeRemoteException: Runtime-Exception die nur dann ausgelöst wird, wenn ein interner Fehler der nicht behoben werden auftritt. `RuntimeRemoteExceptions` führen immer zum Abbruch des betreffenden verteilten Java-Programms. Die Laufzeitumgebung läuft jedoch weiter.

B.5 `jdr.util` Package

Diese Package enthält eine kleine Klassenbibliothek nützlicher *remote classes*. Neben einigen Klassen die bereits aus der Package `java.util` bekannt sind, finden sich hier auch Klassen, die für einfache Synchronisationsanforderungen im JDR-System verwendet werden können.

RemoteMonitor: Realisierung eines gegenseitigen Ausschlusses der innerhalb einer gesamten JDR-Laufzeitumgebung eingehalten wird.

RemoteSignal: Entfernte Signalklasse. Diese erlaubt vor allem einfache Thread-Synchronisationen ohne in Konflik mit den in 3.6.6 beschriebenen Problemsituationen zu geraten.

C Beispielprogramm

Die in Kapitel 3 vorgestellten Transformationen, mit denen entfernte Klassen in normale Java Klassen übersetzt werden, wurden stets nur an kleinen, konstruierten Beispielen verdeutlicht. Zur Veranschaulichung des Zusammenwirkens der einzelnen Transformationen wird nun ein etwas größeres praxisnäheres Beispiel betrachtet.

Das folgende Beispielprogramm berechnet verteilt arithmetische Ausdrücke, die als Argument beim Programmaufruf angegeben werden. Bei den Ausdrücken dürfen lediglich natürliche Zahlen als Operanden und die Operatoren `+` und `*` verwendet werden. Klammern dürfen beliebig gesetzt werden. Leerzeichen werden nicht akzeptiert.

Im Programm werden drei Klassen `CalcException`, `Node` und `Calc` definiert. Die Exception `CalcException` wird stets dann ausgelöst, wenn ein Fehler beliebiger Art innerhalb der Berechnung auftritt. Mit Objekten der *remote class* `Node` wird ein Strukturbaum des zu berechnenden arithmetischen Ausdrucks aufgebaut. Die Zerteilung des als String vorliegenden Ausdrucks erfolgt in der „Hauptklasse“ `Calc`.

```
package example;

class CalcException extends Exception
{}

remote class Node
{
    public static int   nodes = 0;
    private int         opcode;
    private Node        left, right;
    Node(int value)
    {
        this(value,null,null);
    }
    Node(int opcode, Node left, Node right)
    {
        this.opcode = opcode;
        this.left = left;
        this.right = right;
        nodes++;
    }
    int calc() throws CalcException
    {
        if (opcode >= 0) // positive opcode represent values
            return opcode;
        switch (opcode)
        {
            case -1: return left.calc() + right.calc();
            case -2: return left.calc() * right.calc();
            default: throw new CalcException();
        }
    }
}

public remote class Calc
{
    static int pos;
    static int readDigits(String str)
    {
        int n = 0;
        try
        {
            while (Character.isDigit(str.charAt(pos)))
                n = n * 10 + str.charAt(pos++) - 48;
        }
```

```
        catch (IndexOutOfBoundsException e) {}
        return n;
    }
    static Node readExpr0(String str) throws CalcException
    {
        try
        {
            if (str.charAt(pos) == '(')
            {
                pos++;
                Node res = readExpr1(str);
                if (str.charAt(pos++) != ')')
                    throw new CalcException();
                return res;
            }
            else
            if (Character.isDigit(str.charAt(pos)))
                return new Node(readDigits(str));
        }
        catch (IndexOutOfBoundsException e) {}
        throw new CalcException();
    }
    static Node readExpr1(String str) throws CalcException
    {
        Node    res = readExpr0(str);
        try
        {
            while (str.charAt(pos) == '*')
            {
                pos++;
                res = new Node(-2,res,readExpr0(str));
            }
            if (str.charAt(pos) == '+')
            {
                pos++;
                res = new Node(-1,res,readExpr1(str));
            }
        }
        catch (IndexOutOfBoundsException e) {}
        return res;
    }
    public static void main(String[] args)
    {
        if (args.length == 0)
            DistributedRuntime.exit(0);
        Node.nodes = 0;
        pos = 0;
        try
        {
            System.out.println("result: " + readExpr1(args[0]).calc());
            System.out.println("nodes: " + Node.nodes);
        }
        catch (CalcException e) { System.out.println("execution error"); }
    }
}
```

Es folgt nun der vollständige Quellcode der transformierten Klassen. Die Exception-Klasse `CalcException` ist lokal, wird also direkt übernommen. Die beiden anderen Klassen müssen in Klassen- und Instanzenanteil zerlegt werden.

```
package example;

import java.rmi.RemoteException;
import java.rmi.Remote;
import jdr.lang.*;
import jdr.env.*;

class CalcException extends Exception
```

```
{}
...
```

Im *remote interface* `Node_intf` des Instanzenanteils der Klasse `Node` werden hauptsächlich Zugriffsmethoden für die Instanzvariablen deklariert. Die Variable `opcode` ist vom Typ `int`, weswegen zusätzlich zu den Schreib- und Lesemethoden eine Inkrementmethode generiert wird. Würde der Übersetzer feststellen, dass auf `opcode` von keiner statischen Methode der Klasse `Node` zugegriffen wird (wie das im Beispiel der Fall ist), so könnte er auf das Generieren dieser Methode durchaus verzichten. Da `opcode` `private` deklariert wurde, ist dies nämlich der einzige Grund, der eine solche Zugriffsmethode nötig machen könnte.

```
...
public interface Node_intf extends RemoteIntf
{
    int _get_example_Node_opcode() throws RemoteException, MovedException;
    int _set_example_Node_opcode(int _x) throws RemoteException, MovedException;
    int _inc_example_Node_opcode(int _x, boolean _postfix) throws RemoteException, MovedException;
    Node _get_example_Node_left() throws RemoteException, MovedException;
    Node _set_example_Node_left(Node _x) throws RemoteException, MovedException;
    Node _get_example_Node_right() throws RemoteException, MovedException;
    Node _set_example_Node_right(Node _x) throws RemoteException, MovedException;
    public int calc() throws CalcException, RemoteException, MovedException;
}
...
```

In der Implementationsklasse des Instanzenanteils von Klasse `Node` werden zunächst alle Zugriffsmethoden, gefolgt von den benutzerdefinierten Methoden vereinbart. Am Ende der Klassendefinition erfolgt die Deklaration einer Methode `_int`, welche automatisch vom Laufzeitsystem bei einer Objektinstantiierung bzw. einer Objekt-Migration aufgerufen wird. Sie initialisiert unter anderem den `this`-Ersatz `_this` (siehe Abschnitt 3.6.2). Mit der generierten Methode `_deref` wird die in Abschnitt 3.2.2 eingeführte Dereferenzierungsmethode realisiert.

```
...
public class Node_impl extends RemoteInstance implements Node_intf
{
    private int  opcode;
    public final int _get_example_Node_opcode() throws RemoteException, MovedException
    {
        _enter();
        try { return opcode; }
        finally {synchronized (LocalJVM.lock) {_calledMethods--;}}
    }
    public final int _set_example_Node_opcode(int _x) throws RemoteException, MovedException
    {
        _enter();
        try { return opcode = _x; }
        finally {synchronized (LocalJVM.lock) {_calledMethods--;}}
    }
    public final int _inc_example_Node_opcode(int _x, boolean _postfix) throws RemoteException, MovedException
    {
        _enter();
        try
        {
            int _e = opcode;
            opcode += _x;
            if (_postfix) return _e; else return opcode;
        }
        finally {synchronized (LocalJVM.lock) {_calledMethods--;}}
    }
    private Node     left;
    public final Node _get_example_Node_left() throws RemoteException, MovedException
    {
        _enter();
        try { return left; }
        finally {synchronized (LocalJVM.lock) {_calledMethods--;}}
    }
    public final Node _set_example_Node_left(Node _x) throws RemoteException, MovedException
    {
        _enter();
        try { return left = _x; }
        finally {synchronized (LocalJVM.lock) {_calledMethods--;}}
    }
    private Node     right;
    public final Node _get_example_Node_right() throws RemoteException, MovedException
    {
```

```
        _enter();
        try { return right; }
        finally { synchronized (LocalJVM.lock) {_calledMethods--;}}
    }
    public final Node _set_example_Node_right(Node _x) throws RemoteException, MovedException
    {
        _enter();
        try { return right = _x; }
        finally {synchronized (LocalJVM.lock) {_calledMethods--;}}
    }
    public Node_impl(int value) throws RemoteException
    {
        this(value, null, null);
        try {} catch (MovedException _e) {}
    }
    public Node_impl(int opcode, Node left, Node right) throws RemoteException
    {
        super();
        try
        {
            this.opcode = opcode;
            this.left = left;
            this.right = right;
            Node._inc_example_Node_nodes(1, false);
        }
        catch (MovedException _e) {}
    }
    public int calc() throws CalcException, RemoteException, MovedException
    {
        _enter();
        try
        {
            if (opcode >= 0)
                return opcode;
            switch (opcode)
            {
                case -1: return left.calc() + right.calc();
                case -2: return left.calc() * right.calc();
                default: throw new CalcException();
            }
        }
        finally {synchronized (LocalJVM.lock) {_calledMethods--;}}
    }
    public synchronized void _init() throws RemoteException
    {
        __init();
        _this = Node._handleToNode(this);
    }
    public void _deref(int _x) throws RemoteException
    {
        super._deref(_x);
        if (left != null)  left._deref(_x);
        if (right != null) right._deref(_x);
    }
}
...
```

Nun folgt die Realisierung des statischen Klassenanteils der Klasse `Node`. Außer den Zugriffsfunktionen für die statische Variable `nodes` und den beiden Konstruktormethoden wird hier nur noch die interne Methode `_init` benötigt. Sie wird automatisch beim Anlegen eines Klassenobjekts für die Klasse `Node` aufgerufen.

```
public interface Node_class extends RemoteClassIntf
{
    int _get_example_Node_nodes() throws RemoteException;
    int _set_example_Node_nodes(int _x) throws RemoteException;
    int _inc_example_Node_nodes(int _x, boolean _postfix) throws RemoteException;
    public Node_intf _new(int value) throws RemoteException;
    public Node_intf _new(int opcode, Node left, Node right) throws RemoteException;
}

public final class Node_class_impl extends RemoteClass implements Node_class
{
    public  Node_class_impl() throws RemoteException
    {}
    public int       nodes;
    public final int _get_example_Node_nodes() throws RemoteException
    {
        return nodes;
    }
    public final int _set_example_Node_nodes(int _x) throws RemoteException
    {
        return nodes = _x;
    }
    public final int _inc_example_Node_nodes(int _x, boolean _postfix) throws RemoteException
    {
```

```
        int _e = nodes;
        nodes += _x;
        if (_postfix) return _e; else return nodes;
    }
    public Node_intf _new(int value) throws RemoteException
    {
        Node_impl   _x = new Node_impl(value);
        synchronized (LocalJVM.lock) {_x._calledMethods --;}
        return _x;
    }
    public Node_intf _new(int opcode, Node left, Node right) throws RemoteException
    {
        Node_impl        _x = new Node_impl(opcode, left, right);
        synchronized (LocalJVM.lock) {_x._calledMethods --;}
        return _x;
    }
    protected void _init()
    {
        nodes = 0;
    }
}
...
```

Über die Handle-Klasse der Klasse `Node` erfolgen alle entfernten Methodenaufrufe und Variablenzugriffe. Der zusätzliche Konstruktor ganz am Anfang der Klassendefinition wird zusammen mit der Methode `_handleToNode` nur zum Erzeugen der `_this`-Handle in der `_init`-Methode der Klasse `Node_impl` benötigt.

```
...
class Node extends ObjectHandle
{
    protected Node(Remote _x)
    {
        super(_x);
    }
    public static Node _handleToNode(Node_intf _x)
    {
        return new Node(_x);
    }
    public static final int _get_example_Node_nodes()
    {
        while (true)
            try { return ((Node_class)RuntimeEnvironment.getClassObj("example.Node"))._get_example_Node_nodes(); }
            catch (RemoteException _e) {_handleRemoteException("example.Node._get_example_Node_nodes", _e);}
    }
    public static final int _set_example_Node_nodes(int _x)
    {
        while (true)
            try { return ((Node_class)RuntimeEnvironment.getClassObj("example.Node"))._set_example_Node_nodes(_x); }
            catch (RemoteException _e) {_handleRemoteException("example.Node._set_example_Node_nodes", _e);}
    }
    public static final int _inc_example_Node_nodes(int _x, boolean _postfix)
    {
        while (true)
            try { return ((Node_class)RuntimeEnvironment.getClassObj("example.Node"))._inc_example_Node_nodes(_x, _postfix); }
            catch (RemoteException _e) {_handleRemoteException("example.Node._inc_example_Node_nodes", _e);}
    }
    final int _get_example_Node_opcode()
    {
        while (true)
            try { return ((Node_intf)ref)._get_example_Node_opcode(); }
            catch (MovedException _e)  {_adaptRef(_e);}
            catch (RemoteException _e) {_handleRemoteException("example.Node._get_example_Node_opcode", _e);}
    }
    final int _set_example_Node_opcode(int _x)
    {
        while (true)
            try { return ((Node_intf)ref)._set_example_Node_opcode(_x); }
            catch (MovedException _e)  {_adaptRef(_e);}
            catch (RemoteException _e) {_handleRemoteException("example.Node._set_example_Node_opcode", _e);}
    }
    final int _inc_example_Node_opcode(int _x, boolean _postfix)
    {
        while (true)
            try { return ((Node_intf)ref)._inc_example_Node_opcode(_x, _postfix); }
            catch (MovedException _e)  {_adaptRef(_e);}
            catch (RemoteException _e) {_handleRemoteException("example.Node._inc_example_Node_opcode", _e);}
    }
    final Node _get_example_Node_left()
    {
        while (true)
            try { return ((Node_intf)ref)._get_example_Node_left(); }
            catch (MovedException _e)  {_adaptRef(_e);}
            catch (RemoteException _e) {_handleRemoteException("example.Node._get_example_Node_left", _e);}
    }
    final Node _set_example_Node_left(Node _x)
```

```
    {
        while (true)
            try { return ((Node_intf)ref)._set_example_Node_left(_x); }
            catch (MovedException _e)  {_adaptRef(_e);}
            catch (RemoteException _e) {_handleRemoteException("example.Node._set_example_Node_left", _e);}
    }
    final Node _get_example_Node_right()
    {
        while (true)
            try { return ((Node_intf)ref)._get_example_Node_right(); }
            catch (MovedException _e)  {_adaptRef(_e);}
            catch (RemoteException _e) {_handleRemoteException("example.Node._get_example_Node_right", _e);}
    }
    final Node _set_example_Node_right(Node _x)
    {
        while (true)
            try { return ((Node_intf)ref)._set_example_Node_right(_x); }
            catch (MovedException _e)  {_adaptRef(_e);}
            catch (RemoteException _e) {_handleRemoteException("example.Node._set_example_Node_right", _e);}
    }
    Node(int value)
    {
        super(nullHandle);
        try { ref = ((Node_class)RuntimeEnvironment.getConstrObj("example.Node"))._new(value); }
        catch (RemoteException _e) {_handleInstantException("example.Node", _e);}
    }
    Node(int opcode, Node left, Node right)
    {
        super(nullHandle);
        try { ref = ((Node_class)RuntimeEnvironment.getConstrObj("example.Node"))._new(opcode, left, right); }
        catch (RemoteException _e) {_handleInstantException("example.Node", _e);}
    }
    int calc() throws CalcException
    {
        while (true)
            try { return ((Node_intf)ref).calc(); }
            catch (MovedException _e)  {_adaptRef(_e);}
            catch (RemoteException _e) {_handleRemoteException("example.Node.calc", _e);}
    }
}
...
```

Der Instanzenanteil der `Calc`-Klasse ist leer. Daher sind auch in der Implementationsklasse `Calc_impl` lediglich generierte Methoden für administrative Zwecke vorzufinden.

```
...
public interface Calc_intf extends RemoteIntf
{}

public class Calc_impl extends RemoteInstance implements Calc_intf
{
    public Calc_impl() throws RemoteException
    {
        super();
        try {} catch (MovedException _e) {}
    }
    public synchronized void _init() throws RemoteException
    {
        __init();
        _this = Calc._handleToCalc(this);
    }
    public void _deref(int _x) throws RemoteException
    {
        super._deref(_x);
    }
}

public interface Calc_class extends RemoteClassIntf
{
    public Calc_intf _new() throws RemoteException;
    int _get_example_Calc_pos() throws RemoteException;
    int _set_example_Calc_pos(int _x) throws RemoteException;
    int _inc_example_Calc_pos(int _x, boolean _postfix) throws RemoteException;
    public int readDigits(String str) throws RemoteException;
    public Node readExpr0(String str) throws CalcException, RemoteException;
    public Node readExpr1(String str) throws CalcException, RemoteException;
    public void main(String[] args) throws RemoteException;
}

public final class Calc_class_impl extends RemoteClass implements Calc_class
{
    public Calc_class_impl() throws RemoteException
    {}
    public Calc_intf _new() throws RemoteException
    {
        Calc_impl        _x = new Calc_impl();
        synchronized (LocalJVM.lock) {_x._calledMethods --;}
        return _x;
```

```
    }
    int pos;
    public final int _get_example_Calc_pos() throws RemoteException
    {
        return pos;
    }
    public final int _set_example_Calc_pos(int _x) throws RemoteException
    {
        return pos = _x;
    }
    public final int _inc_example_Calc_pos(int _x, boolean _postfix) throws RemoteException
    {
        int          _e = pos;
        pos += _x;
        if (_postfix) return _e; else return pos;
    }
    public int readDigits(String str) throws RemoteException
    {
        int          n = 0;
        try
        {
            while (Character.isDigit(str.charAt(pos)))
                n = n * 10 + str.charAt(pos ++) - 48;
        }
        catch (IndexOutOfBoundsException e) {}
        return n;
    }
    public Node readExpr0(String str) throws CalcException, RemoteException
    {
        try
        {
            if (str.charAt(pos) == 40)
            {
                ++ pos;
                Node             res = readExpr1(str);
                if (str.charAt(pos ++) != 41)
                    throw new CalcException();
                return res;
            }
            else
            if (Character.isDigit(str.charAt(pos)))
                return new Node(readDigits(str));
        }
        catch (IndexOutOfBoundsException e) {}
        throw new CalcException();
    }
    public Node readExpr1(String str) throws CalcException, RemoteException
    {
        Node             res = readExpr0(str);
        try
        {
            while (str.charAt(pos) == 42)
            {
                ++ pos;
                res = new Node(-2, res, readExpr0(str));
            }
            if (str.charAt(pos) == 43)
            {
                ++ pos;
                res = new Node(-1, res, readExpr1(str));
            }
        }
        catch (IndexOutOfBoundsException e) {}
        return res;
    }
    public void main(String[] args) throws RemoteException
    {
        if (args.length == 0)
            DistributedRuntime.exit(0);
        Node._set_example_Node_nodes(0);
        pos = 0;
        try
        {
            System.out.println("result: " + readExpr1(args[0]).calc());
            System.out.println("nodes: " + Node._get_example_Node_nodes());
        }
        catch (CalcException e) { System.out.println("execution error"); }
    }
    protected void _init()
    {}
}

public class Calc extends ObjectHandle
{
    protected Calc(Remote _x)
    {
        super(_x);
    }
    public static Calc _handleToCalc(Calc_intf _x)
    {
        return new Calc(_x);
    }
```

```
    public  Calc()
    {
        super(nullHandle);
        try { ref = ((Calc_class)RuntimeEnvironment.getConstrObj("example.Calc"))._new(); }
        catch (RemoteException _e) {_handleInstantException("example.Calc", _e);}
    }
    static final int _get_example_Calc_pos()
    {
        while (true)
            try { return ((Calc_class)RuntimeEnvironment.getClassObj("example.Calc"))._get_example_Calc_pos(); }
            catch (RemoteException _e) {_handleRemoteException("example.Calc._get_example_Calc_pos", _e);}
    }
    static final int _set_example_Calc_pos(int _x)
    {
        while (true)
            try { return ((Calc_class)RuntimeEnvironment.getClassObj("example.Calc"))._set_example_Calc_pos(_x); }
            catch (RemoteException _e) {_handleRemoteException("example.Calc._set_example_Calc_pos", _e);}
    }
    static final int _inc_example_Calc_pos(int _x, boolean _postfix)
    {
        while (true)
            try { return ((Calc_class)RuntimeEnvironment.getClassObj("example.Calc"))._inc_example_Calc_pos(_x, _postfix); }
            catch (RemoteException _e) {_handleRemoteException("example.Calc._inc_example_Calc_pos", _e);}
    }
    static int readDigits(String str)
    {
        while (true)
            try { return ((Calc_class)RuntimeEnvironment.getClassObj("example.Calc")).readDigits(str); }
            catch (RemoteException _e) {_handleRemoteException("example.Calc.readDigits", _e);}
    }
    static Node readExpr0(String str) throws CalcException
    {
        while (true)
            try { return ((Calc_class)RuntimeEnvironment.getClassObj("example.Calc")).readExpr0(str); }
            catch (RemoteException _e) {_handleRemoteException("example.Calc.readExpr0", _e);}
    }
    static Node readExpr1(String str) throws CalcException
    {
        while (true)
            try { return ((Calc_class)RuntimeEnvironment.getClassObj("example.Calc")).readExpr1(str); }
            catch (RemoteException _e) {_handleRemoteException("example.Calc.readExpr1", _e);}
    }
    public static void main(String[] args)
    {
        while (true)
            try { ((Calc_class)RuntimeEnvironment.getClassObj("example.Calc")).main(args); return; }
            catch (RemoteException _e) {_handleRemoteException("example.Calc.main", _e);}
    }
}
```

Literatur

[ASU_92] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilerbau*. Addison-Wesley, 1992.

[BCL_94] Umesh Bellur, Gary Craig, Doug Lea. *Clustering: Composition for Active Object Systems*. Proc. of the 27th HICSS, 1994.

[CBS_93] Gary Craig, Umesh Bellur, Kevin Shank. *Clusters: A Pragmatic Approach Towards Supporting a Fine Grained Active Object Model In Distributed Systems*. Proc. of the 9th ICSE, Las Vegas, NV, 1993.

[Fla_96] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates Inc., 1996.

[GJS_96] James Gosling, Bill Joy, Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Kin_96] Joseph R. Kiniry. *Distributed Java*. 1996.
`http://csvax.cs.caltech.edu/~kiniry/projects/Research_Papers/OSF_Papers`

[NSL_96] Nataraj Nagaratnam, Arvind Srinivasan, Doug Lea. *Remote Objects In Java*. Syracuse University, 1996.

[OdP_96] Martin Odersky, Michael Philippsen. *EspressoGrinder Quelltext*. University of Karlsruhe, Germany, 1996.

[OMG_95] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.0, 1995.

[RMI_96] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*. Beta Draft, Revision 1.2, 1996.

[Ser_96] Sun Microsystems, Inc. *Java Object Serialization Specification*. Beta Draft, Revision 1.2, 1996.

[Tut_96] Sun Microsystems, Inc. *Java RMI Tutorial*. Beta Draft, Revision 1.2, 1996.

[W^{3}K_94] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall. *A Note On Distributed Computing*. SMLI TR-94-29, Sun Microsystems, Inc., 1994.

[Zen_96] Matthias Zenger. *Architektur von EspressoGrinder* in Java Seminarbeiträge, Technical Report 24/96. University of Karlsruhe, Germany, 1996.