Friedrich-Alexander-Universität Erlangen-Nürnberg
Institut für Mathematische Maschinen und Datenverarbeitung IV

Mobilität in objektorientierten verteilten Systemen

Diplomarbeit
im Fach Informatik

vorgelegt von

Artur Schneider

geboren am 19. März 1972 in Hermannstadt

Betreuer: Prof. Dr. F. Hofmann
Dipl. Inf. Martin Geier

Beginn der Arbeit: 27.07.1998
Abgabe der Arbeit: 24.02.1999

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 24. Februar 1999 : ____

Inhaltsverzeichnis

1 Einleitung 1

1.1 Motivation für Codemobilität 1

1.2 Motivation für mobile Objekte 2

1.3 Ziel und Gliederung dieser Arbeit 3

2 Grundlagen 5

2.1 Objektorientierung 5

2.1.1 Zentrale Begriffe der Objektorientierung 5

2.1.2 Weitere Aspekte objektbasierter Programmiersprachen 7

2.2 Verteilte Systeme 11

2.2.1 Definition und Motivation 11

2.2.2 Entwicklung verteilter Systeme 12

2.3 Objektorientierte Verteilte Systeme 12

2.3.1 Objektreferenzen 13

2.3.2 Kommunikations und Interaktionsmechanismen 14

3 Konzepte mobiler Objektsysteme 17

3.1 Aufgaben mobiler Objektsysteme 17

3.2 Systemarchitektur 19

3.2.1 Ansatzpunkte 19

3.2.2 Ausführungsorte 21

3.2.3 Infrastruktur 22

3.3 Handhabung mobiler Objekte 22

3.3.1 Migrationseinheiten 22

3.3.2 Mobile und immobile Objekte 23

3.3.3 Migrationsanweisungen 24

3.4 Durchführung der Migration . . . 26
3.4.1 Ressourcen . . . 26
3.4.2 Transferdarstellung von Migrationseinheiten . . . 29
3.4.3 Mobiler Code . . . 29
3.4.4 Datenzustand . . . 34
3.4.5 Ausführungszustand . . . 39
3.4.6 Objektidentität und Referenzierung . . . 41
3.5 Sicherheitskonzepte . . . 45

4 Auswertung mobiler Objektsysteme **51**
4.1 Emerald . . . 51
4.1.1 Systemarchitektur . . . 52
4.1.2 Handhabung mobiler Objekte . . . 52
4.1.3 Durchführung der Migration . . . 53
4.2 Java . . . 55
4.2.1 Systemarchitektur . . . 55
4.2.2 Handhabung von Java-Objekten . . . 57
4.2.3 Durchführung der Migration . . . 58
4.2.4 Sicherheitskonzepte . . . 61
4.3 Sumatra (Java basiert) . . . 62
4.3.1 Systemarchitektur . . . 62
4.3.2 Handhabung mobiler Objekte . . . 62
4.3.3 Durchführung der Migration . . . 64
4.4 Voyager (Java basiert) . . . 65
4.4.1 Systemarchitektur . . . 65
4.4.2 Handhabung mobiler Objekte . . . 66
4.4.3 Durchführung der Migration . . . 68
4.4.4 Sicherheitskonzepte . . . 69
4.5 IBM Aglets (Java basiert) . . . 70
4.5.1 Systemarchitektur . . . 70
4.5.2 Handhabung mobiler Objekte . . . 70
4.5.3 Durchführung der Migration . . . 72
4.5.4 Sicherheitskonzepte . . . 74

5 Zusammenfassung und Ausblick **75**

5.1 Zusammenfassung 75

5.2 Ausblick 76

Literatur **79**

Kapitel 1

Einleitung

1.1 Motivation für Codemobilität

In der Entwicklung der Computer und ihrer Software läßt sich ein langfristiger Trend ausmachen. Rechenleistung wird immer billiger, die Anzahl verwendeter Computer wächst. Gleichzeitig wächst auch die Übertragungskapazität der Datennetze ebenso wie die Anzahl der daran angeschlossenen Rechner. Insbesondere das Internet ist hier prominent, aber auch firmeninterne oder institutseigene Netze und Übertragungskapazitäten erleben ständige Modernisierung und Erweiterung.

Diese technische Grundlage führte zur Entstehung verteilter Anwendungen. Die klassischen verteilten Anwendungen basieren dabei auf dem Austausch von Nachrichten, wobei häufig einer der kommunizierenden Rechner die Rolle eines Servers beziehungsweise Diensterbringers übernimmt während der sogenannte Client die Dienste in Anspruch nimmt. Dies geschieht auf einem typischerweise recht hohen Abstraktionsniveau (Funktionsaufruf, Datenbankabfrage, Dateiübertragung etc.). Für viele Anwendungen ist das ausreichend. Hohes Abstraktionsniveau bedeutet oft Reduzierung der benötigten Übertragungskapazitäten und gleichzeitig eine einfache Kontrolle über die nach außen von einem Rechner zur Verfügung gestellten Dienste. Zum Beispiel vergleiche man einen Datenbanksuchdienst, der in SQL formulierte Anfragen gestattet, mit einem Suchdienst, der pro Anfrage nicht mehr als ein Suchkriterium annehmen kann. Im letzteren Fall müßten deutlich mehr Anfragen über das Netz geleitet werden. Außerdem müßten die Anfrageresultate vom Client noch zum gewünschten Ergebnis zusammengefaßt werden.

Schwächen dieses Ansatzes liegen in der geringen Flexibilität. Die angebotenen Dienste können im laufenden Betrieb nur soweit angepaßt werden, wie der Entwickler des entsprechenden Dienstprogramms es vorausgesehen hat. Die geringe Flexibilität kann zudem einen Vorteil des hohen Abstraktionsniveaus, nämlich eine Minimierung der genutzten Kommunikationsbandbreite, zunichte machen, falls die Bedürfnisse eines Client nicht angemessen vom Server befriedigt werden können. Als Beispiel sei eine Abfrage an eine große Datenbank gegeben, bei der das Auswahlkriterium für die gewünschten Datensätze nicht mit der vorhandenen Abfragesprache ausgedrückt werden kann. Hier müssen viele überflüssige Daten übertragen werden, die dann schließlich auf der Empfängerseite nach der entsprechenden Filterung verworfen werden.

In Situationen, die hohe Anforderungen an Flexibilität stellen, kann es von Vorteil sein, Programme zu den Daten zu bewegen statt umgekehrt. Diese Art der Kommunikation wird im Gegensatz zur nachrichtenbasierten Kommunikation beispielsweise von Tschudin [Tsc96a] als instruktionsbasiert bezeichnet. Da es nach Tschudin keine klare Grenzziehung zur Unterscheidung von Nachrichten und Instruktionen gibt, stellt der Einsatz von mobilem Code nur eine Grenzverschiebung dar nicht aber einen abrupten Paradigmenwechsel.

Instruktionsbasierte Kommunikation könnte man wohl als Spezialfall der nachrichtenbasierten begreifen, bei der die Menge der in Nachrichten transportierbaren Anforderungen etwa der Menge der Anweisungen einer Programmiersprache entspricht. Die damit gewonnene Flexibilität in der Rechnerkommunikation bezahlt man mit dem Verlust von Kontrolle. Das kontrollierte Angebot genau festgelegter und bekannter Dienste weicht einer Kontrolle, die Schaden vom Nachrichten- beziehungsweise Instruktionsempfänger abwenden soll.

Für die Verwendung mobilen Codes identifiziert Picco in [Pic98] grundsätzlich drei verschiedene Programmierparadigmen. Beim Code auf Abruf (Code On Demand), wie es beispielsweise bei Applets praktiziert wird, können Programme Code über ein Netzwerk laden und dynamisch einbinden. Die entfernte Ausführung (Remote Evaluation) erweitert praktisch den RPC. Dem Aufruf wird der Code, der ausgeführt werden soll, mitgegeben statt wie beim RPC auf Code angewiesen zu sein, der am Ausführungsort des RPC vorhanden ist. Der mobile Agent schließlich transportiert sowohl seinen Code als auch seinen Zustand mit sich, während er von einem Rechner zum anderen migriert.

1.2 Motivation für mobile Objekte

Das objektorientierte Paradigma in Softwareanalyse, -design und Programmierung kommt der Wahrnehmung und der natürlichen Abstraktionsfähigkeit des Menschen entgegen. Somit ist es das bislang vielversprechendste Mittel zur kontrollierbaren Entwicklung komplexer großer Softwaresysteme. Sie verspricht mehr Produktivität indem sie Modularisierung, Softwarewiederverwendung, Systemerweiterbarkeit und Komplexitätskontrolle unterstützt [Kle98]. Die Popularität der objektorientierten Softwareentwicklung spiegelt sich beispielsweise in der Verbreitung von entsprechenden Werkzeugen. Dies sind beispielsweise objektorientierte Programmiersprachen wie Java, C++ oder Smalltalk aber auch Notationen wie UML sowie Analyse- , Design- und Spezifikationsmethoden und CASE-Werkzeuge.

Verteilte Applikationen sind durch Elemente wie Nebenläufigkeit, verteilter Programm- beziehungsweise Prozeßzustand oder Fehleranfälligkeit des zugrundeliegenden Kommunikationssystems im allgemeinen komplexer als Anwendungen für den Einzelrechner. Außerdem haben verteilte Applikationen ein größeres Wachstumspotential, da sie nicht dadurch eingeschränkt sind, daß sie auf einem einzelnen Rechner ausführbar sein müssen.

Damit haben wir in den verteilten Applikationen große komplexe Softwaresysteme gegeben, für deren Entwicklung sich die Verwendung der Objektorientierung anbietet. Die Verwendung des Objekts als Abstraktionseinheit in verteilten Systemen hat bereits viel Anklang gefunden.

Die weitere Kombination der Objektorientierung mit instruktionsbasierter Kommunikation beziehungsweise mobilem Code scheint geeignet, einen Beitrag zur Entwicklung großer und dabei flexibler verteilter Systeme zu leisten. Insbesondere das Programmierparadigma der mobilen Agenten hat in letzter Zeit viel Aufmerksamkeit und Entwicklung erfahren. Gerade für die mobilen Agenten, aber nicht nur für sie, bietet sich Objektmobilität als Basistechnologie an. Viele der in dieser Arbeit untersuchten Systeme sind als mobile Agentensysteme deklariert.

1.3 Ziel und Gliederung dieser Arbeit

In dieser Arbeit geht es darum, programmiersprachliche Objekte als Einheit der Programmmobilität zu betrachten. D.h. es wird untersucht, wie man Objekte zwischen Rechnern migrieren lassen kann, und dabei ihre charakteristischen Eigenschaften Zustand, Identität und Verhalten (siehe Abschnitt 2.1.1) soweit als möglich oder nötig erhalten kann. Die dabei auftretenden Probleme und bekannte Lösungen werden vorgestellt. Die angesprochenen Konzepte sind teilweise nicht nur in Verbindung mit Objektmobilität relevant, sondern häufig bereits in anderem Zusammenhang untersucht worden. Hier geht es jedoch vor allem um ihre Bedeutung im Zusammenhang mit Objektmobilität, ebenso wie um einen möglichst umfassenden Überblick.

Kapitel 2 wird zunächst auf einige Grundlagen von Objektorientierung und verteilten Systemen eingehen. Auf der Basis dieser Grundlagen werden in Kapitel 3 Konzepte erläutert, die bei der Objektmigration verwendet werden. Dabei geht es insbesondere darum, die *verschiedenen* möglichen Lösungsansätze für Problemstellungen der Objektmigration vorzustellen und zu kommentieren. In Kapitel 4 werden schließlich verfügbare mobile Objektsysteme untersucht. Sie sollen auf die Verwendung der in Kapitel 3 herausgearbeiteten Lösungsansätze untersucht werden.

Kapitel 2

Grundlagen

2.1 Objektorientierung

Eine umfassende Definition der Objektorientierung kann im Rahmen dieser Diplomarbeit sicher nicht gegeben werden. Dennoch werden in den folgenden Abschnitten einige wichtige Begriffsdefinitionen der objektorientierten Programmierung gegeben.

2.1.1 Zentrale Begriffe der Objektorientierung

Die Konzepte *Objekt*, *Klasse* und *Vererbung* machen den Kern der Objektorientierung aus. Wegner untersucht in [Weg87] allgemein objektbasierte Sprachen und teilt Sprachen anhand des Vorhandenseins der jeweiligen Konzepte in Kategorien ein.

Jede Programmiersprache, die das Konzept des Objektes unterstützt, wird als objektbasierte Sprache bezeichnet. Klassenbasierte Sprachen sind dabei eine Teilmenge der objektbasierten Sprachen und schließen ihrerseits die Menge der objektorientierten Sprachen ein (siehe Abbildung 2.1).

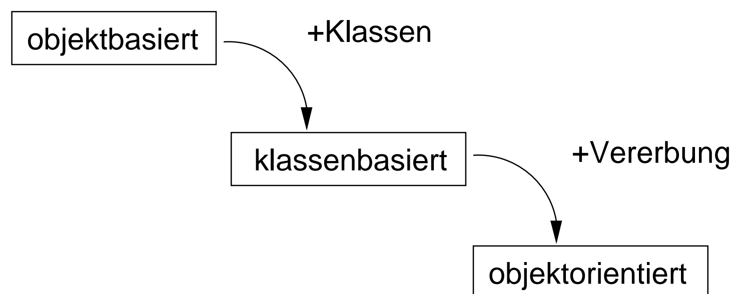

Abbildung 2.1: Unterteilung objektbasierter Programmiersprachen (nach [Weg87])

Objekt

Objekte im Kontext der objektorientierten Softwareentwicklung sind schon häufig und von verschiedenen Autoren definiert worden. In [COR95, Seite 1-1] wird ein Objekt

schlicht als eine identifizierbare, gekapselte Entität betrachtet, die einen oder mehrere Dienste anbietet. Grady Booch geht etwas über den Rahmen der Programmierung hinaus und zählt Beispiele für Objekte auf: Gegenstände, konzeptionelle Einheiten, Entitäten usw. Die Definition des Objektes wird schließlich an Eigenschaften festgemacht. Es ist charakterisiert durch seinen *Zustand*, sein *Verhalten* und seine *Identität* [Boo94, Seite 83]. In [Weg87] findet man eine auf programmiersprachliche Objekte zugeschnittene Definition: ein Objekt hat eine Menge von Operationen und einen Zustand, der sich die Auswirkung von ausgeführten Operationen merkt. Die Identität des Objektes wird hier aber nicht erwähnt. Aus programmiersprachlicher Sicht wird das Verhalten eines Objektes durch eine Menge von sogenannten *Methoden* beziehungsweise Operationen bestimmt und der Zustand durch Attribute beziehungsweise Variablen repräsentiert.

Objekte kooperieren miteinander durch die sogenannte Versendung von *Nachrichten*. Die Nachricht legt fest, welche Methode das Empfängerobjekt ausführen soll und führt die zur Ausführung benötigten Parameter mit sich. Das Versenden von Nachrichten erfordert natürlich, daß verschiedene Objekte sich gegenseitig adressieren beziehungsweise referenzieren können.

Die Versendung von Nachrichten kann verschiedene Semantiken haben. Im synchronen Fall wartet der Sender bis zum Empfang einer Antwort. Es sind aber auch asynchrone Fälle möglich, in denen der Sender einer Nachricht gar keine Antwort erwartet oder sie zu einem späteren Zeitpunkt entgegennehmen will.

Die Weitergabe von Nachrichten kann auf unterschiedliche Weise realisiert werden. Bei Objekten im gleichen Adreßraum wird sie im allgemeinen auf den normalen Prozeduraufruf mit synchroner Semantik abgebildet.

Klasse

Neben dem Objekt als Abstraktionseinheit bieten objektorientierte Sprachen auch das Konzept der Klasse an. In [Boo94, Seite 103] ist die Klasse als eine Menge von Objekten beschrieben, die eine gemeinsame Struktur und ein gemeinsames Verhalten haben. Aus der Sicht des Programmierers ist die Klasse eine Vorlage, nach der Objekte erzeugt (instantiiert) werden können. Klassen dienen vor allem als Container für die Implementierung von Objekten. Bei der Instantiierung eines Objekts, werden die Datenstrukturen angelegt, die seinen Zustand darstellen. Den Code der Methoden teilen sich die Instanzen eines Objektes.

In einigen Programmiersprachen können Klassen – zum Beispiel Metaklassen genannt – ihrerseits als Objekte aufgefaßt werden. Diese Objekte können dann benutzt werden, um Informationen über die Klasse, beispielsweise den Namen der Basisklasse oder Namen von Methoden, zu erfahren oder gar zu verändern.

In *klassenlosen* objektbasierten Sprachen müssen die Eigenschaften eines Objektes bei der Erzeugung entweder komplett angegeben werden oder aber von einem bereits existierenden Objekt – einem Prototypen – übernommen und erweitert oder verändert werden. Man kann sich sogenannter Erzeugerobjekte oder aber auch einfacher Funktionen als Objektfabriken bedienen. Damit kann man zwar recht elegant Klassen simulieren, ei-

ne Simulation der Vererbung erfordert aber enormen Aufwand oder ist gar nicht in vollem Umfang darstellbar.

Neben Klassen, die sich zur Instantiierung von Objekten eignen, gibt es auch solche, die nicht alle oder gar keine der von Ihnen deklarierten Methoden implementieren. Solche Klassen heißen abstrakte Klassen oder im letzteren Fall auch Schnittstellen. Die Methoden müssen dann in abgeleiteten (siehe Abschnitt über Vererbung) oder von anderen Klassen implementiert werden. Schnittstellen und abstrakte Klassen fungieren vor allem als Datentypen (siehe Abschnitt 2.1.2).

Vererbung

Klassen schließlich können Teil einer sogenannten Vererbungshierarchie sein. Dabei wird eine neue (Sub-)Klasse auf der Basis bereits existierender (Basis-)Klassen definiert. Die Subklasse kann die Menge der in den Basisklassen definierten Methoden oder Zustandsvariabeln erweitern. Außerdem kann sie Methoden aus der Basisklasse ausblenden indem sie sie neu implementiert. Der große Nutzen der Vererbung ist die Codewiederverwendung, die dadurch erzielt werden kann. Klassen können in eine Hierarchie eingeordnet werden. Subklassen stellen jeweils eine Spezialisierung der von der Basisklasse dargestellten Abstraktion dar, was sich darin wiederspiegelt, daß die Subklasse die Eigenschaften erbt und die Spezialisierung durch Hinzufügen neuer Eigenschaften erzielt.

Es gibt verschiedene Ausprägungen der Vererbung. Die wohl wichtigste Unterscheidung kann man dabei zwischen Einfach- und Mehrfachvererbung machen. Bei der Einfachvererbung kann eine Klasse nur die Methoden und Variablen einer Basisklasse erben, was Problemen wie zum Beispiel Namenskonflikten vorbeugt. Trotzdem kann es auch Fälle geben, in denen Mehrfachvererbung sehr gut zur Wiederverwendung von Code geeignet ist und Einfachvererbung eine Einschränkung darstellt.

2.1.2 Weitere Aspekte objektbasierter Programmiersprachen

Kapselung

Unter Kapselung versteht man die Kontrolle über den Zugriff auf den Zustand eines Objektes. Dabei bieten verschiedene Sprachen unterschiedliche Möglichkeiten die Sichtbarkeit von Zustandsvariablen und Methoden nach außen hin zu bestimmen. Es gibt ein weites Spektrum von Sprachen, die dem Programmierer hier sehr flexible Mechanismen in die Hand geben bis hin zu Sprachen, bei denen die Sichtbarkeit von Methoden und Variablen bereits fest vorgegeben sind.

Die Programmiersprache Java beispielsweise sieht für die Sichtbarkeit von Methoden und Variablen folgende Schlüsselworte vor: `public`, `default`, `protected`, `private protected` und `private` [Fla96]. Durch diese Schlüsselworte kann der Programmierer die Sichtbarkeit von Variablen und Methoden insbesondere auch in Bezug auf Module und Vererbung festlegen.

In Smalltalk-80 hingegen sind alle Variablen eines Objekts nach außen hin unsichtbar (in Java könnte das durch `private` ausgedrückt werden) während alle Methoden sichtbar (in Java `public`) sind.

Datentypen

Das Konzept der Datentypen ist auch außerhalb der objektorientierten Sprachen relevant. In objektorientierten Sprachen gibt es allerdings die Möglichkeit, Klassen, abstrakte Klassen und Schnittstellen als Datentypen zu verwenden. Eine Klasse definiert einen Datentyp. Alle Instanzen dieser Klasse sind dann vom entsprechenden Typ.

Dabei gibt es auch hier stark unterschiedliche Ausprägungen. In stark getypten Sprachen ergibt sich der Typ für den Wert jeder Variablen aus dem Kontext des Programmtextes. Dies kann man flexibilisieren, indem man unter den Datentypen Kompatibilitätsbeziehungen – zum Beispiel abgeleitet aus den Vererbungsbeziehungen der entsprechenden Klassen oder der Verwendung von Schnittstellen – etabliert. Eine Variable kann dann Werte annehmen, deren Typ kompatibel zu Ihrem deklarierten Typ ist. Datentypen kann man aber auch ganz aus der Sprachsyntax herauslassen wie zum Beispiel in Smalltalk. Der Programmtext liefert dann keine Aussagen über den Typ von Variablen.

Objektidentität

Die *Identität* wurde in der Definition des Objektes in Abschnitt 2.1.1, Seite 5, als wesentliches Merkmal von Objekten angegeben. Die Definition der Identität wiederum ist ebenfalls nicht ganz trivial. In [DK87] wird sie in der allgemeinsten Form als *vollkommene Gleichheit oder Übereinstimmung in Bezug auf Dinge oder Personen* definiert. Damit sind auch schon die beiden bekanntesten Beispiele für Besitzer von Identität genannt. Gegenstände oder auch Personen verschiedener Identität unterscheiden sich stets durch irgendeine Eigenschaft beziehungsweise ein Attribut und sei es nur die Tatsache, daß sie nicht gleichzeitig genau am gleichen Ort sein können.

Ein Problem bei programmiersprachlichen Objekten ist, daß sie allenfalls Abstraktionen realer Objekte sind und somit nur eine endlich große Menge an Attributen haben. Diese können für zwei Objekte in allen Punkten übereinstimmen. Auch für diesen Fall will man jedoch eine Unterscheidungsmöglichkeit. Die Objektidentität wird schließlich selbst zur Eigenschaft erklärt, die ein Objekt von allen anderen Objekten unterscheidet [KC86]. Copeland und Koshafian [KC86] machen für die Unterstützung von Objektidentität zwei wichtige Dimensionen aus, nämlich ihre Repräsentierung und sowie die zeitliche Beständigkeit von Objektreferenzen.

Bei der Repräsentierung geht es vor allem darum, wie Objekte vom Programmierer *identifiziert* werden können und wie Operatoren, zum Beispiel Wertzuweisungen und Gleichheitstests, im Zusammenhang mit der Objektidentität wirken. Hier unterscheiden Copeland und Koshafian drei Abstufungen, für die sie Beispiele anführen, die allerdings nur teilweise aus dem Bereich der objektorientierten Programmierung stammen. Die schwächste Form der Repräsentation ist die durch Datenwerte, sprich Attribute. In klassischen relationalen Datenbanken sind Datensätze ausschließlich durch ihre Attribute

identifizierbar. Somit können Zuweisungen zu Attributen die Identität eines Datensatzes ändern, ohne daß man dabei explizit einen Datensatz löscht und einen neuen erzeugt. Als bessere Form der Repräsentation sehen sie die durch vom Benutzer zugewiesene Namen beziehungsweise durch Variablen. Dabei unterscheiden sie noch einmal, ob zum Beispiel neben Gleichheitstests, die nur die inhaltliche Gleichheit zweier durch verschiedene Variablen repräsentierter Objekte überprüfen, auch solche zur Verfügung stehen, die überprüfen, ob zwei Variablen auf das gleiche Objekt verweisen. Als Beispiel für ein System, bei dem es zwar möglich ist, mehrere Namen für das gleiche Objekt zu vergeben es jedoch keinen Identitätsvergleich für Namen gibt, wird das UNIX Dateisystem angeführt. Hier kann man Verknüpfungen zu bestehenden Dateien anlegen, die dann gleichberechtigt zum bereits bestehenden Dateinamen existieren. Es ist jedoch keine Möglichkeit vorgesehen, um zu überprüfen ob verschiedene Dateinamen auf dieselbe Datei verweisen. Im Sinne von [KC86] unterstützen heute verbreitete objektorientierte Programmiersprachen wie Smalltalk, C++ oder Java eine starke Form der Objektidentität durch Variablen, die Objekte identifizieren, und meist als Referenzen bezeichnet werden. Dabei kann es jedoch unterschiedliche Arten von Referenzen geben (zum Beispiel in C++), die sich im Bezug auf Zuweisungen und Gleichheitstests unterschiedlich verhalten.

Ein Objekt erhält seine Identität bei der Erzeugung und behält diese über seine gesamte Lebensdauer hinweg. Daher ist die Lebensdauer von Objekten und somit ihrer Identität, die sogenannte Persistenz, die zweite wichtige Dimension für Objektidentitäten.

Persistenz

Darunter versteht man im allgemeinen die „Lebensdauer“ von Daten beziehungsweise im vorliegenden Kontext die Lebensdauer von Objekten. Die Spanne der möglichen Lebenszeiten von Objekten umfaßt die Ausführungsdauer von Prozeduren oder von Programmen, darüberhinaus die Einsatzzeiten bestimmter Programme oder Programmversionen und sogar Zeiten, die darüber hinausgehen. Interessant sind dabei besonders die Fälle, in denen Objekte die Programmausführung, während der sie erzeugt wurden, überdauern sollen. Die Sprache beziehungsweise das Laufzeitsystem stellt einen Persistenz-Dienst zur Verfügung, der die Transformation von Objekten in eine abspeicherbare Form sowie die Rekonstruktion von Objekten aus einer abgespeicherten Form übernimmt. Dabei müssen Objektzustand, Objektidentität und Verhalten so eingefroren werden, daß sie später – möglicherweise von einem anderem Programm – wiederhergestellt und verwendet werden können.

Die eben beschriebenen Aufgabenstellungen, die im Rahmen der Objektpersistenz zu lösen sind, sind zum Teil sehr verwandt mit den Aufgabenstellungen, die für mobile Objekte gelöst werden müssen. Tatsächlich ist es eine Form der Mobilität, wenn Objekte aus einer Applikation und somit einem Adreßraum beispielsweise in eine Objektdatenbank abgelegt werden und dann später von einer anderen Applikation aus der Datenbank abgeholt werden. Grady Booch betrachtet die Mobilität quasi als Teil der Persistenz. Seine Definition:

> *Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or*

> *space (i.e. the object's location moves from the address space in which it was created)* [Boo94, Seite 77].

Die Zielsetzung der Objektpersistenz ist jedoch eine andere als die der Mobilität. Hauptziel der Persistenz ist nämlich, die Ablage und Abfrage von Daten besser und einfacher in Applikationen einbinden zu können. Damit gilt es vor allem, große Mengen von persistenten Objekten zu verwalten und einen gezielten Zugriff auf Objekte zu ermöglichen. Diese Aufgabe wird im allgemeinen sogenannten objektorientierten Datenbanken übertragen, die die Vorteile von relationalen Datenbanken, die schnelle Suche nach Datensätzen mit bestimmten Attributen sowie hohe Sicherheit für Datenbestände, mit den Vorteilen der Objektorientierung, der Kapselung von Daten mit Verhalten (Bedeutung) und der Softwarewiederverwendung durch Vererbung, verbinden sollen.

Nebenläufigkeit, Parallelismus

Die Ausführung von Programmen ist nichts anderes als die fortschreitende Ausführung der in ihnen enthaltenen Anweisungen. Die Anweisungen generieren einen sogenannten *Steuerfluß* [RP97]. In konventionellen Systemen existiert pro ausgeführtem Programm ein Steuerfluß beziehungsweise *Aktivitätsträger*, der sich typischerweise in einem Betriebssytemprozeß manifestiert.

Gibt es mehrere gleichzeitig oder quasiqleichzeitig aktive Steuerflüße so spricht man von Nebenläufigkeit. Triviale Fälle von Nebenläufigkeit ergeben sich durch parallele Ausführung von Programmen auf verschiedenen Rechnern oder durch die quasiparallele Ausführung von Programmen auf dem gleichen Rechner. Bei Quasiparallelität existieren gleichzeitig mehrere Steuerflüsse, von denen allerdings zu jedem Zeitpunkt maximal einer aktiv ist. Quasiparallelität wird dadurch erreicht, daß die CPU im Zeitscheibenverfahren abwechselnd den verschiedenen Aktivitätsträgern zugeteilt wird. Neben der parallelen Abarbeitung von Programmen in getrennten Adreßräumen gibt es die Möglichkeit von Nebenläufigkeit innerhalb eines Adreßraums.

Im Zusammenhang mit der Objektorientierung ist besonders das Verhältnis zwischen Objekten und Aktivitätsträgern interessant, dabei insbesondere die Rolle der Objekte in der Kontrolle der Nebenläufigkeit. Papathomas [Pap89] teilt die objektorientierten Sprachen, die Nebenläufigkeit unterstützen, wie in Abbildung 2.2 ein.

In der Klasse der *orthogonalen* Sprachen sind die Konzepte der Nebenläufigkeit und des Objektes unabhängig voneinander. Die Kontrolle der Nebenläufigkeit erfolgt mit den gleichen Mitteln, wie in klassischen Prozeduralen Systemen durch explizite Verwendung von Semaphoren, Monitoren usw. Die Schnittstelle des Objektes sagt nichts über seine Eigenschaften bezüglich Nebenläufigkeit aus. Haben alle Objekte einer *nicht-orthogonalen* Sprache die gleichen Eigenschaften bezüglich Nebenläufigkeit, so heißt die Sprache *uniform*. In *nicht-uniformen* Sprachen gibt es zwei Klassen von Objekten nämlich solche, die die Nebenläufige Ausführung ihrer Methoden kontrollieren können, und solche, die das nicht können. Die *uniformen* Sprachen wiederum lassen sich unterscheiden in *integrierte* Sprachen, in denen Objekte gleichzeitig Aktivitätsträger sind beziehungsweise eigene

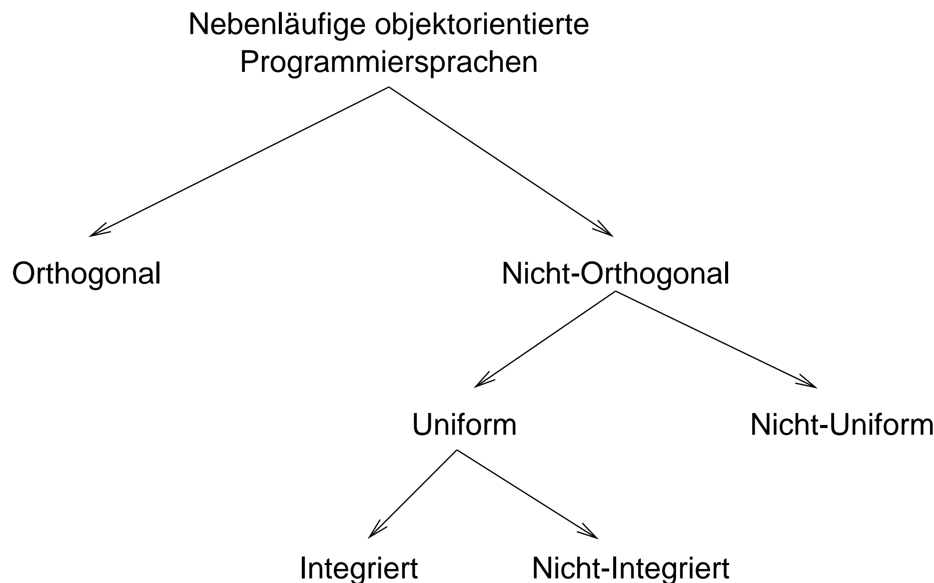

Abbildung 2.2: Nebenläufigkeit in objektorientierten Programmiersprachen (nach [Pap89])

Aktivitätsträger besitzen, und *nicht-integrierte* Sprachen, in denen das Konzept des Aktivitätsträgers – zum Beispiel in Form von Prozessen oder sogenannten Threads – vom Objektkonzept getrennt wird.

2.2 Verteilte Systeme

2.2.1 Definition und Motivation

Neben dem Trend zur Objektorientierung gibt es das Bestreben, verteilte Systeme beherrschbar zu machen und zu nutzen. Die Definition des verteilten Systems ist recht vage. Prinzipiell trifft diese Bezeichnung auf jedes System mit mehr als einer CPU zu, das nach dem MIMD[1] Prinzip arbeitet [Kle98]. Das schließt den eng gekoppelten Multiprozessor ebenso ein wie lose über ein LAN oder WAN miteinander verbundene Rechner.

Verteilte Systeme haben gegenüber zentralen Großrechnern etliche potentielle Vorteile. Diese reichen von einem günstigeren Kosten/Nutzen Verhältnis über höhere Zuverlässigkeit, Verfügbarkeit und Skalierbarkeit bis zu der Tatsache, daß Rechenleistung oft ohnehin auf verschiedene Einheiten – beispielsweise Arbeitsplatzrechner – verteilt ist. Die Vorteile des verteilten Systems sind nur potentiell, da sie sich nicht automatisch aus der Nutzung mehrerer miteinander verbundener Rechner ergeben. Verteilte Systeme bringen dazu etliche Probleme mit sich, die in monolithischen Systemen nicht oder weit weniger gegeben sind. So muß man sich hier beispielsweise mit echter Nebenläufigkeit, Fehleranfälligkeit von Kommunikationssystemen und erhöhter Sicherheitsproblematik beschäftigen. Zusammenhänge in verteilten Anwendungen sind deutlich komplexer als in zentralen Systemen.

[1] Multiple Instruction Multiple Data

2.2.2 Entwicklung verteilter Systeme

Die Dienste zur Nutzung verteilter Systeme waren zunächst recht einfach und an Kommunikationsverbindungen orientiert wie etwa *rlogin*, *rsh* und ähnliche Programme. Später kam der entfernte Prozeduraufruf *RPC* und verteilte Dateisysteme zum Einsatz. Es wurden auch verteilte Betriebssysteme entwickelt, die zum Ziel hatten das verteilte System für den Benutzer wie einen einzelnen Rechner aussehen zu lassen. Diese von *verteilten Betriebssystemen* geforderte Eigenschaft heißt Transparenz und umfaßt nach Tanenbaum [Tan92], daß Ressourcen des Systems stets unverändert genutzt werden können unabhängig davon an welchem Ort beziehungsweise Rechner sie sich befinden, ob sie migrieren, repliziert sind oder nebenläufig auf sie zugegriffen sind. Neben der Transparenz werden auch Flexibilität, Zuverlässigkeit, Leistung und Skalierbarkeit als Entwurfsziele von verteilten Betriebssystemen ausgemacht.

Die vollkommene Implementierung solcher von Tanenbaum *True Distributed Systems* genannten Systeme scheint auf homogene Netztopologien oder Anwendungsgebiete beschränkt zu sein. Die Voraussetzung der Homogenität ist jedoch in vielen Rechnernetzen, insbesondere im prominenten Internet, nicht gegeben. Hier gibt es Rechner mit unterschiedlicher Hardware und unterschiedlichen Betriebssystemen, die nicht zuletzt unterschiedlichen Besitzern gehören. Die Optimierung von Anwendungen kann häufig nicht durch eine automatische vom Betriebssystem getroffene Auswahl der Ressourcen erzielt werden, sondern hängt stark von der Anwendung selbst ab. Man kann sich für verschiedene Anwendungen oder Szenarien unterschiedliche zum Teil gegenläufige Optimierungsziele vorstellen, wie zum Beispiel gute Lastverteilung, Minimierung des zu übertragenden Datenvolumens oder schnelle Antwortzeiten. Hier ist es nötig, dem Anwender oder Anwendungsprogrammierer den Zugriff auf verteilte Ressourcen ebenso wie ihre *Auswahl* zugänglich zu machen. Dies bedeutet einen Schritt weg von der vollkommenen Transparenz, wie sie von verteilten Betriebssystemen wie zum Beispiel Amoeba im Sinne der True Distributed Systems realisiert werden sollte. Stattdessen geht der Trend hin zu sogenannter Middleware, die nicht in das Betriebssystem integriert ist. Middleware bietet dem Anwendungsentwickler verteilte Interaktionsmechanismen an, die sich in der Regel an Programmiermodellen orientieren. Für einige Ausführungen über objektorientierte Middleware siehe Abschnitt 2.3.

2.3 Objektorientierte Verteilte Systeme

Neuere Entwicklungen auf dem Gebiet der verteilten Anwendungsentwicklung sind beispielsweise die objektorientierten Systeme *JavaRMI*, *CORBA* und *DCOM*. Diese versuchen, die Abstraktionseinheit für die Programmierung – das Objekt – auch zur Einheit der Verteilung zu machen. Sie bieten dem Programmierer die Möglichkeit, entfernte beziehungsweise verteilte Objekte (gelegentlich auch Komponenten genannt) in ähnlicher Weise zu verwenden wie lokale Objekte. D.h. Referenzen und Methodenaufrufe haben die gleiche oder ähnliche Syntax und eine ähnliche Semantik für entfernte wie lokale Objekte. Die Verwendung von Kommunikationsmedien ist für den Programmierer bis auf zusätzliche Fehlermöglichkeiten transparent. Die Realisierung der Systeme findet außerhalb des Betriebssystems statt.

Dabei kann man unterscheiden, ob die entsprechenden Systeme in eine entsprechende Programmiersprache integriert sind, oder aber in Form von Bibliotheken auf einer oder mehreren Sprachen aufsetzen. Außerdem kann man unterscheiden, ob alle Objekte bezüglich entfernter Referenzierung und Interaktion gleich gehandhabt werden. Mögliche Unterschiede lassen sich anhand von Objektreferenzierung und Kommunikationsmechanismen aufzeigen, da gerade diese Mechanismen den Schritt vom Objektsystem in einem einzelnen Adreßraum zum verteilten Objektsystem tragen müssen.

2.3.1 Objektreferenzen

In objektorientierten verteilten Systemen spielt die Übergabe von Objektreferenzen über Adreßräume hinweg eine wichtige Rolle. In [COR95, Seite 1-3] wird die Objektreferenz definiert als ein Wert, der ein Objekt eindeutig identifiziert. Im Allgemeinen assoziiert man mit Referenzierung jedoch eher die Adressierung von Objekten. Eine Objektreferenz wird typischerweise auf zwei Arten verwendet, nämlich zur Adressierung von Objekten bei der Versendung von Nachrichten und zur Weitergabe an andere Objekte. In lokalen Systemen werden alle Objekte bezüglich Referenzierung und Methodenaufruf typischerweise gleich behandelt.

Stellvertretend für alle prinzipiell möglichen Unterschiede zwischen lokalen und nichtlokalen Objektreferenzen, die sich in der Verwendung von Objektreferenzen bemerkbar machen, seien die folgenden genannt:

1. Die Menge der referenzierbaren Objekte. Einige Systeme unterscheiden zwischen Objekten, deren Referenz zu anderen Adreßräumen exportiert werden kann, und rein lokalen Objekten. Man spricht hier von einem **nicht uniformen Objektmodell**. Dies ist zum Beispiel bei *JavaRMI* und *CORBA* der Fall. Hier muß der Programmierer bereits zum Zeitpunkt der Programmierung entscheiden, welche Objekte von anderen Adreßräumen aus zugänglich sein sollen. Grund für eine statische Festlegung ist oft der ungleich höhere Aufwand für die Implementierung nichtlokaler Referenzen. Andererseits gibt es auch verteilte Systeme, die hier keinen Unterschied machen (**uniformes Objektmodell**) wie zum Beispiel *Emerald* oder *Obliq*. Die Referenzen aller Objekte sind exportierbar. Dies hat den Vorteil, daß man Entscheidungen über die Verteilung von Objekten über verschiedene Adreßräume und die Referenzierungen unter den Objekten vom Zeitpunkt der Implementierung zum Zeitpunkt der Ausführung aufschieben kann.

2. Die Erzeugung oder Übergabe von Objektreferenzen. Bei *CORBA* beispielsweise müssen Objekte, die nicht-lokal referenziert werden sollen, nach der Erzeugung zunächst explizit exportiert werden. Erst durch diese Anmeldung entsteht eine nicht-lokale Objektreferenz.

3. Konsistenz von Referenzen. Sie erfordert, daß das referenzierte Objekt auch tatsächlich existiert. Dies ist in einem abgeschlossenen Adreßraum noch mit Sicherheit und zu jedem Zeitpunkt feststellbar[2] . In verteilten Systemen jedoch können Referenzen durch Netzpartitionierungen zeitweise und durch Systemabstürze möglicherweise vollständig ungültig werden.

Die Implementierung von Objektreferenzen kann auf unterschiedlichste Weisen gelöst werden. In lokalen Systemen kann man sich eine Objektreferenz beispielsweise als Adresse auf einen Speicherbereich vorstellen, der das entsprechende Objekt enthält. Da es in einigen lokalen Systemen zur Verlagerung von Objekten *innerhalb* eines Adreßraums kommen kann, zum Beispiel wegen Verlagerung von Objekten im Rahmen der sogenannten *garbage collection*, kann es dort sinnvoll sein indirekte Objektreferenzen zu verwenden. In Smalltalk-80 [GR85, Seite 659-661] beispielsweise ist jede Objektreferenz ein Index in eine Objekttabelle, die ihrerseits die Speicheradressen der Objekte enthält. Auf diese Weise muß bei Objektverlagerungen nur der Eintrag in der Objekttabelle geändert werden.

In verteilten Systemen ändert sich das Wesen der Objektreferenz dahingehend, daß sie ein Objekt nicht mehr „nur" innerhalb eines Adreßraums eindeutig identifizieren muß, sondern auch systemweit beziehungsweise netzwerkweit. Dies kann zunächst einfach dadurch geschehen, daß eine lokale Objektreferenz mit der Adresse des jeweiligen Adreßraums verknüpft wird. Dazu muß natürlich der Adreßraum selbst adressierbar sein. Diese Adressierung kann ihrerseits zusammengesetzt sein und wird im Bereich der Kommunikationssysteme und Betriebssysteme geregelt. Sie soll hier schlicht als gegeben angenommen werden.

Dem Programmierer wird ein Stellvertreterobjekt angeboten, das die Interaktion mit dem referenzierten Objekt ermöglicht. Das referenzierte Objekt nimmt seinerseits Aufrufe ebenfalls durch ein Stellvertreterobjekt in seinem Adreßraum entgegen. Die Stellvertreterobjekte können dabei direkt von der Programmiersprache angeboten und intern erzeugt und verwaltet werden. In diesem Falle ist die Verteilung in die Programmiersprache integriert. In nicht-integrierten Systemen sind Stellvertreter normale Objekte im Sinne der zugrundeliegenden Sprache, deren Implementation einzig und allein darauf ausgerichtet ist, andere Objekte zu referenzieren und zu vertreten.

2.3.2 Kommunikations und Interaktionsmechanismen

In objektorientierten Systemen werden Applikationen als Interaktionen von Objekten modelliert. Der wichtigste Interaktionsmechanismus dabei ist der Methodenaufruf beziehungsweise die sogenannte Nachrichtenversendung (message passing). In verteilten Systemen werden Methodenaufrufe an nicht-lokale Objekte von den im vorigen Abschnitt genannten Platzhaltern entgegengenommen. Die Parameter werden in eine Transferdarstellung gebracht, zum referenzierten Objekt geschickt und dort, nach der Rücktransformation aus der Transferdarstellung, der entsprechenden Methode des referenzierten Objektes übergeben. Das gleiche passiert in umgekehrter Richtung mit den Ergebnissen. Diese Art des Methodenaufrufs ist der objektorientierte Nachfolger des entfernten Prozeduraufrufs RPC und wird gemeinhin als RMI (Remote Method Invocation) bezeichnet.

Der entfernte Methodenaufruf kann sich sowohl syntaktisch als auch semantisch vom lokalen Methodenaufruf unterscheiden. Die Unterschiede in der Semantik sind zum Teil

[2] Die Tatsache, daß die Konsistenz von Referenzen in lokalen Systemen grundsätzlich überprüfbar ist, heißt nicht, daß diese Überprüfung auch stattfindet. Dies kann man z.B. an der Programmiersprache C++ sehen.

durch die möglichen Fehler des zugrundeliegenden Kommunikationssystems unvermeidlich. Dies betrifft zum Beispiel die Anzahl der Ausführungen der aufgerufenen Methode. Während die genau-einmal Semantik (engl. exactly-once) beim lokalen System ohne weiteres garantiert werden kann, ist dies in verteilten Systemen kaum möglich, da das entfernte Objekt unerreichbar sein kann. Häufig beschränkt man sich darauf, eine sogenannte at-most-once Semantik zu garantieren, d.h. daß eine Methode höchstens einmal ausgeführt wird. Rechnet man das zeitliche Verhalten einer Methode zur Semantik so fällt auch die verlängerte Antwortzeit hier ins Gewicht.

Semantische Unterschiede können aber auch gewünscht sein. Im lokalen Fall sind Methodenaufrufe meist synchron. D.h. der Aufrufer wird so lange blockiert, bis die aufgerufene Methode abgearbeitet ist. Die Ausführung der aufgerufenen Methode auf einem anderen Rechner bietet sich besonders an, um Nebenläufigkeit einzuführen. Der Aufrufer kann Methodenaufrufe absetzen, deren Ergebnisse er gar nicht oder erst zu einem späteren Zeitpunkt braucht. Die Ausführung der aufgerufenen Methode findet parallel zur weiteren Ausführung des Aufrufers statt. Implizit wird ein neuer Aktivitätsträger erzeugt, mit dem sich der Aufrufer später zur Entgegennahme des Ergebnisses synchronisieren kann. Dies steht im Gegensatz zum lokalen System, in dem neue Aktivitätsträger typischerweise explizit erzeugt werden.

Der entfernte Methodenaufruf kann sich auch syntaktisch vom lokalen Methodenaufruf unterscheiden. Im auf Java aufsetzenden System Aglets von IBM etwa müssen spezielle Nachrichtenobjekte erzeugt werden, die vom Aufrufer an eine `sendMessage()` Funktion des lokalen Platzhalters übergeben werden.

Neben dem Methodenaufruf gibt es noch weitere Interaktionsmechanismen, die nicht nur in objektorientierten Systemen verwendet werden. Als Beispiele seien Ereignisse, Tupelräume, klassischer Nachrichtenaustausch und gemeinsamer Speicher beziehungsweise gemeinsame Variablen genannt.

Kapitel 3

Konzepte mobiler Objektsysteme

Mobile Objektsysteme erweitern das Konzept verteilter objektorientierter Systeme um die Möglichkeit der Objektmigration. Mit dem Begriff mobiles Objektsystem seien im folgenden Systeme gemeint, die sogenannte *mobile Objekte* alleine oder als Teil einer Migrationseinheit (siehe Abschnitt 3.3.1) zwischen verschiedenen Rechnern bewegen können. Dabei bleiben Zustand, Verhalten und Identität aller oder zumindest der meisten der an Migrationen beteiligten Objekte ganz oder teilweise gewahrt.[1]

Wie in Abschnitt 1.2 bereits gesagt ist das Ziel dieser Arbeit die Untersuchung der Objektmobilität.

3.1 Aufgaben mobiler Objektsysteme

Die für uns interessanteste Aufgabe mobiler Objektsysteme ist die Übertragung von mobilen Objekten zwischen verschiedenen Ausführungsumgebungen, insbesondere auf verschiedenen Rechnern, unter bestmöglicher Erhaltung ihres Zustands, ihres Verhaltens und ihrer Identität.

Konventionelle Objektorientierte Systeme

Programmiersprachliche Objekte sind stets in einer Ausführungsumgebung enthalten und in einer Entwicklungsumgebung formuliert. Zu deren Aufgaben bezüglich der darin enthaltenen Objekte gehört in klassischen auf einen Adreßraum beschränkten Systemen:

- Erzeugen und Freigeben von Objekten.
- Systemressourcen für Objekte zugänglich machen und kontrollieren.
- Aktivitätsträger zur Verfügung stellen.

[1] Die Literatur bietet auch gänzlich andere Definitionen des Begriffs *Mobiles Objektsystem* bzw *Mobile Object System* an. In [VST96] wird dieser Begriff als migrierbarer Objektgraph definiert, was im Sprachgebrauch dieser Diplomarbeit als eine mögliche Migrationseinheit (siehe 3.3.1) beschrieben wird.

Beim Erzeugen von Objekten muß Speicherplatz angefordert und der Zustand des Objektes initialisiert werden, sei es automatisch oder durch einen vom Objekt beziehungsweise von seiner Klassendefinition festgelegten Konstruktor. Außerdem muß der Code für die Methoden an das Objekt gebunden werden. Beim Freigeben wiederum muß die Ausführungsumbegung den vom Objekt belegten Speicher und eventuell weitere Ressourcen freigeben. Systemressourcen wie zum Beispiel Benutzerschnittstellen und Betriebssystemdienste werden als global zugängliche Objekte, Prozeduren oder Klassenbibliotheken zur Verfügung gestellt. Ein oder mehrere Aktivitätsträger, die die Methoden der Objekte aktivieren können, müssen erzeugt und verwaltet werden. Diese Aufgaben können dabei je nach Implementierung von der Laufzeitumgebung, zum Beispiel von einem Interpreter, wahrgenommen werden oder aber von der Entwicklungsumgebung, d.h. von einem Übersetzer, der programmiersprachliche Konstrukte in maschinennahe Instruktionen umwandelt. Auch hybride Ansätze sind hier möglich.

Verteilte Objektorientierte Systeme

Hier kommen noch weitere Aufgaben hinzu. Siehe auch Abschnitt 2.3.

- der Export von Referenzen in andere Adreßräume ebenso wie der Import von Objektreferenzen aus anderen Adreßräumen und
- die Implementation von verteilten Kommunikationsmechanismen wie zum Beispiel dem entfernten Methodenaufruf (RMI).

Mobile Objektsysteme

Die Unterstützung mobiler Objekte macht die Ausführungs*umgebung* zum Ausführungs*ort*, den das Objekt wechseln kann. Das erweitert das Aufgabenfeld noch einmal.

- Export und Import von Objektzuständen.
- Codeübertragung und -bindung.
- Unterstützung der Referenzierung und Identität migrierender Objekte.
- Migration von Aktivitätsträgern / Zuordnung von Aktivitätsträgern zu mobilen Objekten.
- Verwaltung, Steuerung und Überwachung von mobilen Objekten.
- Adressierung von Ausführungsorten zum Zweck der Migration.

Diese neuen Aufgaben erfordern auch eine Erweiterung derjenigen Aufgaben, die bereits für lokale und verteilte objektorientierte Systeme genannt wurden. Ausführungsorte, die

die Migration von Objekten unterstützen, müssen ein Protokoll zur Übertragung von Objekten unterstützen. Der Import von Objekten ähnelt in gewisser Weise ihrer Erzeugung. Allerdings wird die Initialisierung des Objektes zu einem Standardzustand durch die Rekonstruktion des Zustandes, die es vor der Migration innehatte, ersetzt. Der Export von Objekten erfordert ihre *Herauslösung* aus der Ausführungsumgebung sowie die Transformation in eine Übertragungsdarstellung. Außerdem gestaltet sich die Bindung des Codes für die Methoden anders, da der Code für eintreffende Objekte häufig nicht lokal vorliegt und über das Netzwerk geladen werden muß. Ist die Herkunft des Codes nicht mit Sicherheit vertrauenswürdig, so muß außerdem seine Ausführung und insbesondere der Zugriff auf Systemressourcen kontrolliert werden. Schließlich erfordert die Konsistenterhaltung der Referenzen auf migrierende Objekte Lösungen, die über die Behandlung von Objekten in verteilten Systemen mit stationären Objekten hinausgehen. Mobile Objekte können ihre Aufgaben natürlich nur erfüllen, wenn sie aktiviert werden. Auch die Verwaltung von Objekten, insbesondere die Kontrolle der Lebensdauer von Objekten muß erweitert werden.

Nicht alle mobilen Objektsysteme lösen diese Aufgaben dabei im gleichen Umfang und mit den gleichen Mechanismen. Die folgenden Abschnitte geben einen Überblick darüber, wie die hier angesprochenen Aufgaben von mobilen Objektsystemen gelöst werden können.

3.2 Systemarchitektur

3.2.1 Ansatzpunkte

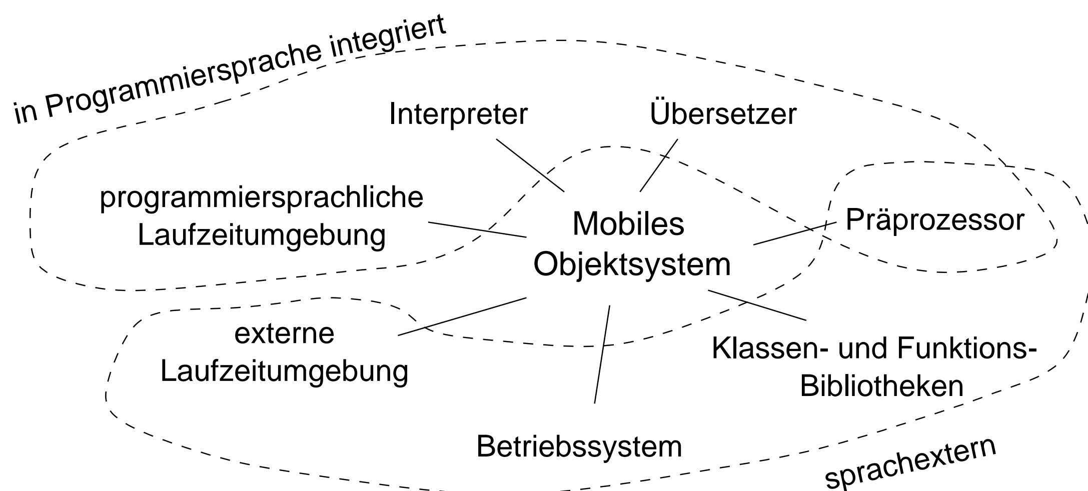

Abbildung 3.1: Ansatzpunkte zur Realisierung mobiler Objektsysteme

Nachdem die Aufgaben eines mobilen Objektsystems in Abschnitt 3.1 umrissen wurden, soll hier gezeigt werden, wo ihre Realisierung angesetzt werden kann. Abbildung 3.1 zeigt eine Übersicht über die möglichen Ansatzpunkte. Da es um programmiersprachliche

Objekte geht, ist es von großer Bedeutung, wie die Unterstützung mobiler Objekte relativ zur Programmiersprache positioniert wird.

Lösungen können direkt in die Sprache integriert sein, in Komponenten, die eine Programmiersprache ausmachen. Dies sind häufig entweder **Übersetzer**, die Quellcode in Maschinencode umsetzen, oder **Interpreter**, die Quellcode als Eingabe erhalten und abarbeiten. Daneben gibt es vor allem in jüngerer Zeit eine Kombination von beidem wie zum Beispiel bei Java, wobei Programme aus dem Quellcode in eine leicht zu interpretierende Form übersetzt und dann interpretiert werden. Außerdem gibt es noch die Möglichkeit, Maschinencode nicht als allein ausführbare Datei zu erzeugen, sondern wie beispielsweise beim Emerald System [JLHB88] in eine **sprachspezifische Laufzeitumgebung** zu laden. Diese Laufzeitumgebung kann beispielsweise beim Laden den Maschinencode überprüfen und bei der Abarbeitung Funktionalität zur Verfügung stellen.

Objektmobilität kann auch außerhalb der Programmiersprache realisiert werden. Besonders häufig wird die Unterstützung der Objektmigration in **Funktions- und Klassenbibliotheken** untergebracht. Objekte können Migrationsunterstützung durch Erben von speziellen Klassen erhalten (beispielsweise bei Aglets [OKO98], Voyager [Obj97b]) oder aber als Parameter an entsprechende Prozeduren übergeben werden.

Alternativ oder ergänzend kann Quellcode vor der Interpretation oder Übersetzung durch einen sogenannten **Präprozessor** modifiziert werden (zum Beispiel in Voyager Version 1.0.0 [Obj97b]). Nimmt ein Präprozessor auf die Syntax einer Sprache Einfluß, so gehört er prinzipiell zur Programmiersprache selbst dazu. Oft wird mit Präprozessoren jedoch eine klassische Programmiersprache erweitert. Bezüglich der damit neu entstehenden erweiterten Sprache ist der Präprozessor ein *integraler* Bestandteil, jedoch *extern* zu der zugrundeliegenden Sprache.

In Kombination mit Bibliotheken oder Präprozessoren kann Funktionalität auch von speziellen außerhalb von Interpretern angesiedelten Programmen, **externen Laufzeitumgebungen** übernommen werden. Als solches kommt prinzipiell auch das **Betriebssystem** in Frage, das uneingeschränkten Zugang zur Hardware wie z.B zur Speicherverwaltung (MMU) hat. Diese läßt sich für das transparente Einlagern von Objekten in den aktuellen Adreßraum (siehe [SGM89], [KTM+96]) ebenso nutzen, wie für die Kontrolle von Speicherzugriffen. Die von Betriebssystemen im Rahmen des Mehrbenutzerbetriebs angebotenen Sicherheitskonzepte könnten auf mobile Objekte zugeschnitten werden. Eine direkte Einbindung des Betriebssystems ist jedoch bei keinem der näher untersuchten mobilen Objektsysteme beschrieben.

Voll integrierte Lösungen haben den Vorteil, daß Objektzustand, -identität und -referenzierung sowie Mechanismen für Methodenaufrufe und Aktivitätsträger direkt zugänglich und manipulierbar sind. Sie ermöglichen uniforme Objektmodelle und hohe Transparenz. Sollen Mobilitätsaspekte syntaktisch repräsentiert werden, so ist grundsätzlich eine neue Programmiersprache vonnöten.

Nicht integrierte Lösungen sind darauf angewiesen, Zugang zu den vorhin erwähnten Informationen über Objekte und Aktivitäten durch Verwendung von Anweisungen der entsprechenden Programmiersprache zu erhalten und sie manipulieren zu können. Programmiersprachen können diesen Zugriff entweder direkt anbieten zum Beispiel indem sie Ob-

jektzustände[2] exportieren, oder aber indirekt über generelle Erweiterungsmechanismen der Sprache, die einen allgemeinen Zugang zu Interna von Interpretern oder Compilern gestatten.

In vielen heute weit verbreiteten Programmiersprachen ist der Zugriff auf Interna nur eingeschränkt möglich, was die Möglichkeiten der Objektmigration limitieren kann. Sind diese Einschränkungen für die angestrebte Semantik der Objektmigration zu stark, so führt kein Weg an einer Modifizierung der entsprechenden Sprachimplementation vorbei. Ein Großteil der Funktionalität mobiler Objektsysteme kann jedoch extern belassen werden und nutzt über entsprechende Schnittstellen die in die Programmiersprache eingebaute Funktionalität.

Für die Einordnung von mobilen Objektsystemen ist es relevant, welche der erwähnten Ansatzpunkte zum Einsatz kommen und welche Aufgaben sie übernehmen.

3.2.2 Ausführungsorte

Mobile Objekte befinden sich in Ausführungsumgebungen beziehungsweise Ausführungsorten. In der Literatur finden sich dafür Bezeichnungen wie Computational Environment [Pic98], Place [Genb], Location [SBH96] oder Context [OKO98]. Dabei handelt es sich um Abstraktionen, die sich auf einen ganzen Rechner, auf einen Betriebssystemprozeß, oder Strukturen innerhalb eines Prozesses beziehen können. Solche internen Strukturen können dabei gleichberechtigt nebeneinander existieren oder aber in eine Hierarchie eingebettet sein. Durch dieses Konzept lassen sich innerhalb einer Ausführungsumgebung unterschiedliche Sicherheitsstufen realisieren oder mehrere Programmiersprachen einsetzen.

Die Ausführungsorte sind in aller Regel nicht mobil. Sie stellen den mobilen Objekten Kommunikations-, Migrations- und andere Dienste zu Verfügung und sind zuständig für die Überwachung der Ausführung von Objekten. Ausführungsorte und darin enthaltenen interne Strukturen sind zum Zweck der Objektmigration *adressierbare Einheiten*.

Ist die adressierbare Einheit gegeben, so stellt sich noch die Frage nach der Repräsentation der Adressierung. Hier bieten sich verschiedene, unterschiedlich transparente Möglichkeiten an. Da alle Kommunikation und auch Objektmigration letztlich auf klassischen Kommunikationsdiensten beruhen, besitzen die diversen Laufzeitumgebungen also Zugang zu einem Kommunikationsendpunkt, der natürlich eine Adresse besitzt[3]. Dies bietet sich natürlich auch an, um Ausführungsorte zu adressieren. Das Adressierungsschema kann erweitert werden, um interne Strukturen von Ausführungsumgebungen zu berücksichtigen.

Als Alternative dazu bietet sich der in objektorientierten, verteilten Systemen verbreitete Namensdienst beziehungsweise Nameserver an. Jede adressierbare Einheit tritt als entfernt referenzierbares Objekt in Erscheinung und macht seine Referenz zusammen mit

[2] Dies kann z.B. zum Zweck der Implementierung von Persistenz bereits vorliegen.

[3] Die wohl bekanntesten im Einsatz befindlichen Kommunikationsprotokolle entstammen der TCP/IP-Protokollfamilie. Darin sind Recher durch vier Byte lange IP Adressen adressierbar und Kommunikationspunkte innerhalb eines solchen Rechners durch Port-Nummern.

einem Namen einem Nameserver bekannt. Diese Referenz kann dann verwendet werden, um das Migrationsziel eines Objektes zu identifizieren. Auch die Referenz eines mobilen Objektes kommt als Adressierungsinstrument in Frage, wenn es zum Beispiel wichtig ist, zwei Objekte für intensive Interaktion zusammenzuführen.

3.2.3 Infrastruktur

Ausführungsorte schließlich müssen in irgendeiner Form zusammenwirken um ein verteiltes, mobiles Objektsystem zu realisieren. Dabei können alle mitwirkenden Komponenten *uniform* sein und müssen in diesem Fall alle anfallenden Aufgaben übernehmen. Im Gegensatz dazu kann ein System auch aus verschiedenen spezialisierten Komponenten bestehen. Beispiele für spezialisierte Komponenten sind zum Beispiel Codeserver (Java) oder Komponenten, die zwar mobile Objekte in anderen Ausführungsumgebungen steuern können, selber aber keine mobilen Objekte aufnehmen (Aglet Clients [OKO98]).

Weiterhin lassen sich Architekturen nach den Anforderungen an Kommunikationsverbindung der Ausführungsumgebungen unterscheiden. Die meisten Systeme, zum Beispiel Aglets [OKO98], Voyager [Obj97b] oder Emerald [JLHB88], erfordern eine direkte Verbindung von Ausführungsorten zum Zeitpunkt der Objektmigration. Häufig ist auch später eine regelmäßige Verbindungsaufnahme nötig, um mobile Objekte vor vorzeitiger Zerstörung zu bewahren. Vorstellbar sind auch Systeme, die ohne direkte Verbindung von Ausführungsorten auskommen, wie zum Beispiel das System PLANET [KTM+96], das einen verteilten Objektspeicher zur Verfügung stellt. Mobile Objekte werden wie eine Email einem Lieferservice übergeben, von dem sie später abgeholt werden können.

3.3 Handhabung mobiler Objekte

3.3.1 Migrationseinheiten

Das Objekt als Einheit der Abstraktion bei der objektorientierten Softwareentwicklung, bietet sich auch als Einheit der Migration an. Oft ist es jedoch sinnvoll, mehrere Objekte zur Migration zusammenzufassen. Denkbare Migrationseinheiten sind einzelne Objekte, Objektgruppen, Objektgraphen, Aktivitätsträger oder ganze Applikationen.

Objekt Die Migration einzelner Objekte erfordert die klare Abgrenzung des Objektes beziehungsweise seiner Zustandsinformation. Das heißt vor allem, für jedes referenzierte Objekt zu entscheiden, ob es Teil des zu migrierenden Objektes ist oder nicht (Aggregation versus Link, [Boo94]). Die Abgrenzung kann sich an Daten- und Objekttypen orientieren oder aber an speziellen Hinweisen seitens des Programmierers. (siehe auch 3.4.4.3).

Objektgruppe Für eine explizite und genaue Kontrolle von Migrationseinheiten bieten sich Objektgruppen (siehe [ARS96]) oder Objektcontainer an. Für die Handhabung

von Objektgruppen sind viele Mechanismen denkbar. Wichtig ist dabei vor allem, ob ein Objekt mehreren Objektgruppen angehören kann und wie die Gruppenzugehörigkeit von Objekten hergestellt beziehungsweise beendet wird. Jedes Objekt einzeln explizit zuzuordnen kann recht aufwendig sein. Alle Möglichkeiten zur Manipulation von Objektgruppen aufzuführen würde sicher zu weit führen. Nur als Beispiele seien erwähnt die Vererbung der Gruppenzugehörigkeit bei der Erzeugung neuer Objekte oder aber hierarchische Gruppen, die andere Gruppen enthalten können.

Objektgraph Ein abgeschlossener Objektgraph (siehe zum Beispiel [Sun98]) wird typischerweise durch ein Objekt sowie die transitive Hülle [SGM89] der von diesem Objekt aus direkt und indirekt referenzierbaren Objekte festgelegt.[4] Objektgraphen sind nicht so gut kontrollierbar wie Gruppen. Da sie aber ein Geflecht potentiell miteinander interagierender Objekte darstellen, können sie als Migrationseinheit geeignet sein. Problematisch ist, daß solche Objektgraphen oft sehr groß werden können (siehe [Gri98a]) oder aber auch Teile der Laufzeitumgebung umfassen. Aus einem solchen Graphen kann die Migrationseinheit durch Einschränkung der Objektmenge gewonnen werden. Es entsteht dabei ein Rand aus Objekten, die die Begrenzung der Migrationseinheit markieren. Zu weiteren Informationen siehe 3.4.4.4.

Aktivitätsträger Zur Festlegung der zu migrierenden Objektmenge können auch Aktivitätsträger (englisch: Threads) verwendet werden. Versteht man einen Aktivitätsträger als Objekt, das die von ihm aktivierten Objekte referenziert, so ergibt sich die Menge der zu migrierenden Objekte wie im Falle der Objektgraphen.

In der Literatur finden sich für migrierende Einheiten oft Bezeichnungen wie zum Beispiel Executing Units [CGP96, Pic98] oder Mobile Agenten. Diese Begriffe sind nicht an objektorientierte Konzepte gebunden. Im Falle einer objektorientierten Implementierung derselben, kann man sie auf eine der hier erwähnten Migrationseinheiten abbilden. Als Migrationseinheit kommen auch ganze Applikationen, d.h. ganz Prozesse beziehungsweise Adreßräume in Frage. Dies fällt jedoch mehr in den Bereich Prozeßmigration als Objektmobilität.

3.3.2 Mobile und immobile Objekte

Neben der Zusammenfassung von Objektmengen zu Migrationseinheiten spielt es auch eine Rolle, ob Objekte bezüglich Objektmigration unterscheidbar sind. Sind alle Objekte gleichermassen migrierbar so spricht man von einem bezüglich Mobilität uniformen Objektmodell. Generell kann man jedoch differenzieren zwischen Objekten, die allein migrieren können, solchen die nur als Teil einer Migrationseinheit migrierbar sind und nicht-mobilen Objekten. Der Begriff des **mobilen Objekts** bezeichne im folgenden programmiersprachliche Objekte, die Teil einer Migrationseinheit sein können.

[4] Anmerkung: Zwei nicht identische, nach der gegebenen Definition abgeschlossene Objektgraphen sind nicht automatisch disjunkt!

Eine Einordnung von Objekten in diese Kategorien kann sich ergeben aus Vererbung durch entsprechende Basisklassen (Odyssey, Voyager), spezielle Schlüsselworte (Java: `Remote`, `Serializable`; SOS [SGM89]: `dynamic`), die Art der Instantiierung (Voyager) oder aber durch dynamische Objektmodifikation (Emerald: `fix`, `unfix`; Voyager: nachträgliche Erstellung von Stellvertreterobjekten). Dabei lassen sich unterschiedliche Relationen zwischen der Gültigkeitsdauer dieser Einordung und der Lebensdauer entsprechender Objekte erkennen. Eigenschaften, die durch Vererbung oder die Angabe von Schlüsselworten bei Klassendeklarationen festgelegt werden, überdauern die Lebenszeit von Objekten. Einflußnamen bei der Instantiierung von Objekten bleiben in der Regel für die Lebensdauer eines Objektes erhalten, während Anweisungen für die dynamische Änderung der Migrationsfähigkeit eines Objektes noch kürzere Zeiträume abdecken.

Interessant ist auch, ob die Zuordnung von Objekten zu Migrationseinheiten dynamisch oder statisch ist und ob für alle Objekte die gleiche Migrationssemantik gilt. Je kürzer die Festlegungen bezüglich Objektmobilität im Vergleich zur Lebensdauer von Objekten gültig sind, desto flexibler läßt sich Objektmigration einsetzen.

Mobile Objekte unterscheiden sich von immobilen Objekten manchmal auch lokal durch die Art des Zugriffs, sei es durch die Repräsentation und die Deklaration von Referenzen (z.B. Voyager), oder die Art der Methodenaufrufe (Aglets).

Ferner gestaltet sich die Kontrolle der Lebensdauer mobiler Objekte oft anders als im lokalen Fall. In konventionellen Systemen sind Verfahren bekannt, die sich am Programmtext (z.B. lokale Variablen in C oder C++) oder an der Referenzierung von Objekten (siehe Java) orientieren. Diese Verfahren werden jedoch verteilten Systemen mit teilweise unterbrochenen Kommunikationsverbindungen nicht immer gerecht. Hier kann eine explizite flexible Kontrolle der Lebensdauer durch den Anwendungsprogrammierer von Nutzen sein (siehe Voyager). Problematisch ist dabei insbesondere, daß mobile Objekte aufgrund von Programmierfehlern, Systemabstürzen oder Fehlern im Kommunikationssystem für eine unkontrollierbare Zeit im System aktiv bleiben und Ressourcen verbrauchen. Die Kontrolle der Lebensdauer nach dem klassischen Prinzip des Referenzzählens kann hingegen zu einer vorzeitigen Zerstörung von Objekten führen.

3.3.3 Migrationsanweisungen

Migration muß gesteuert werden. Steuerungsmechanismen können insbesondere nach Lokalisierung, Aufbau und Semantik der Migrations- und Kontrollanweisungen unterschieden werden.

3.3.3.1 Aktive und passive Migration

Kann eine Migrationseinheit ihre Migration selbst auslösen, spricht man von *aktiver* oder *autonomer* Migration. Dies ist vor allem für die Implementierung von mobilen Agenten von großer Bedeutung. Bei der *passiven* Migration findet die Steuerung außerhalb der Migrationseinheit, zum Beispiel durch andere Objekte oder spezielle Konstrukte wie Reiserouten (engl. *Itinerary*) [WPW+97] oder Aufgabenlisten (engl. *Tasklist*) [Gena], statt.

Die Steuerung kann auch bei der Ausführungsumgebung, liegen, was zum Beispiel beim Einsatz von Objektmobilität zur Optimierung von Lastverteilung naheliegt. Wichtig ist auch, ob Migrationskontrolle nur vom aktuellen Ausführungsort einer Migrationseinheit ausgehen kann, oder ob auch entfernte Migrationssteuerung möglich ist.

Entsprechende Anweisungen befinden sich bei aktiver Migration im Programmcode der entsprechenden Migrationseinheit. Weiterhin kann man unterscheiden, ob diese Anweisungen an beliebiger Stelle im Code untergebracht sein können, oder ob sie nur in speziell ausgezeichneten Objekten der Migrationseinheit, in bestimmten Methoden oder an bestimmten Stellen in Methoden – beispielsweise als letzte Anweisung – enthalten sein können. Derlei Einschränkungen können insbesondere die Problematik bei der Behandlung von Aktivitätsträgern vereinfachen [WBDF98]. So können sie garantieren, daß keine Methoden aktiviert sind, der Aktivitätsträger also leer ist und nicht migriert werden muß. Außerdem können sie dazu dienen, die Konsistenz der von der Migration betroffenen Objektzustände zu gewährleisten und sie können die Menge der möglichen Programmstellen an denen eine weitere Ausführung wiederaufgenommen werden soll verringern und somit einfacher handhabbar machen.

3.3.3.2 Aufbau und Semantik

Migrationsanweisungen müssen prinzipiell drei Dinge festlegen: die *Migrationseinheit*, den *Zielort* und einen *Programmpunkt* für die Ausführung der Migrationseinheit nach der Migration. Jede dieser Angaben kann explizit vom Programmierer gefordert sein oder aber implizit vorliegen. Die Migrationseinheit kann insbesondere bei aktiver Migration implizit vorliegen. Wird eine Migrationseinheit nicht verschickt, sondern von einem Ausführungsort angefordert beziehungsweise abgeholt so ist die explizite Angabe des Zielorts redundant. Die Berücksichtigung des Kontrollflusses in Migrationsanweisungen hängt generell von der Behandlung von Aktivitätsträgern im System ab (siehe 3.4.5).

Je nach Integration der Migration in die Programmiersprache sind Migrationsanweisungen entweder durch Methodenaufrufe oder aber spezielle Operatoren oder Schlüsselwörter realisiert. Vor allem bei passiver Migration ist es möglich, die Implementation der mobilen Objekte von der Migrationssteuerung beispielsweise durch eine entsprechende Kontrollsprache, die Plazierung von Anweisungen in entsprechende Kontrollblöcke oder durch Vorgabe von Reiserouten, die in der Regel als Felder oder Listen von zu trennen.[5] Neben der expliziten Migrationssteuerung durch entsprechende Anweisungen ist auch eine implizite Form denkbar, etwa im Rahmen der Parameterübergabe beim entfernten Methodenaufruf.

[5] Verwandt damit ist die Trennung von Implementation und Koordination in nebenläufigen objektorientierten Programmiersprachen.[Gei96] Eine Trennung der Programm„implementation“ von -interaktion und -migration liegt bei MobileUNITY [PRM97] vor, einer Notation und Logik für die Beschreibung und Untersuchung von verteilten Anwendungen mit mobilem Code. MobileUNITY beschreibt nebenläufige, verteilte Systeme, bestehend aus Komponenten, sogenannten Programmen, die miteinander interagieren. Obwohl die Komponenten nicht Objekte im Sinne der objektorientierten Porgrammierung sind, ist das Konzept der getrennten Definition von Komponentenimplementation, Systemzusammensetzung und Interaction (inklusive Migration) auch im objektorientierten Kontext denkbar.

Gerade bei passiver Migration gibt es auch Konstrukte, die die Migration zwar nicht steuern aber dennoch an Ihr beteiligt sind. So gibt es Anweisungen, die Synchronisationspunkte darstellen, bei denen die Migrationeinheit migriert werden kann (siehe [Fün98]). Derlei Synchronisationspunkte können verwendet werden, um zu gewährleisten, daß Objekte nur in konsistentem Zustand bewegt werden. Auch kann es nötig oder möglich sein, daß bevorstehende Bewegungen der Migrationseinheit beispielsweise durch Callback-Funktionen signalisiert werden, die daraufhin ihren Zustand entsprechend aufbereitet.

Neben der syntaktischen Repräsentation der Migrationskontrolle und Ihrer Positionierung relativ zu den Migrationseinheiten spielt auch die Semantik der zur Verfügung stehenden Anweisungen eine Rolle. Dabei kann man unterscheiden nach strikt imperativen Anweisungen, die unbedingt ausgeführt werden sollen und solchen, die mehr als *Hinweis* an das System verstanden werden. Das System entscheidet, ob es die Migration sofort, zu einem späteren Zeitpunkt oder gar nicht ausführt. (siehe [JLHB88]).

Letztlich ist noch von Belang, ob und wie Migrationseinheiten Erfolg oder Abbruch von Bewegungen überprüfen können. Dies kann explizit durch Callbackfunktionen, Ereignisse oder Fehlerbehandlungsmechanismen wie Ausnahmen und Fehlercodes geschehen. Implizit kann diese Information durch unterschiedliche Fortsetzungen des Kontrollflusses zugänglich sein. Möglich ist aber auch, daß ein Objekt nicht über Migrationsereignisse unterrichtet wird und nur durch Überprüfung seines Ausführungsortes darüber erfahren kann.

3.4 Durchführung der Migration

In Abschnitt 3.3.1 wurden bereits Objekte, Objektgruppen und Objektgraphen sowie Aktivitätsträger als Migrationseinheiten identifiziert. Die genaue Abgrenzung dieser Migrationseinheiten sowie die Behandlung des Zustands bei der Migration läßt sich in einen größeren Rahmen stellen, der nicht nur im Zusammenhang der Objektorientierung gültig ist. Migrationseinheiten enthalten oder referenzieren Ressourcen (siehe 3.4.1), seien dies primitive Daten, Objekte, Objektreferenzen oder umgebungsabhängige Werte wie Dateiidentifizierer oder ähnliches. Obwohl Picco [Pic98, 2.2.2 Data Space Management] das Ressourcenkonzept nur auf die Datenverwaltung bezieht, scheint es geeignet zu sein, auch die Behandlung von Programmcode und Ausführungszustand zu beschreiben. Abschnitt 3.4.1 wird allgemein das Wesen und die Behandlung von Ressourcen im Zusammenhang mit dem Zustand von Migrationseinheiten näher beleuchten.

3.4.1 Ressourcen

Nach Picco [Pic98] sind Ressourcen definiert durch Identifizierer, Wert und Typ. Aus diesen drei Eigenschaften ergeben sich drei Arten wie Ressourcen zu Migrationseinheiten gebunden sein können.

3.4.1.1 Ressourcenbindung

Die Bindung durch Identifizierer (binding by identifier) ist die stärkste Bindung. Die Erhaltung der Identität der referenzierten Ressource muß dabei auch nach der Migration erhalten bleiben. Objektreferenzen, insbesondere Links, gehören in diese Kategorie. Wesentlich dabei ist, daß die Bindung der entsprechenden Ressource an andere nicht migrierende Objekte unbeschadet bleibt. Der gemeinsame Zugriff verschiedener Objekte auf die gleiche Ressource bleibt auch erhalten, wenn diese Objekte durch Migration getrennt werden.

Statt der Identität kann auch nur der Wert der referenzierten Ressource von Belang sein. Objektattribute wie primitive Daten (Zahlen, etc.) sind ein Beispiel hierfür. Auch komplexere Objekte können auf diese Weise referenziert sein. Wesentlich hierbei ist, daß eine Kopie entsprechender Ressourcen keine Änderungen in der Semantik des Zugriffs nach sich zieht. Dies kann zum Beispiel bei Objekten der Fall sein, die nach ihrer Erzeugung nicht mehr modifiziert werden sollen. Auch Programmcode ist für gewöhnlich konstant und kann daher kopiert werden. Bei Objekten, die ausschließlich von *einer* Migrationseinheit referenziert werden, ist es nicht unbedingt nötig, eine *systemweite* Identität aufrechtzuerhalten. Dennoch muß in der Regel die *lokale* Identität erhalten oder wiederhergestellt werden.

Die schwächste Bindung, ist die Bindung nach Typ. Beispiele für derart gebundene Ressourcen sind graphische Benutzerschnittstellen und klassische Ein/Ausgabeströme. In objektorientierten Sprachen tauchen Ressourcen mit derlei Bindungen häufig nicht explizit als Attribute von Objekten auf. Sie werden stattdessen durch sogenannte Klassenvariablen oder sonstige innerhalb eines Ausführungsortes global identifizierbare Variablen repräsentiert, die im Programmcode direkt referenziert werden können. Die Menge der Ressourcen, die zu einer Migrationseinheit nur dem Typ nach gebunden sind, ergibt sich in diesem Falle nicht aus der Analyse der Objektattribute sondern aus der Analyse der Methodenimplementierungen. Wie sich aus den Ausführungen und Beispielen ergibt, bedeutet das Wort Typ in diesem Zusammenhang nicht unbedingt nur Datentyp sondern erfordert auch, daß entsprechende Ressourcen auf eine festgelegte Weise auffindbar sind.

3.4.1.2 Transferierbarkeit von Ressourcen

Neben der Bindung der Ressourcen an die Migrationseinheiten spielt auch die Mobilität der Ressourcen ein Rolle. Picco [Pic98, Abschnitt 2.2.2] gibt auch hier wiederum drei Stufen an. Zunächst unterscheidet er danach, ob Ressourcen überhaupt übertragbar sind. Nicht übertragbare Ressourcen sind zum Beispiel Hardwareteile und Dienste, die untrennbar damit verbunden sind. Die prinzipiell übertragbaren Ressourcen kann man noch einmal unterteilen in solche, die frei übertragbar sind, zum Beispiel mobile Objekte, und solche deren Übertragbarkeit dahingehend eingeschränkt ist, daß die Ressource zwar kopierbar ist, aber nicht aus seiner aktuellen Ausführungsumgebung herausgelöst werden kann.

3.4.1.3 Ressourcenverwaltung

Für jede in irgendeiner Form an eine Migrationseinheit gebundene Ressource muß das System entscheiden, wie es deren Standort und die Bindungen bei Migration verändert. Tabelle 3.1 zeigt die vorhandenen Möglichkeiten im Überblick.

Insgesamt kommen fünf Strategien in Frage:

1. Migration: Die Ressource ist Teil der Migrationseinheit. Insbesondere bedeutet dies auch ihre Entfernung aus dem aktuellen Ausführungsort. In Fällen, wo eine explizite Abgrenzung von Migrationseinheiten nicht gegeben ist, ergibt sich die Zugehörigkeit implizit aus der Ressourcenbindung und ihrer Transferierbarkeit. Insbesondere die Eingrenzung von Objektgraphen kann hieraus abgeleitet werden.

2. Kopie: Auch in diesem Fall gilt die Ressource als Teil der Migrationseinheit, allerdings wird sie nicht aus dem aktuellen Ausführungsort entfernt.

3. Netzreferenz: Die entsprechende Ressource bleibt in ihrer aktuellen Umgebung. Die lokalen Referenzen, über die die Migrationseinheit sie referenziert, werden in Netzreferenzen umgewandelt (siehe auch 3.4.6). Diese Umwandlung kann auch im Zusammenhang mit Migration verwendet werden, um die Bindung zu der wegmigrierten Ressource aufrechtzuerhalten.

4. Neubindung: Hierbei kommt es darauf an, im Zielort der Migrationseinheit eine geeignete Ressource zu finden. Die Kriterien für die Eignung können dabei höchst unterschiedlich sein und ergeben sich häufig aus der (lokalen) Identität. Vorstellbar ist auch der Wert als Auswahlkriterium, oder aber der Grad der Kompatibilität einer Ressource zum geforderten Ressourcentyp.

5. Entfernung der Bindung: Diese Möglichkeit ist sicher am einfachsten zu implementieren, kommt aber nur in Frage, wenn auf die entsprechende Ressource nicht mehr zugegriffen werden muß oder alle anderen Möglichkeiten aus technischen oder andere Gründen ausscheiden.

Übertragbarkeit → Bindung ↓	frei übertragbar	eingeschränkt übertragbar	nicht übertragbar
nach Identifizierer	Migration (Netzreferenz)	Netzreferenz	Netzreferenz
nach Wert	Kopie (Migration, Netzreferenz)	Kopie (Netzreferenz)	(Netzreferenz)
nach Typ	Neubindung (Netzreferenz, Kopie, Migration)	Neubindung (Netzreferenz, Kopie)	Neubindung (Netzreferenz)

Tabelle 3.1: Ressourcenbindung, -transferierbarkeit und -verwaltung (nach [Pic98])

In Tabelle 3.1 ist für jede Kombination von Ressourcenbindung und -transferierbarkeit die Menge der in Frage kommenden Strategien angegeben. Dabei ist jeweils die wahrscheinlich bevorzugte Möglichkeit nicht in Klammern gestellt. Bindungsentfernung wurde nicht

mit angegeben, da sie prinzipiell immer möglich ist aber im allgemeinen nicht bevorzugt wird. Speziell bei Migration und Kopie kann man noch weiter unterscheiden, ob alle Ressourcen gleichzeitig und auf dem gleichen Übertragungswege bewegt werden.

Die im folgenden vorgebrachten Konzepte zur Übertragung von Datenzuständen, Programmcode und Aktivitätsträgern sind stets Ausprägungen der in diesem Abschnitt dargelegten Strategien zur Verwaltung von Ressourcen.

3.4.2 Transferdarstellung von Migrationseinheiten

Der Zustand einer Migrationseinheit muß in eine übertragbare Form gebracht werden. Dabei wird Information über Code, Daten- und Ausführungszustand übertragen, so daß daraus eine identische Kopie erstellt werden kann. Diese Information kann auf höchst unterschiedliche Weisen unterteilt, dargestellt und gewonnen werden. Für die Unterteilung der Zustandsinformation gibt es zwei Dimensionen.

So kann Information über Code, Daten- und Ausführungszustand entweder in einem Block zusammengefaßt oder aber getrennt und möglicherweise auf unterschiedlichen Wegen übertragen werden. Insbesondere im Falle der getrennten Übertragung ist es nötig, Informationen einzufügen, die später eine geeignete Wiederzusammensetzung am Zielort erlauben. Die Übertragung von Information kann teilweise auch unterbleiben, zum Beispiel wenn der Code am Zielort bereits vorliegt oder aber der Verlust des Ausführungszustandes hingenommen wird.

Übertragene Informationen für Code, Daten- und Ausführungszustand können unterschiedlich unterteilt sein. Für den Datenzustand bietet sich der Objektzustand der einzelnen Objekte im Sinne der objektorientierten Programmierung als Einheit an. Beim Code liegt für objektorientierte Sprachen die Klasse beziehungsweise für objektbasierte Sprachen das Objekt selbst als Einheit nahe. Bei Aktivitätsträgern beziehungsweise Threads ist eine Unterteilung auf Objektniveau möglich, wenn auch aufwendig.

3.4.3 Mobiler Code

3.4.3.1 Codeformate

Für die Übertragung von Programmcode bieten sich verschiedene Formate an. Das Spektrum reicht dabei vom Quellcode in textueller Form bis zum Maschinencode, und schließt diverse Darstellungen ein, wie sie bei der Verarbeitung von Quellcode zu Maschinencode als Zwischenschritte anfallen.

Quellcode ist insbesondere bei interpretierten Programmiersprachen das Übertragungsformat der Wahl, bei denen es keine andere Darstellungsform für Programmcode gibt. Aber auch wenn andere Darstellungen möglich sind, so ist die Syntax und Semantik des Quellcodes meist maschinenunabhängig und besser standardisiert und somit portabler als andere Darstellungen, die daraus gewonnen werden können. Auch vom Standpunkt der

Sicherheit bei der Ausführung von nicht vertrauenswürdigem Code kann Quellcode Vorteile haben, da sicherheitsbedingte Einschränkungen auf der Ebene der Programmiersprache formuliert und kontrolliert werden können (siehe 3.5). Nachteile ergeben sich sowohl im Bereich der Ausführungs- als auch der Übertragungsgeschwindigkeit. Da Quellcode für den Programmierer verständlich sein muß, enthält er redundante Information. Schlüsselworte der Sprache, ebenso wie Namen von Variablen, Methoden und Klassen weisen eine weit größere Länge auf, als zur bloßen Unterscheidung nötig wäre. Diese redundante Information kostet Speicherplatz ebenso wie Übertragungszeit.

Programme, die interpretiert werden, laufen erfahrungsgemäß um Größenordnungen langsamer ab als Maschinencode mit der gleichen Funktionalität. Um dem abzuhelfen, kann der Code nach der Übertragung in Maschinencode übersetzt werden. Diese Übersetzung ist jedoch aufwendig und verzögert zunächst die Ausführung. Sie lohnt sich dann, wenn der entsprechende Programmcode zumindest so lange ausgeführt wird, daß Übersetzung und Ausführung zusammen weniger Zeit in Anspruch nehmen als eine interpretative Ausführung. Ein weiterer Nachteil der Übersetzung auf der Empfängerseite ist, daß syntaktische Programmierfehler bis zum Ausführungszeitpunkt unentdeckt bleiben können.

Token Der erste Verarbeitungsschritt von Quellcode ist die lexikalische Analyse. Dabei können die für den Programmierer verständlichen Schlüsselworte und Bezeichner durch stark verkürzte Darstellungen, sogenannte Token ersetzt werden[6](siehe Messenger Environment M0 [Tsc96b]). Die im vorigen Abschnitt erwähnte Redundanz und somit auch der Umfang des Codes wird reduziert.

Abstrakte Syntaxbäume Einen Schritt weiter geht die syntaktische Analyse des Quellcodes, deren Ergebnis ein abstrakter Syntaxbaum ist. In [Fra96] stellt Michael Franz ein auf diese Darstellungsform spezialisiertes adaptives Komprimierungsverfahren vor, mit dem sogenannte *slim binaries* erzeugt werden. Deren Informationsdichte liegt einem Vergleich des Autors zufolge deutlich über dem, was mit konventioneller Komprimierung anderer Coderepräsentationen wie Quellcode, Maschinencode oder Java-Bytecode (siehe unten) erreicht werden kann. Die für Quelltext angegebenen Nachteile des großen Umfangs und der nicht stattfindenden syntaktischen Analyse auf Senderseite sind hier nicht gegeben. Ebenso ist diese Darstellung sicher geeignet, die weitere Übersetzungszeit zu verkürzen und möglicherweise auch die Geschwindigkeit der Interpretation zu steigern. Die Vorteile der Maschinenunabhängigkeit und bezüglich der Ausführungssicherheit sind im gleichen Umfang gegeben wie bei Quellcode.

Maschinencode Das Endergebnis von Codeübersetzung ist in der Regel Maschinencode. Sein herausragender Vorteil ist die hohe Ausführungsgeschwindigkeit. Der Umfang des erzeugten Codes kann dabei stark von der entsprechenden Zielplattform abhängen, besonders von dem zur Verfügung stehenden Befehls- und Registersatz. Typischerweise

[6] Bei der lexikalischen Analyse muß nicht zwangsläufig Informationsverdichtung wie hier angegeben stattfinden. Besonders wenn Informationen für Debugging auf Quelltextebene erzeugt wird, können Bezeichner nicht verkürzt werden.

ist das wichtigste Optimierungsziel beim Entwurf von CPUs nicht kompakter Code sondern hohe Ausführungsgeschwindigkeit ([Fra96]). Auch stellt die direkte Abarbeitung von Maschinencode ein großes Sicherheitsproblem dar (siehe 3.5). Der größte Nachteil des Maschinencodes mit Blick auf Codemobilität ist die Plattformabhängigkeit. Ist die Beschränkung auf eine Hardwareplattform nicht hinnehmbar so bieten sich verschiedene Strategien an.

Eine Möglichkeit, die Beschränkung auf eine Hardwareplattform bei Verwendung von Maschinencode zu umgehen, ist, die Darstellung für alle möglichen Plattformen vorrätig zu halten und jeweils alle oder aber die für den Zielrechner passende zu übertragen. Für offene Netze mit einer nicht vorhersehbaren Vielfalt an Hardware ist dieser Ansatz untauglich. Realistischer scheint da die Kombination von Maschinencode mit einer unabhängigen Darstellung (siehe [Kna96]). Diese Variante bietet sich an, wenn die Zielplattform für den Code mit großer Wahrscheinlichkeit bekannt ist jedoch nicht garantiert werden kann. Im günstigen Falle erlangt man schnelle Ausführungszeiten ohne zusätzlichen Übersetzungsaufwand. Dagegen steht jedoch in jedem Fall der erhöhte Übertragungsaufwand.

Bytecode Eine weitere Möglichkeit ist die Emulation einer entsprechenden Plattform durch einen Interpreter beim Codeempfänger. Java hat eine Variante dieses Konzepts den sogenannten Bytecode populär gemacht. Dabei handelt es sich um eine dem Maschinencode sehr verwandte Darstellung, die jedoch nicht auf einen realen Rechner ausgelegt wurde, sondern für die Abarbeitung durch eine standardisierte *virtuelle Maschine* konzipiert ist. Diese virtuelle Maschine ist als Interpreter realisiert und auf allen gängigen Hardwareplattformen implementiert, wodurch Bytecode maschinenunabhängig wird. Alternativ zur Interpretation bietet sich auch bei Bytecode die Übersetzung in Maschinencode für die jeweils konkret vorliegende Hardware an. Franz [Fra96] weist hier jedoch darauf h in, daß Codeoptimierungen wie sie in vielen Übersetzern eingesetzt werden, für optimale Ergebnisse auch auf strukturelle Informationen über das Programm angewiesen sind. Diese Information ist im Bytecode allerdings größtenteils verloren.

Aus der Diskussion der verwendeten Formate für mobilen Code ergeben sich vier wesentliche Aspekte nach denen sie bewertet werden können. Diese sind **Portabilität**, **Sicherheit**, **Ausführungsgeschwindigkeit** und **Codeumfang**. Abbildung 3.2 zeigt tendenziell die Eignung der diversen Codeformate in Bezug auf diese Aspekte. Dabei handelt es sich wohlgemerkt *nicht* um eine Einschätzung von Systemen, die mit dem jeweiligen Codeformat arbeiten, da sie prinzipiell gegebene Vorteile eines Formats ungenutzt lassen können, während Nachteile durch aufwendige und sorgfältige Implementierung abgeschwächt werden können.

3.4.3.2 Codefragmente

Neben der Wahl eines Übertragungsformates für Code spielt die Unterteilung in separat übertragbare und funktionsfähige *Codefragmente* eine Rolle. In objektorientierten und klassenbasierten Systemen bietet sich, wie etwa bei Java, die Klasse als Einheit an. In objektbasierten Sprachen hat jedes Objekt seinen eigenen Code oder aber er ist in Form spezieller Objekte (zum Beispiel in Emerald [JLHB88]) realisiert. Daneben gibt es noch die

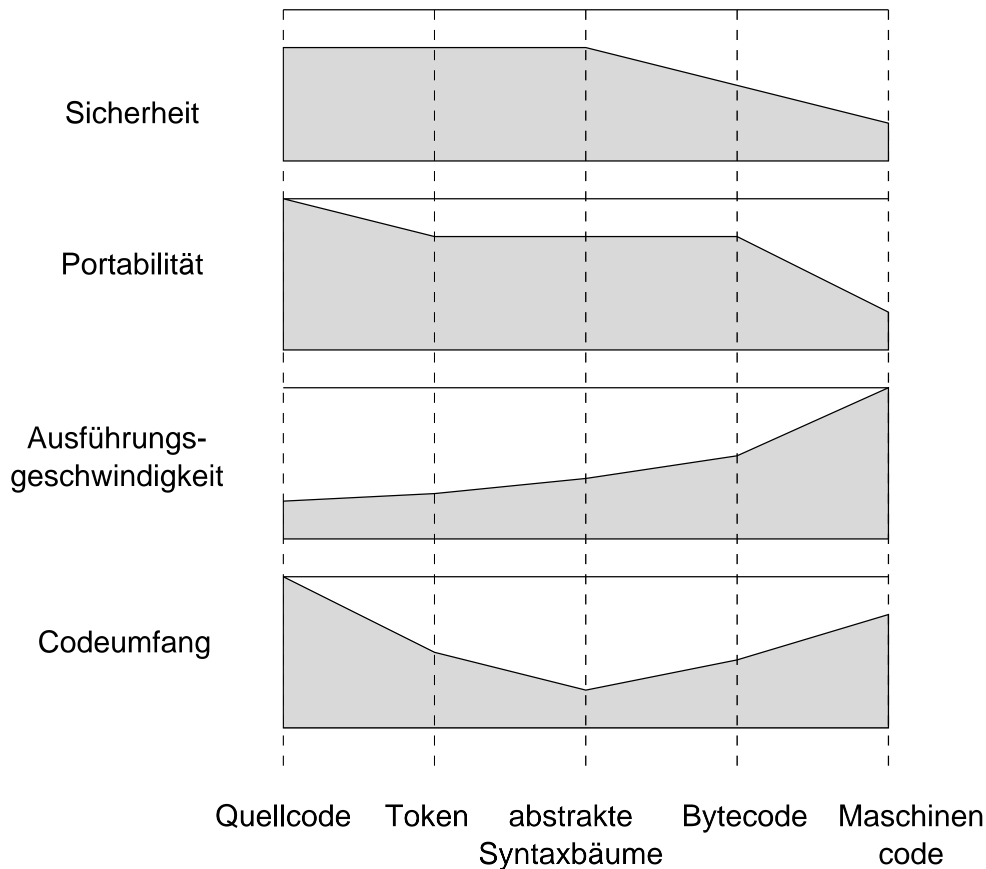

Abbildung 3.2: Bewertung von Formaten für mobilen Code

in objektbasierten Systemen selten eingesetzten Möglichkeiten, Programmcode in Prozeduren (zum Beispiel in Obliq [Car95]) beziehungsweise Methoden oder Module zu unterteilen oder aber ganze Programme oder Programmdateien (AgentTCL) zu übertragen.

3.4.3.3 Codereferenzierung

Codefragmente müssen sowohl untereinander, zum Beispiel für Klassenvererbung oder gegenseitige Aufrufe, als auch aus der Repräsentation des Datenzustands heraus *identifizierbar* sein. Ist Code als eigenes Objekt repräsentiert oder integrierter Bestandteil eines Objektes, so bieten sich hier die gleichen Referenzierungsmechanismen wie für andere Objekte an (zum Beispiel in Emerald [JLHB88]).

Handelt es sich bei den Codefragmenten um Klassen, Prozeduren oder Module, bieten sich deren Namen zur Referenzierung an. Problematisch ist dabei die sogenannte *Namensauflösung*, bei der angeforderte Codefragmente ausfindig gemacht werden und zu der entsprechenden Migrationseinheit gebunden werden. Hierbei wiederum ist insbesondere der Gültigkeitsbereich von Namen relevant. Liegt kein hierarchischer Namensraum mit global eindeutigen Namen vor, wie das etwa bei Java in Anlehnung an das *Domain Name System* des Internet möglich ist, so können verschiedene Migrationseinheiten Codefragmente gleichen Namens enthalten. In diesem Fall ist die Verwaltung der Na-

mensräume verschiedener Migrationseinheiten wichtig. Insbesondere in Systemen, die ohne Codeübertragung arbeiten wie zum Beispiel Distributed Smalltalk [Ben87], werden Namen von Klassen im Kontext des jeweiligen Ausführungsortes ausgewertet. Dabei findet eine Kompatibilitätsprüfung statt, die sicherstellt, daß die Schnittstelle und die Attribute der am Zielort vorhandenen Klasse kompatibel sind zu der Klasse, mit der ein entsprechendes migrierendes Objekt instantiiert wurde. Andere Systeme trennen die Namensräume für Codefragmente verschiedener Migrationseinheiten völlig, kombinieren private Namensräume für Migrationseinheiten mit dem Namensraum des entsprechenden Ausführungsortes (Java) oder erlauben eine beliebige Kontrolle der Namensauflösung durch den Anwendungsprogrammierer.

Ein Problem bei der Identifizierung von Codefragmenten ist das Vorhandensein verschiedener *Versionen*. Ein Ansatz zur Lösung dieses Problems ist die oben angesprochene Kompatibilitätsprüfung. Alternativ dazu kann die Versionierung in die Identifizierung der Codefragmente eingebaut werden. Grundsätzlich gilt es zu entscheiden, ob ein mobiles Objekt während seiner Lebensdauer stets exakt mit dem gleichen Code assoziiert bleiben muß, oder ob ein Übergang zu einer anderen Implementierung möglich oder gar gewünscht ist. Die dynamische Veränderung oder die Neubindung von Code ermöglicht beispielsweise die Interaktion mit Objekten, die a priori nicht bekannt sein können [HBS97]. Dabei muß jedoch darauf geachtet werden, daß diese Änderungen des Codes weder die Konsistenz des Datenzustands noch die des Ausführungszustands der betroffenen Migrationseinheiten gefährden.

3.4.3.4 Codeverwaltung

Auf Codefragmente können besonders die in Abschnitt 3.4.1 vorgestellten Ressourcenverwaltungskonzepte Kopie und Neubindung oder Kombinationen von beidem angewandt werden. Eine Migration mit gleichzeitiger Löschung des Codes am Herkunftsort wird in keinem der untersuchten Systeme angewandt. Stattdessen findet im allgemeinen eine Replikation von Codefragmenten statt. Wie bereits angesprochen, findet die Übertragung von Code nicht unbedingt auf dem selben Wege statt, wie der des Datenzustands. Java (ohne RMI) beispielsweise trennt die Verwaltung und Übertragung von Daten und Code nicht nur in den technischen Implementierungsdetails sondern auch auf der für Anwendungsprogrammierer sichtbaren konzeptionellen Ebene, während auf Java aufsetzende mobile Objektsysteme diese Trennung zumindest auf konzeptioneller Ebene aufheben indem sie dafür sorgen, daß für jedes übertragene Objekt auch der entsprechende Code zur Instantiierung bereitgestellt wird.

Neben der Unterteilung des Codes in Fragmente ist auch die Auswahl der am Zielort benötigten Codefragmente wichtig. Gibt es eine 1:1 Beziehung zwischen einer Migrationseinheit und ihrem Codefragment so, ist die Auswahl klar. In objektorientierten Systemen mit Klassen und Vererbung und Migrationseinheiten, die aus mehreren Objekten bestehen können, ist es jedoch wahrscheinlich, daß mehrere Codefragmente zur Ausführung benötigt werden und ausfindig gemacht werden müssen. Die Menge der zu übertragenden Codefragmente ist jedoch in der Regel kleiner als die Menge aller an die Migrationseinheit gebundenen Fragmente. Übertragener Code wird häufig mehrfach verwendet anstatt nur für die mobilen Objekte wegen derer er übertragen wurde. Codefragmente, die in

allen Ausführungsorten identisch vorliegen, können ohne jegliche Übertragung verwendet werden. In [AS96] wird der Kontrollfluß von prozeduralen Programmen analysiert um festzustellen, welche Programmteile am Zielort überhaupt erreicht werden können, bevor der nächste Migrationsbefehl ansteht. Ein wichtiger Aspekt jener Arbeit ist, daß Bindungsfehler durch nicht verfügbare Codefragmente nur bei der Ausführung von Migrationsanweisungen auftreten können.

Die Ermittlung der benötigten Codefragmente wird gerade in Java-basierten Systemen häufig dem Zielort überlassen, der zunächst nicht den Code selbst erhält sondern nur Informationen über die benötigten Codefragmente (siehe Abschnitt 3.4.4). Der Zielort fordert davon nur diejenigen Fragmente an, die noch nicht bei ihm vorliegen. Diese Anforderung kann vom Zeitpunkt der Migration einer Migrationseinheit zum Zeitpunkt der Verwendung von Codefragmenten verlagert werden (*lazy binding*). Damit läßt sich die Latenzzeit bis zur Ausführung einer Migrationseinheit am Zielort verkürzen und die Menge des übertragenen Codes reduzieren. Die verzögerte Codeübertragung und -bindung ist jedoch anfällig gegen Ausfälle des Netzes oder der Codequelle.

Das Codeformat, das zur Übertragung verwendet wird, ist nicht unbedingt identisch mit dem Format in dem der Code abgearbeitet wird. Häufig wird zur Leistungssteigerung eine Übersetzung in Maschinencode vorgenommen. Soll ein Objekt mehrfach migriert werden, so muß sein Code stets auch im übertragbaren Format verfügbar sein. Dies kann durch Vorhaltung des übertragbaren Codeformates am aktuellen Ausführungsort, durch Zurückübersetzung des Codes (eine eher theoretische Möglichkeit) oder mit Hilfe von speziellen oder in Ausführungsorte integrierten Codeservern realisiert werden.

3.4.4 Datenzustand

Der Datenzustand einer Migrationseinheit setzt sich bei den von uns betrachteten objektorientierten Systemen aus den dazugehörigen Objekten zusammen. Dazu wollen wir uns ansehen wie die zur Übertragung taugliche Darstellung einzelner Objektzustände ebenso wie der Zustände von Objektgruppen beziehungsweise -graphen bewerkstelligt werden kann. Die Umwandlung des Datenzustands in Transferdarstellung wird typischerweise Export oder Serialisierung genannt. Der umgekehrte Vorgang heißt Import oder Deserialisierung.

3.4.4.1 Übertragungsformate für Datenzustände

Für die Übertragung von Datenzuständen bieten sich drei grundsätzlich verschiedene Formate an, die für unterschiedliche Szenarien geeignet sind. Die Übertragung von *Speicherbereichen* eignet sich besonders in Zusammenhang mit maschinennahem Code und kombiniert unter Umständen dabei die Übertragung von Datenzustand mit Codeübertragung, evtl sogar mit der Übertragung von Aktivitätsträgern. Der Einsatz der *flachen Transferdarstellung* erlaubt eine bessere Trennung von Datenzustand und Code sowie einen flexibleren Umgang mit unterschiedlichen Implementationsversionen. Die Übertragung von Daten *eingebettet in Code* erfordert, daß die verwendete Sprache die Möglichkeit bietet, Zustände von Objekten in uneingeschränkter Weise zu manipulieren.

Speicherabbild Der Datenzustand von Objekten und somit auch von ganzen Migrationseinheiten ist in irgendeiner Form im Speicher eines Ausführungsortes repräsentiert. Die betroffenen Speicherbereiche lassen sich kopieren. Diese Art der Zustandsübertragung erfordert, daß Objekte in allen beteiligten Ausführungsorten in gleicher Weise repräsentiert werden. Das erfordert eine identische Darstellung primitiver Daten ebenso wie eine unveränderte Abbildung von Objektstrukturen wie zum Beispiel der Attributreihenfolge auf die Speicherrepäsentation. Somit funktioniert dies nur in homogenen Systemen und schränkt die Verwendung unterschiedlicher Implementationsversionen stark ein.

Wird nicht ein ganzer Adreßraum übertragen, so müssen Objektreferenzen ebenso wie die Referenzierung von Codefragmenten für diese Form der Übertragung entweder bereits so vorliegen, daß die Übertragung ihre Gültigkeit nicht beeinträchtigt, oder aber bei der Übertragung so verändert werden, daß sie im neuen Ausführungsort gelten. Eine Abspeicherung von Objektreferenzen in einer umgebungsunabhängigen Form zum Beispiel als global gültige Objektidentifizierer bedeutet jedoch für die normale Abarbeitung von Programmen eine stark verringerte Geschwindigkeit. Das andere Extrem der Referenzdarstellung, nämlich der Adreßzeiger, ist ohne weitere Informationen kaum zu behandeln. Daher bietet sich eine kombinierte Darstellung an, die sowohl global gültige Anteile enthält als auch lokal schnell auswertbare (Emerald [JLHB88], [SGM89]). Die lokal schnell auswertbaren Anteile werden bei der Migration ungültig und müssen entweder bei der ersten Benutzung der Referenz oder aber im Anschluß an die Übertragung neu ermittelt werden.

Flache Transferdarstellung Aus den klassischen Kommunikationssystemen und RPC-Systemen sind maschinenunabhängige Darstellungsformate (zum Beispiel ASN.1) für primitive und strukturierte Daten bekannt. Dabei existieren für alle primitiven Datentypen und darauf aufbauende Strukturierungsmechanismen (zum Beispiel Felder oder Structs) *Abbildungsvorschriften*, die ein im Speicher vorliegendes Datum oder eine Datenstruktur in einen Zeichen- oder Bytestrom umwandeln sowie inverse Abbildungen, die aus dieser Darstellung entsprechende Kopien erstellen. Diese Mechanismen müssen um eine Behandlung von Objektreferenzen – seien dies Netzreferenzen oder zur Migrationseinheit lokale Referenzen – erweitert werden ebenso wie das Konzept der Datenstrukturen und Datentypen an Objekte und damit an die Bindung an Codefragmente angepaßt werden muß. Diese Art der Zustandsübertragung findet beispielsweise in Java [Sun98] statt, und wird dort Serialisierung genannt.

In klassischen verteilten Systemen sind symbolische Informationen, wie zum Beispiel Namen von Variablen beziehungsweise Attributen oder im objektorientierten Kontext Methodennamen und -signaturen prinzipiell redundant und werden in der Transferdarstellung meist weggelassen. Dies ergibt sich daraus, daß die zu übertragenden Datentypen und -strukturen häufig auf Sender- wie auch auf Empfängerseite im voraus bekannt sein müssen, was dem Vorhandensein entsprechenden Codes für die Abbildungsvorschriften auf beiden Seiten entspricht. Mobile Objektsysteme treten jedoch gerade an, diese Einschränkung mit mobilem Code zu überwinden. Auch wenn mit mobilem Code Information über Datenformate übertragbar ist, kann die Einbindung symbolischer Information in die Übertragungsdarstellung von Datenzuständen die Bindung an geringfügig verschiede-

ne aber kompatible Codefragmente erlauben. Auch erweitern sich dadurch die Möglichkeiten für die Entdeckung von Fehlern und Inkompatibilitäten.

Code Eine weitere Möglichkeit zur Übertragung von Zustandsinformation mobiler Objekte ist die Generierung von Programmcode, der am Zielort entsprechende Objekte erzeugt und die Attribute entsprechend setzt. Primitive Daten spiegeln sich dabei in Form von Konstanten im erzeugten Code wieder. Objektstrukturen sowie Objektreferenzen werden vom Code durch entsprechende Anweisungen generiert. Diese Art der Zustandsübertragung setzt beim Empfänger keine besonderen Mechanismen voraus ist aber auf der Senderseite aufwendiger als die anderen beiden Formate. Diese Form der Zustandstransformation wird zum Beispiel in Smalltalk-80 zur Unterstützung von Persistenz angewandt, kann dort jedoch nur zyklenfreie kleine Strukturen verarbeiten [Veg86].

3.4.4.2 Kontrolle der Datenzustandsübertragung

Die Umwandlung von Migrationseinheiten in Transferdarstellung kann durch verschiedene Maßnahmen kontrolliert werden.

Da ist zunächst einmal die Abgrenzung von Objektzuständen, Objektgraphen und allgemein von Migrationseinheiten (Abschnitt 3.3.1). Dabei geht es um die Abgrenzung zum Zweck der Übertragung im Gegensatz zur Abgrenzung von Objekten und Objektgraphen während der Ausführung. In der Regel basiert die Zustandsabgrenzung für die Übertragung auf derjenigen für die Ausführung, nimmt davon aber einige Teile aus. Die genaue Abgrenzung der Migrationseinheiten, inklusive übertragener Codefragmente und teilweise auch der Aktivitätsträger, findet letzten Endes durch die Kontrolle über den Datenzustand statt.

Neben der Möglichkeit zur Abgrenzung besteht in einigen Systemen auch die Möglichkeit, das Exportieren und Importieren von Objekten oder Graphen ganz oder teilweise in die Hände des Anwendungsprogrammierers zu legen (zum Beispiel Java [Sun98]). Dabei ist es wichtig, daß der Zustand zugänglich ist. Dies kann entweder dadurch gegeben sein, daß direkter Zugriff auf den Arbeitsspeicher möglich ist (zum Beispiel in C++), oder aber daß zumindest entsprechende Umwandlungsroutinen vorhanden sind, die primitive Daten in Datenströme ausgeben und einlesen können (Java). Bei der Implementierung der Datenzustandsübertragung durch den Programmierer ist festzulegen welche Codefragmente (zum Beispiel Klassen) für welchen Teil des Objektzustands zuständig sind. Die Realisierung in Java beweist dabei, daß Serialisierung mit Vererbung durchaus verträglich ist (im Gegensatz zur Nebenläufigkeit, die ja Vererbungsanomalien verursachen kann). Problematischer ist hier das Vorhandensein verschiedener inkompatibler Versionen von Codefragmenten.

Viele Systeme schirmen den Anwender jedoch von der Zustandsübertragung völlig ab (Emerald, Obliq) und führen den Import und Export von Datenzustand intern durch.

3.4.4.3 Objektzustand

Für die Erzeugung einer identischen Kopie eines Objektes sind zumindest Informationen über die Implementierung der Objektmethoden sowie über den Datenzustand des Objektes zwingend erforderlich. Zu Informationen über den Ausführungszustand siehe Abschnitt 3.4.5.

Implementierungsinformation Für die Wiederherstellung eines Objektes ist es von Bedeutung, daß mit seinem Datenzustand eine Implementierung in eindeutiger Weise assoziiert wird. Die Information über die Objektimplementierung kann dabei prinzipiell direkt als Code in irgendeiner Form mit dem Objektzustand mit übertragen werden. In vorhandenen mobilen Objektsystemen liegt jedoch fast immer eine zumindest logische Trennung des Codes vom Datenzustand des Objektes vor, da es meist mehrere Objekte mit der gleichen Implementation gibt. Der Code liegt also in der Regel als referenzierbare Einheit oder Ressource vor. Entsprechende Codereferenzen müssen in die Transferdarstellung des Datenzustandes eingebunden werden. Für weitere Ausführungen zu Codeübertragung siehe Abschnitt 3.4.3.

Abgrenzung des Objektzustandes Der Datenzustand eines Objektes manifestiert sich in seinen sogenannten Attributen. Diese Attribute wiederum sind Variablen, die Werte enthalten. Diese Werte können unterschiedlicher Natur sein. Zum einen kann es sich um sogenannte, von vielen Programmiersprachen unterstützte, *primitive Werte* handeln. Dies sind beispielsweise Zahlen, Zeichenketten oder Boolesche Werte, die für sich selbst stehen und in einer von der zugrundeliegenden Plattform oder einem entsprechenden Interpreter direkt verwertbaren Format vorliegen.

Daneben gibt es noch *Referenzen* auf andere Objekte. Hier muß zunächst unterschieden werden, ob das referenzierte Objekt eigenständig ist, oder aber nur im Kontext des referenzierenden Objektes existiert und verwendet wird. Referenzen auf eigenständige Objekte werden als Links beziehungsweise Verknüpfungen bezeichnet. Andernfalls spricht man von Aggregation beziehungsweise davon, daß ein Objekt als Attribut in einem anderen enthalten ist. Aggregation bedeutet auf der semantischen Ebene, daß die Lebensdauer der auf diese Weise eingebetteten Objekte unmittelbar mit der Lebensdauer des enthaltenden Objektes verknüpft ist. Für die Implementation von objektorientierten Programmiersprachen kann Aggregation bedeuten, daß die Repräsentation enthaltener Objekte direkt in enthaltende Objekte eingebettet wird.

Ob es sich bei Referenzen auf andere Objekte um Verknüpfung (Link) oder Aggregation handelt, kann sich allgemein aus dem Objektmodell einer Sprache ergeben[7], implizit aus dem Objekttyp beziehungsweise aus der Deklaration des Attributs[8], oder aber aus

[7] In Java z.B. sind alle Objekte eigenständig. Insbesondere sind die Lebenszyklen verschiedener Objekte nur mittelbar miteinader verknüpft.

[8] In C++ sind Attribute, die nicht explizit als Referenz oder Zeiger deklariert sind, mit dem enthaltenden Objekt aggregiert. Eine Zerstörung des enthaltenden Objektes bewirkt auch eine Zerstörung der darin enthaltenen Objekte, selbst wenn noch anderweitige Referenzen auf sie bestehen.

entsprechenden Schlüsselwörtern, die dem Programmierer zur Verfügung stehen[9].

Im Fall der Verknüpfung gehört die entsprechende Referenz zum Objektzustand, während im Falle der Aggregation das referenzierte (Teil-)Objekt selbst zum Objektzustand gehört. Dies muß bei der Zustandssicherung eines Objektes berücksichtigt werden.

Übertragung des Objektzustands Die Behandlung von primitiven Daten ist relativ einfach durch Kopieren möglich. Unter Umständen müssen sie noch in ein maschinenunabhängiges Format umgewandelt werden, was durch eine einfache Abbildung realisierbar ist.

Aggregierte Objekte können wie hierarchische Datenstrukturen behandelt werden. Schwieriger ist es da mit Verküpfungen zu anderen Objekten. Allgemein gilt, daß Referenzen auf andere Objekte für die Übertragung so umgesetzt werden müssen, daß aus dieser Information wieder gültige Referenzen im Sinne der Programmiersprache erzeugt werden können (siehe 3.4.6).

Dabei ist danach zu unterscheiden, ob die referenzierten Objekte mitmigrieren oder nicht. Für nicht mitmigrierende Objekte bietet sich nach Abschnitt 3.4.1.3 Neubindung, Netzreferenz (siehe 3.4.6) oder Bindungsentfernung an. Für die Erzeugung von Netzreferenzen sind entsprechende Informationen in die Übertragungsdarstellung aufzunehmen, die eine angemessene Behandlung auf der Empfängerseite ermöglichen. Neubindung, insbesondere an zu diesem Zweck neu erzeugte Objekte, kann sich implizit durch die Nichtübertragung entsprechender Attribute ergeben oder aber in der Übertragungsdarstellung beispielsweise durch Angabe von Kriterien näher spezifiziert sein. Bindungsentfernung findet in der Regel nicht statt oder ist ein Spezialfall der Neubindung, indem entsprechende Attribute auf ungültige Werte, zum Beispiel NULL oder NIL, gesetzt werden. Für die Behandlung mitmigrierender Objekte siehe den folgenden Abschnitt über Objektgraphen.

3.4.4.4 Objektgraphen

In vielen mobilen Objektsystemen – zum Beispiel bei denen, die auf die Programmiersprache Java aufsetzen – werden nicht einzelne Objekte migriert, sondern Objektgraphen (siehe Abschnitt 3.3.1). Dabei besteht ein Objektgraph typischerweise aus einem ausgezeichneten Objekt, einem sogenannten Anker, sowie aus direkt und indirekt von diesem referenzierbaren Objekten. Bei diesen Referenzen betrachten wir Verknüpfungen im Sinne des vorigen Abschnittes.

Abgrenzung Zur Bestimmung der Zugehörigkeit eines Objektes zum Objektgraphen lassen sich wie bei der Abgrenzung des Objektzustandes Referenzen heranziehen. Auch hier kann sich die Zugehörigkeit eines Objektes allgemein aus dem Objektmodell der Sprache ergeben, implizit aus dem Objekttyp beziehungsweise aus der Deklaration von Attributen, oder aber aus entsprechenden Schlüsselwörtern, die dem Programmierer zur Verfügung stehen.

[9] In Emerald ist es durch das Schlüsselwort `attached` möglich, Attribute mit dem sie enthaltenden Objekt zu aggregieren.

Während bei der Abgrenzung des Objektzustandes für jedes enthaltene Objekt jedoch die Betrachtung einer *einzelnen* Referenz ausschlaggebend ist, können Objekte im Rahmen von Objektgraphen von mehreren Stellen aus durch Links *gleichberechtigt* referenziert werden. Die Betrachtung unterschiedlicher Referenzen auf das gleiche Objekt kann prinzipiell jedoch zu verschiedenen Bewertungen führen. Hier empfiehlt sich der Übergang zu einer Bewertung auf der Grundlage aller Referenzen auf ein Objekt, insbesondere dann wenn der gemeinsame Zugriff auf Objekte erhalten bleiben soll.

Neben der Betrachtung der Referenzen spielt auch die Transferierbarkeit von Objekten (siehe Abschnitte 3.4.1.2 und 3.3.1), d.h. die Unterscheidung zwischen mobilen und immobilen Objekten, für die Abgrenzung von Graphen eine Rolle.

Übertragung von Objektgraphen Ausgehend vom Ankerobjekt eines Objektgraphen werden alle direkt und indirekt referenzierten Objekte ausfindig gemacht und ihre Zugehörigkeit wird aus den im vorigen Absatz angegebenen Kriterien ermittelt.

Dabei werden Objektreferenzen rekursiv verfolgt. Beim Verfolgen von Objektreferenzen bieten sich ähnliche Vorgehensweisen an wie sie etwa beim Durchsuchen von Bäumen angewandt werden. Jedes gefundene Objekt wird erfaßt und erhält falls nötig einen Identifizierer, der in der Transferdarstellung gültig ist. Anders als beim Durchsuchen von Bäumen muß die rekursive Suche nach Objekten nicht nur bei Blättern d.h. bei Objekten, die ihrerseits keine Referenzen mehr enthalten, enden sondern auch bei bereits bearbeiteten Objekten. Praktisch wird ein Objektgraph dadurch in einen Baum transformiert. Verfahren für die Umwandlung von Objektgraphen in Transferdarstellung können sich darin unterscheiden, ob sie Objekte erst exportieren, wenn sie den Objektgraphen abgegrenzt haben, oder ob beides gleichzeitig geschieht. Bei letzterem Vorgehen können sich Vorwärtsreferenzen auf noch nicht behandelte Objekte von Rückwärtsreferenzen unterscheiden.

Die Objektzustände können wie in Abschnitt 3.4.4.3 beschrieben in Transferdarstellung umgewandelt werden. Dabei fußt die gegenseitige Referenzierung der Objekte auf den Identifizierern, die bei der Graphtraversierung erstellt werden.

3.4.5 Ausführungszustand

In Abschnitt 2.1.2, Seite 10 wurde bereits das Konzept von Steuerflüssen beziehungsweise Aktivitätsträgern erläutert. In Programmen mit Funktions- oder Methodenaufrufen und Rücksprüngen ergibt sich der Fortgang des Steuerflusses nicht allein aus dem Programmcode. Ein Teil der Information, die den Fortgang der Programmbearbeitung – insbesondere Rücksprünge aus Funktionen – erlaubt, muß dynamisch angelegt und verwaltet werden. Diese zu einem Steuerfluß gehörende Information wird Ausführungszustand genannt.

3.4.5.1 Repräsentation von Ausführungszuständen

In den meisten Programmiersprachen ist der Ausführungszustand in einem Stapelspeicher abgelegt, auf dem lokale Variablen aktivierter Methoden zusammen mit Programmzähler-

werten, den Rücksprungadressen zu denen der Steuerfluß nach Abarbeitung der jeweiligen Methode zurückkehrt, liegen. Desweiteren gehört der Programmzähler, der auf die nächste auszuführende Anweisung verweist, zum Zustand des Aktivitätsträgers. Dieser Programmzähler, ebenso wie andere relevante Informationen, zum Beispiel temporäre und maschinenabhängige Daten, liegen oft nicht auf dem Stapel sondern sind in Prozessorregistern gespeichert. Durch die Speicherung von Programmzählerwerten entsteht eine starke Abhängigkeit vom Codeformat, das zur Ausführung verwendet wird. Nach [SJ95] ist der Code Bestandteil des Ausführungszustands.

In verteilten Systemen kommt es zu einer verteilten Speicherung von Ausführungszuständen. Bei einem synchronen entfernten Methodenaufruf wird der aufrufende Aktivitätsträger blockiert. Die Abarbeitung des Aufrufs wird im entfernten System von einem bereits vorhandenen oder neu erzeugten Aktivitätsträger übernommen. Wenn der Aufruf abgearbeitet ist, wird die Kontrolle an den aufrufenden Aktivitätsträger zurückgegeben. Nach [SJ95] migriert der Steuerfluß so von einem Rechner zum anderen. Dabei wird jedoch keine Zustandsinformation übertragen.

3.4.5.2 Starke und schwache Mobilität

Picco unterscheidet bei Mobilen Code Systemen [Pic98] zwischen diesen zwei Formen der Mobilität. Starke Mobilität bedeutet, daß der Ausführungszustand einer Migrationseinheit mit übertragen werden kann, während sich schwache Migration auf die Übertragung von Code und möglicherweise Datenzustand beschränkt.

Bei der **starken Mobilität** (realisiert z.B in Emerald [JLHB88] [SJ95] und Sumatra [ARS96]), kann ein Steuerfluß, der sich zum Zeitpunkt der Migration in der migrierenden Einheit befinden, am Zielort dort fortgesetzt werden, wo die Ausführung vor der Migration gestoppt wurde [SBH96]. Außerdem können auch Rücksprünge aus Funktionen, deren Aufruf noch vor der Migration stattfand, am neuen Ausführungsort durchgeführt werden. Ohne Übertragung der Ausführungszustands, wie zum Beispiel beim entfernten Methodenaufruf, müßte der Steuerfluß dafür zu dem entsprechenden Ausführungsort zurückkehren.

Die Übertragung von Ausführungszuständen ist jedoch sehr aufwendig. Das Format des Ausführungszustandes ist selbst bei interpretierten Sprachen (zum Beispiel Java) oft maschinenabhängig, und somit nur in homogenen Netzen übertragbar. Der Entwurf eines maschinenunabhängigen Transportformates ist sehr aufwendig [SBH96], wenn auch machbar ([SJ95]). Will man bereits vorhandene Sprachen erweitern, so ist dort häufig weder lesender noch schreibender Zugriff auf den Ausführungszustand vorgesehen.

Eine weitere wichtige Frage bei der Übertragung von Ausführungszuständen ist die Zuordnung von Steuerflüssen und Migrationseinheiten. Am einfachsten ist der Fall, wo eine n:1 Zuordnung besteht und alle in der Migrationseinheit aktiven Steuerflüsse mitmigrieren sollen. Steuerflüsse betreffen hier nur Codefragmente und Objekte, die zur Migrationseinheit gehören. Die entsprechenden Ausführungszustände können ohne Zerlegung übertragen und weiterverwendet werden. Problematisch wird es, wenn nur ein Steuerfluß samt Ausführungszustand migriert werden soll, während andere in der Migrationseinheit

aktive Steuerflüsse am aktuellen Ausführungsort belassen werden sollen. Dies kann nicht ohne ganze oder teilweise Duplikation der Migrationseinheit erreicht werden [SBH96].

Sind Steuerflüsse jedoch nicht nur innerhalb der betroffenen Migrationseinheit aktiv, so wird eine Partitionierung des Ausführungszustandes notwendig. Der Teil des Ausführungszustandes, der den Steuerfluß innerhalb eines mobilen Objektes beschreibt, muß mit dem Objekt mitbewegt werden [SJ95].

Aufgrund des oben beschriebenen hohen Aufwands bei der Übertragung von Ausführungszuständen, bieten etliche Systeme nur **schwache Migration** an. In den zu migrierenden Objekten darf zum Zeitpunkt der Migration kein Steuerfluß aktiv sein. Mit der Ausführung der Migration wird solange gewartet bis dieser Zustand eintritt (zum Beispiel bei Voyager). Falls nötig werden Steuerflüsse zum Zweck der Migration beendet. Dies ist insbesondere dann möglich, wenn der Steuerfluß nur innerhalb der Migrationseinheit aktiv ist.

Gerade falls es sich bei der Migrationseinheit um einen mobilen Agenten handelt, so muß sie am Zielort in geeigneter Weise aktiviert werden. Dazu muß ein Steuerfluß generiert und ein Einstiegspunkt vorgegeben werden. Die Einstiegspunkte sind in der Regel durch Methodennamen gegeben. Der Methodenname muß beim Migrationsbefehl angegeben werden, aus einer wie auch immer gearteten Reiseroute hervorgehen oder aber die Migrationseinheit muß eine Methode mit vorgegebenem Namen implementieren.

Neben der automatischen Aktivierung migrierter Objekte am Zielort ist auch die durch entfernte Methodenaufrufe oder aber dort bereits bestehende Steuerflüsse möglich. In Obliq [Car95] wird am Zielort erst ein Steuerfluß generiert zu dem das Objekt dann hinwandern kann.

3.4.6 Objektidentität und Referenzierung

Objektidentität und -referenzierung sind wie zwei Seiten derselben Medaille. Während die Objektidentität eine Eigenschaft von Objekten ist, quasi ein besonderes Attribut (siehe Abschnitt 2.1.1), ist die Referenzierung für die Identifizierung und Adressierung von Objekten zuständig. Dabei ergibt sich die Identität von Objekten nicht allein aus den Referenzen, wie sie im üblichen Sprachgebrauch in der Informatik verstanden werden (Abschnitt 2.1.2). Typischerweise betrachtet man hier die Referenz nicht so sehr als Identifzierungs- sondern vielmehr als Adressierungswerkzeug.

Grundsätzlich stellt sich die Frage, ob beziehungsweise in welchem Maße Objektidentität über Migration hinweg erhalten bleiben kann. Schließlich wird bei dem Vorgang der Migration zunächst einmal eine identische Kopie des migrierenden Objektes angelegt, die dann die Aufgaben des Objektes weiterführt, während die alte Objektinstanz gelöscht wird. Aus dieser Sicht der Dinge heraus geht die Identität eines Objektes verloren. Sie beruht jedoch mehr auf implementationstechnischen Gesichtspunkten als darauf wie Objekte verwendet werden. Da es jedoch bereits in rein lokalen und verteilten Systemen ohne Objektmigration höchst unterschiedliche Implementationstechniken für die Repräsentation von Objektidentität gibt, scheint diese Sichtweise nicht angebracht.

Für den Anwender zählt vor allem, ob Referenzen trotz Objektmigration weiterverwendet werden können. Ist dies möglich, so kann man sicher von einem Fortbestand der Objektidentität sprechen.

Wird die Referenzierbarkeit eines Objektes durch Migration unterbrochen, so ist die Lage nicht so klar. Etliche Systeme schränken daher die Lebensdauer eines Objektes auf die Zeit ein, in der es referenziert wird (Voyager, Emerald). Werden mobile Objekte jedoch zur Realisierung mobiler Agenten verwendet, so schränkt dies die für mobile Agenten geforderte Autonomie ein. In solchen Fällen muß eine Identifizierung beziehungsweise eine *Wiedererkennung* durch systemgenerierte Repräsentationen von Objektidentität (siehe unten) zur Verfügung gestellt werden.

Nicht alle Objekte, die Teil einer Migrationseinheit sind, behalten ihre Identität und Referenzierbarkeit in gleicher Weise. Oft bleibt nur die Migrationseinheit als ganzes oder zum Beispiel bei Objektgraphen ein ausgezeichnetes Objekt global adressierbar und identifizierbar. Einige Objekte können bei der Migration sogar dupliziert werden, wenn sie nicht ausschließlich aus der Migrationseinheit heraus referenziert werden (zum Beispiel bei Voyager).

In Abschnitt 2.3.1 wurde bereits beschrieben, wie die Erfordernisse verteilter Systeme sich auf die Realisierung von Objektreferenzen auswirken. Die Mobilität von Objekten erweitert die Anforderungen an die Implementierung von Objektreferenzierung und -identifizierung noch einmal. Adressierungsinformationen sind nun nicht mehr statisch. Dadurch wird eine dynamische Lokalisierung von Migrationseinheiten beziehungsweise mobilen Objekten notwendig. Objektidentitäten lassen sich nun nicht mehr ohne weiteres aus Adressierungsinformationen ableiten.

3.4.6.1 Interne Repräsentation von Objektidentität

In Programmiersprachen greift man auf Objekte über Bezeichner beziehungsweise Referenzen zu. Das System ist dafür verantwortlich, die *Bindung* zwischen Objekten und entsprechenden Referenzen herzustellen und aufrechtzuerhalten sowie Bindungen zum Beispiel durch Zuweisungsoperatoren zu manipulieren oder durch Gleichheitstests zu überprüfen. Während in Programmiersprachen häufig die Adressierung von Objekten auch für ihre Identifizierung herhalten muß, wird in objektorientierten Datenbanken die Repräsentation der Identität durch normale oder spezielle (d.h. nur systemintern sichtbare) Attribute angewandt. Für mobile Objekte bieten sich Kombination beider Ansätze an.

Abbildung 3.3 gibt einen Überblick über verschiedene Implementationstechniken, und ordnet sie bezüglich ihrer Ortsabhängigkeit ebenso wie ihrer Abhängigkeit von Datenstrukturen und -werten ein. Da die datenabhängigen Implementationen sich auf Datenbanktechniken beziehen, sollen sie uns nicht weiter interessieren.

Bei den strukturierten Bezeichnern (*structured identifiers*) handelt es sich um die in Abschnitt 2.3.1 bereits angesprochenen zusammengesetzten Objektreferenzen wie sie in objektorientierten verteilten Systemen (zum Beispiel in JavaRMI [Sun97]) eingesetzt werden. Sie bestehen aus Anteilen, die den Adreßraum identifizieren, sowie aus einem Anteil, der das Objekt lokal im entsprechenden Adreßraum identifiziert. Durch Objektmigration

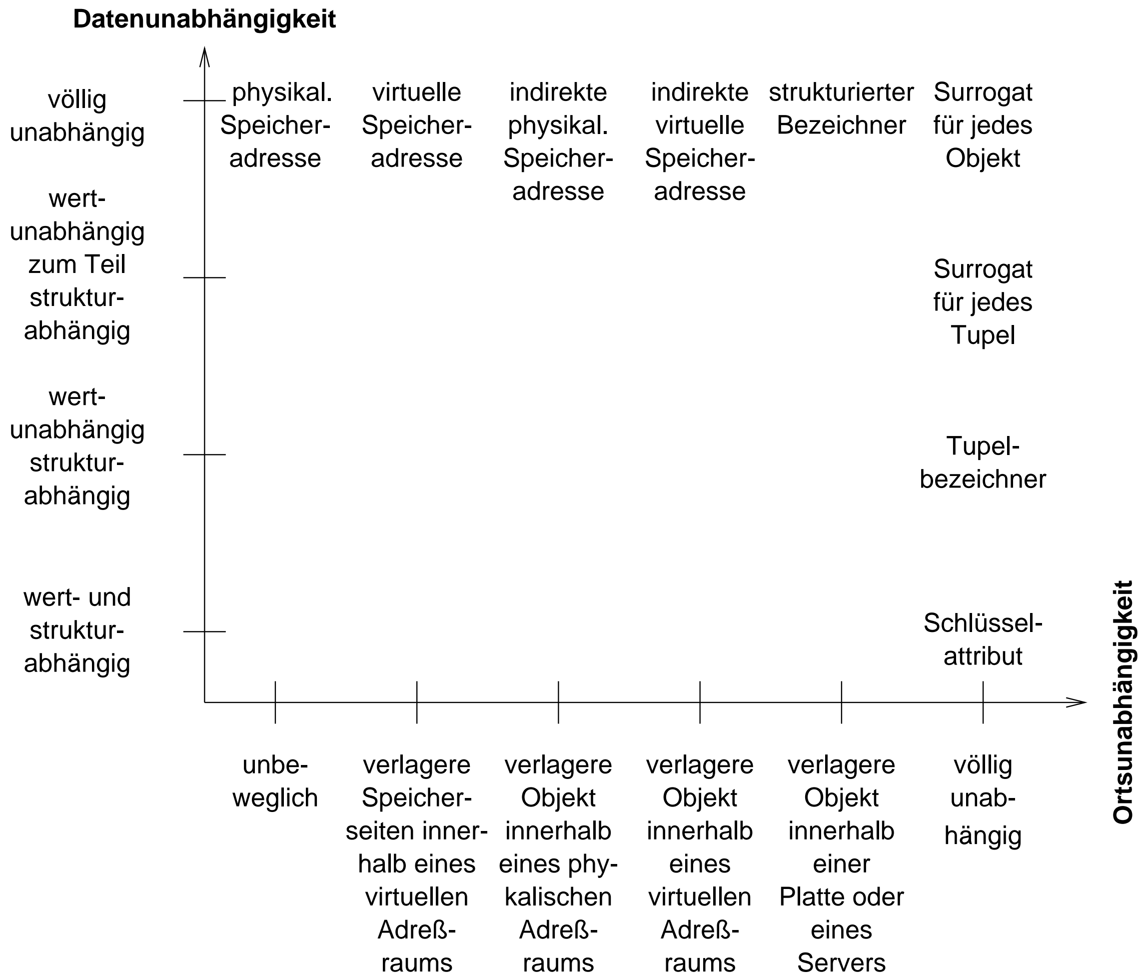

Abbildung 3.3: Implementationstaxonomie (nach [KC86])

verlieren strukturierte Bezeichner die Fähigkeit, Objekte zu adressieren, sofern keine Aktualisierungsmechanismen zur Verfügung stehen.

Surrogate sind nach Copeland [KC86] systemgenerierte, global eindeutige und ortsunabhängige Bezeichner. Sie eignen sich nicht dazu, ein Objekt direkt zu adressieren, sondern müssen bei Verwendung aufgelöst werden. Wesentliche Eigenschaft dabei ist die globale Eindeutigkeit der generierten Bezeichner. Die Erzeugung solcher Bezeichner kann sich dabei am Aufbau strukturierter Bezeichner orientieren, jedoch muß vermieden werden, daß ein Ausführungsort denselben Bezeichner mehrmals erzeugt, auch wenn das dadurch identifizierte Objekte wegmigriert ist, oder gar nicht mehr existiert. Die Wiederverwendung eines Bezeichners ist nur dann erlaubt, wenn garantiert werden kann, daß er von keiner Referenz mehr verwendet wird. Eine weiterer wesentlicher Aspekt des Surrogates ist, daß es auch als spezielles Attribut des entsprechenden Objektes ständig mit diesem assoziiert bleibt.

Einige Systeme (Emerald [JLHB88], SOS [SGM89]) verwenden Kombinationen von Surrogaten und lokalen Adreßzeigern für die Referenzierung von Objekten, um lokal schnelle Adressierbarkeit und gleichzeitig global eindeutige Identifizierung zu erreichen. Dabei muß stets entscheidbar sein, ob der lokale Adreßzeiger gültig ist oder der globale Be-

zeichner verwendet werden muß.

Von Vorteil ist, wenn auch für lokale Referenzierung zumindest eine Stufe der Indirektion vorliegt. Dadurch ergibt sich bei der Migration von Objekten eine zentrale Stelle, die geändert werden kann, anstatt alle Referenzen auf das migrierende Objekt aktualisieren zu müssen.

Funktionalität von Referenzen, wie zum Beispiel Nachrichtenzustellung oder Aktualisierung von Adressierungsinformation kann entweder im Laufzeitsystem (siehe [JLHB88]) oder zumindest teilweise in speziellen Stellvertreterobjekten (engl. *Proxy-Objects*) untergebracht sein (siehe Network Objects[10] [BNOW95], Voyager [Obj97a] und andere Java-basierte Systeme, Aglets [OKO98]). Einige Systeme verwenden auch für lokale Methodenaufrufe stets Stellvertreterobjekte statt lokaler Referenzen, um im Falle der Migration die Umwandlung von Referenzen zu vermeiden. Insbesondere bei Java-basierten Systemen ist eine Manipulation lokaler Referenzen zum Zweck der Mobilität nicht möglich, ohne in die Implementation des Interpreters einzugreifen. Hier bleibt nur die Einführung von Stellvertreterobjekten, die jedoch mit einer enormen Verlangsamung bei der Durchführung von Methodenaufrufen verbunden ist. Eine in [Obj97b] angegebene Beispielanwendung wird dadurch gar um den Faktor 350 langsamer.

3.4.6.2 Lokalisierung von Migrationseinheiten

Migrationseinheiten müssen zum Zweck von Nachrichtenzustellung beziehungsweise Kommunikation oder Kontrolle und Verwaltung lokalisiert werden. In [AO98] werden drei grundlegende Strategien genannt, die dafür in Frage kommen. Nach [MGW97] sind dies Strategien, wie sie auch bei der Prozeßmigration eingesetzt werden.

1. **Weiterleitung** Die Migrationseinheit hinterläßt an jedem Ausführungsort, den es verläßt, ein Weiterleitungsobjekt oder einen Weiterleitungshinweis mit der nächsten Adresse. Bei häufigen Ortswechseln können lange Ketten entstehen, wodurch das Verfahren anfällig gegen Ausfälle einzelner Rechner wird. Wird ein Ausführungsort mehrmals besucht, so können Schleifen entstehen, die das System erkennen und eliminieren sollte.
   Weiterleitungsobjekte können auf zweierlei Art verwendet werden. Entweder verfolgt das System sukzessive die Weiterleitungshinweise bis die Migrationseinheit lokalisiert ist oder aber die jeweils nächste bekannte Adresse wird an den Nachfrager zurückgegeben. Dieser muß dann eine neuen Nachfrage mit der neuen Adresse starten, wo er dann entweder das gesuchte Objekt vorfindet oder aber wiederum ein Weiterleitungsobjekt.

2. **Registrierung** Die Migrationseinheit meldet regelmäßig oder bei jeder Migration ihre neue Adresse an einen zentralen Server. Nachfrager wenden sich mit der Objektidentität beispielsweise in Form eines Surrogates an den entsprechenden Server und erhalten die aktuelle Adressinformation. Alternativ kann eine Migrationseinheit zum Beispiel an dem Ausführungsort, in dem sie erzeugt wurde, ein

[10] Platzhalterobjekte werden in [BNOW95] als *surrogate objects* bezeichnet. Diese Verwendung des Begriffs Surrogat weicht von der oben gegebenen Beschreibung nach Copeland [KC86] ab.

Auskunftsobjekt zurücklassen, an das solche Aktualisierungsmeldungen geschickt werden [MGW97].
Hier entsteht jedoch eine Abhängigkeit von dem Ort an den diese Aktualisierungsinformation zu übermitteln ist. Aufgrund von Nachrichtenlaufzeiten ist außerdem nicht garantiert, daß die so ermittelte Adresse bei Verwendung noch aktuell ist [JLHB88].

3. **Suche** Ist die Menge der in Frage kommenden Ausführungsorte nicht besonders groß, so kommt auch eine Anfrage an alle diese Ausführungsorte (*broadcast*) für die Objektfindung in Frage. Der Ausführungsort, der das Objekt mit der gewünschten Identität enthält, liefert die aktuelle Adresse zurück.

Etliche Systeme verwenden Kombinationen der genannten Strategien, um die jeweiligen Nachteile zu vermeiden. Emerald [JLHB88], das für lokale Netze ausgelegt ist, verwendet zunächst das Konzept der Weiterleitung. Scheitert dies, so wird das Objekt in allen bekannten Ausführungsorten gesucht. Bei dem mobilen Agentensystem Mole [BR98] wird eine aufwendige Kombination von Weiterleitung und Registrierung verwendet, die mobilen Agenten Unabhängigkeit von ihrem Erzeugungsort verschafft, gleichzeitig die Kette von Weiterleitungsobjekten nicht zu lang werden läßt und mit dem Algorithmus auch die Lebensdauer mobiler Agenten kontrolliert.

Das Aglets Agentensystem [OKO98] kann in der Version 1.0 Migrationseinheiten nur über einen Migrationsschritt verfolgen. Nach [AO98] sind in neueren Versionen alle oben beschriebenen Strategien möglich, wobei der Benutzer für jedes Stellvertreterobjekt die Strategie festlegen kann. Die Suche nach Objekten muß er zum Teil sogar selber implementieren.

3.4.6.3 Nachrichtenzustellung

Eng verknüpft mit der Problematik der Lokalisierung von Objekten ist die Zustellung von Nachrichten an mobile Objekte. In [AO98] werden dafür zwei grundlegende Konzepte angeführt.

Das Objekt kann erst mit einem der oben angeführten Verfahren lokalisiert werden. Die Nachricht wird dann direkt an die ermittelte Adresse geschickt.

Die zweite Möglichkeit besteht in einer Kombination mit dem oben genannten Weiterleitungsverfahren. Weiterleitungsobjekte reichen eingehende Aufrufe zur nächsten ihnen bekannten Adresse durch. Das Ergebnis wird dem Aufrufer üblicherweise direkt zugestellt anstatt wieder den Weg über die Weiterleitungsobjekte zu nehmen. Gleichzeitig kann mit dem Ergebnis die neue Adresse der Migrationseinheit übermittelt werden.

3.5 Sicherheitskonzepte

Ein sehr sensibles Thema im Zusammenhang mit mobilem Code und mobilen Anwendungen in offenen Systemen ist die Gewährleistung von Sicherheit. Wir wollen uns dazu

ansehen, welchen Angriffen sowohl Ausführungsorte als auch Migrationseinheiten ausgesetzt sind, und mit welchen Mitteln man diesen entgegenwirken kann.

Mögliche Angriffe lassen sich in Kategorien einteilen, wie sie bereits aus Betriebssystemen, Kommunikationssystemen und klassischen verteilten Systemen bekannt sind. Dies sind unberechtigter Zugang zu geheimen Daten und Ressourcen (*breach of secrecy*), unerlaubte Veränderung von Daten oder Systemzuständen (*breach of integrity*), Nutzung einer fremden Identität (*masquerading*) und Verbrauch aller verfügbaren Systemressourcen, um anderen den Zugang zu einem System zu verwehren (*denial of service*) [VST96].

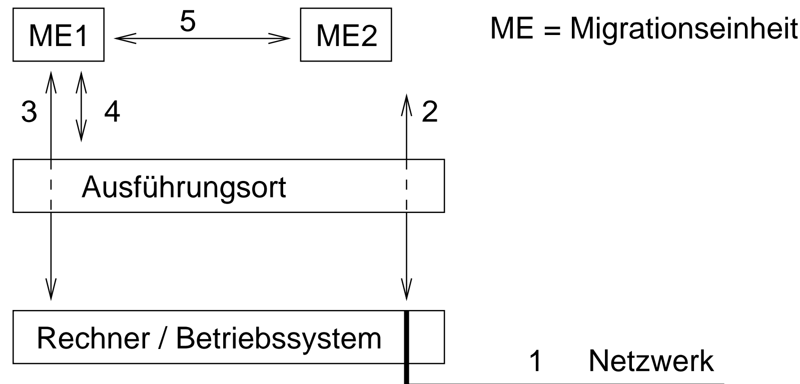

Abbildung 3.4: Sicherheitsaspekte: (1) Übertragungssicherheit, (2) Authentifizierung und Authorisierung, (3) Schutz des Rechners, (4) Schutz des Ausführungsortes, (5) Schutz der Migrationseinheiten (Abb. übersetzt und leicht modifiziert nach [VST96])

In Abbildung 3.4 sind beispielhaft wesentliche Bestandteile eines mobilen Objektsystems skizziert. Die Pfeile und Ziffern deuten an, welche Bestandteile einen Bezug zu den jeweiligen unter dem Bild genannten Sicherheitsaspekten haben.

Übertragungssicherheit, Authentifizierung und Authorisierung

Diese Sicherheitsaspekte sind typisch für verteilte Systeme und Lösungen sind verfügbar. Übertragungssicherheit kann durch Verschlüsselung gewährleistet werden. Authentifizierung kann durch digitale Signaturen erreicht werden. Diese Mechanismen werden bei Nachrichtenübertragung bereits eingesetzt und können auf mobile Objekte angewandt werden.

Sind Migrationseinheiten mit einem Hinweis auf ihren Benutzer, Besitzer oder zumindest Herkunftsort versehen, kann dies von Ausführungsorten und anderen Migrationseinheiten verwendet werden, um sie in Vertrauensstufen einzuordnen und ihre Ausführungsrechte festzustellen. Authorisierung kann durch den jeweiligen Ausführungsort zugeteilt werden, aber auch Migrationseinheiten können ihre Interaktion untereinander abhängig von Authentifizierung machen (siehe Telescript). Kann durch Authentifizierung die Vertrauenswürdigkeit von Migrationseinheiten oder zumindest Codefragmenten gesichert werden, so können die nachfolgend genannten Schutzmechanismen zum Teil weggelassen werden.

Bei der Verwaltung der Authorisierung ist auf eine saubere Trennung von Migrationseinheiten mit unterschiedlichen Ausführungsrechten zu achten. Die freie Kommunikation zwischen ihnen kann dazu führen, daß sie wie eine Einheit mit der Vereinigungsmenge ihrer Ausführungsrechte agieren können. Gestattet man einer Migrationseinheit beispielsweise keinen Dateizugriff dafür aber Zugang zum Netzwerk, während man einer anderen keinen Netzwerkzugang dafür aber Dateizugriffe erlaubt, so untergräbt jegliche Kommunikationsmöglichkeit zwischen den beiden Einheiten die Intentionen der Rechtevergabe (Beispiel aus [VST96]).

Schutz des Rechnersystems

Der Zugang zu Rechnersystemen wird von Betriebssystemen verwaltet. Die in Betriebssystemen realisierten Prozeß- beziehungsweise Adreßraumbezogenen Sicherheitskonzepte sind nicht geeignet, Aktivitäten verschiedener Herkunft, die innerhalb eines Adreßraums beziehungsweise innerhalb der Ausführungsumgebung ablaufen, unterschiedliche Rechte zuzuweisen. Die einzige Möglichkeit, aus Betriebssystemsicht den Zugriff auf das System einzuschränken, besteht darin, die Rechte der gesamten Ausführungsumgebung einzuschränken. Diese Möglichkeit ist jedoch unflexibel und scheidet aus, wenn einzelnen Migrationseinheiten bekannter Herkunft uneingeschränkter Zugang zum Betriebssystem gewährt werden soll. Der Schutz des Rechnersystems obliegt dann der Ausführungsumgebung, die keine direkten oder unkontrollierten Systemaufrufe von Migrationseinheiten zulassen darf.

Schutz der Ausführungsumgebung

Die Ausführungsumgebung muß sich selbst vor den Migrationseinheiten, deren Code sie ausführt, schützen. Es muß vor allem gewährleistet werden, daß geladener Code keine unkontrollierten Zugriffe auf Interna erhält oder gar die Ausführungsumgebung modifizieren kann. Dienste werden nur über eine definierte Schnittstelle angeboten, und sollten so eingeschränkt sein, daß keiner der oben angeführten Attacken durchgeführt werden kann.

Nach Möglichkeit sollten keine *gefährlichen* Anweisungen wie zum Beispiel direkte Speicherzugriffe im Sprachumfang des Codeformates, das zur Übertragung verwendet wird, enthalten sein. Dies ist jedoch bei etlichen vorhandenen Sprachen nicht gegeben oder nicht vollständig realisierbar. Standardbeispiel für ein Sprachkonstrukt, das indirekt freien Zugriff auf Speicherbereiche erlaubt, sind Felder. Felder sind nur in Kombination mit einer zur Laufzeit ausgeführten Indexüberprüfung sicher.

Gelegentlich werden die in objektorientierten Sprachen gegebenen Mechanismen zur Datenkapselung als Schutzmechanismen angeführt. Dies funktioniert am besten, wenn das zur Übertragung verwendete Codeformat semantisch völlig äquivalent mit dem Quellcode ist, wie das zum Beispiel bei abstrakten Systaxbäumen oder Token der Fall ist (siehe 3.2). Da die Regeln der Datenkapselung auf Quellcodeebene festgelegt sind, ist die Kontrolle ihre Einhaltung nach der Übersetzung in Maschinen- oder Bytecode sehr aufwendig und fehleranfällig.

Für die kontrollierte Ausführung von als unsicher eingestuftem Code sind folgende Ansätze bekannt:

1. Dynamische Kontrolle von Code bei der Interpretation (zum Beispiel Java Security Manager [LY96]). Dieser Ansatz ist sehr flexibel, da Sicherheitsbeschränkungen sehr detailliert und dynamisch festgelegt werden können. Größter Nachteil ist die Beeinträchtigung der Ausführungsgeschwindigkeit.

2. Überprüfung von Code vor der Ausführung. Hier kann überprüft werden, ob statische Bedingungen eingehalten werden. Dazu gehört zum Beispiel die Kontrolle der oben bereits genannten Datenkapselung, Typprüfung oder Analyse des Kontrollflusses (zum Beispiel Java Code Verifier [LY96]). Die statische Überprüfung von Code allein garantiert im allgemeinen keine sichere Ausführung von Code, kann jedoch die dynamische Kontrolle der Ausführung entlasten. Wird Code vor der Ausführung übersetzt, so findet eine derartige Überprüfung zum Teil automatisch statt.

3. Modifikation von Code (*Software Fault Isolation* [WLAG93]). Dieses Verfahren ist für Maschinencode geeignet. Eine Migrationseinheit erhält bestimmte Speichersegmente als Schutzbereiche zugewiesen. Alle Speicherzugriffe oder Sprungbefehle, die nicht statisch überprüft werden können, werden um Codesequenzen erweitert, die entweder überprüfen, ob die entsprechenden Adressen in die richtigen Speichersegmente verweisen, oder das Setzen der erlaubten Speichersegmente erzwingen. Während die erste Methode die Verfolgung von Segmentverletzungen erlaubt, begrenzt die zweite Methode (auch *sandboxing* genannt [Kat96]) deren Folgen auf festgelegte Bereiche.

4. Der Schutz von Speicherbereichen durch Hardwareunterstützung (zum Beispiel in PLANET [Kat96]) ist verwandt mit der von Betriebsystemen eingesetzten Trennung von Adreßräumen mit Hilfe der virtuellen Speicherverwaltung. Durch Sperrung von Speicherbereichen lassen sich auch Migrationseinheiten, die in Form von Maschinencode vorliegen, voneinander und von der Ausführungsumgebung abgrenzen. Aufrufe oder Zugriffe in gesperrte Speicherbereiche können abgefangen und vor Ausführung überprüft werden.

Schutz der Migrationseinheiten

Mindestens ebenso großen Gefahren wie Ausführungsorte, sind die Migrationseinheiten selbst ausgesetzt. Als mögliche Angreifer kommen neben dem Kommunikationssystem (siehe oben) andere Migrationseinheiten ebenso wie die Ausführungsumgebung in Frage.

Insbesondere gegenüber der Ausführungsumgebung können Migrationseinheiten praktisch nicht geschützt werden. Sie hat uneingeschränkten Zugang zu den Daten ebenso wie zum Code, der häufig auch ein schützenswertes Gut ist. Auch könnten nicht erbrachte Dienste in Rechnung gestellt oder gezielt Fehlinformationen an die Migrationseinheit gegeben werden. Hier hilft im Prinzip nur die Herstellung von Vertrauen durch Authentifizierung.

Der Schutz gegen andere Migrationseinheiten muß von der Ausführungsumgebung unterstützt werden. Dazu können Schutzbereiche eingerichtet werden, die eine oder mehrere Migrationseinheiten enthalten können. Interaktionen innerhalb eines Schutzbereiches können ohne Kontrolle durchgeführt werden. Für Kommunikation über Schutzbereiche hinweg muß gewährleistet werden, daß nur die jeweils dafür vorgesehenen Schnittstellen der Migrationseinheiten verwendet werden. Während bei gewöhnlichen Methodenaufrufen das aufgerufene Objekte den Aufrufer nicht kennt, kann es hier sinnvoll sein, eine gegenseitige Authentifizierung einzuführen (siehe Telescript [VST96, Seite 185]).

Kapitel 4

Auswertung mobiler Objektsysteme

Die in Kapitel 3 beschriebenen Aspekte mobiler Objektsysteme sollen hier herangezogen werden, um nach einheitlichem Muster einige der vielen realisierten mobilen Objektsysteme zu analysieren. Abbildung 4.1 zeigt die Aspekte noch einmal im Überblick.

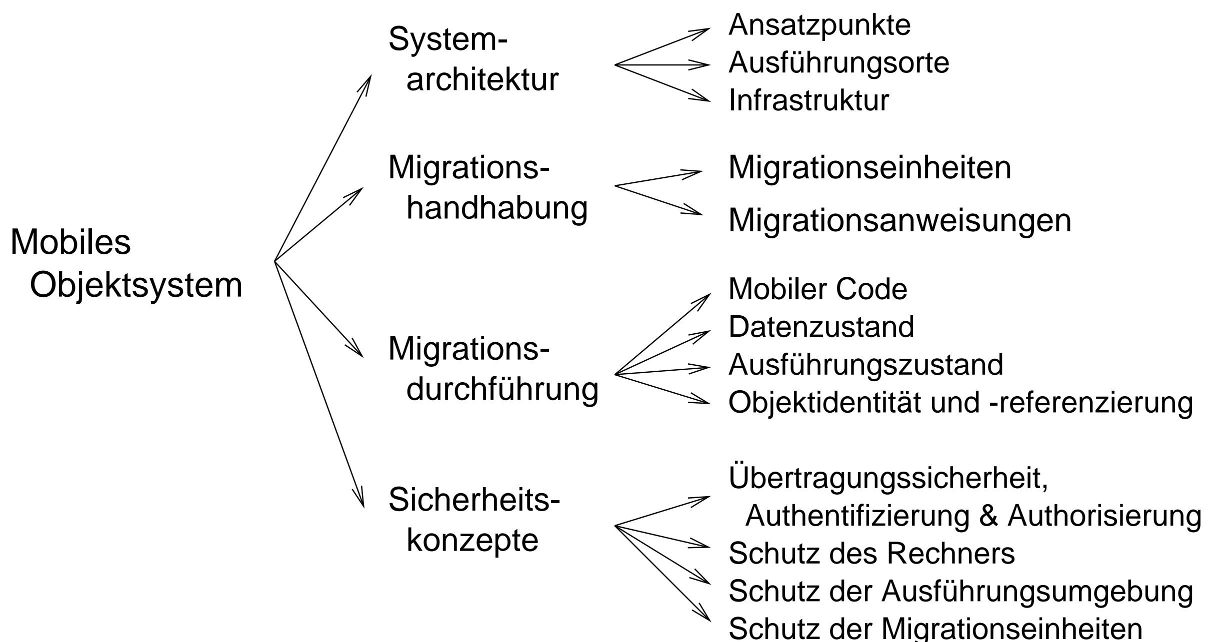

Abbildung 4.1: Auswertungskriterien für mobile Objektsysteme

4.1 Emerald

Literatur: [BHJL86], [HR91], [JLHB88], [Hut96], [SJ95]

Emerald ist eine objektbasierte Sprache und ein System für die Entwicklung verteilter Programme. Besonders interessant an Emerald ist dabei das uniforme Objektmodell und die aufwendige Unterstützung starker Mobilität, das heißt der Übertragung

von Ausführungszuständen. Ein wichtiges Ziel beim Entwurf von Emerald war, trotz Unterstützung von Mobilität eine hohe Ausführungsgeschwindigkeit für Programme zu gewährleisten. Der angestrebte Einsatzbereich von Emerald sind lokale Netze mit bis zu hundert Rechnern [JLHB88].

Objekte in Emerald werden durch die Ausführung von Objektkonstruktoren erzeugt. Diese sind Emerald Ausdrücke, die eine Beschreibung des zu erzeugenden Objektes, also Attribute und Operationen enthalten. Will man mehrere Instanzen eines Objektes, so bettet man einen entsprechenden Konstruktor in eine Operation eines anderen Objektes ein. Die Operation des Erzeugerobjektes gibt das erzeugte Objekt als Ergebnis zurück. Die Objektbasierung wird in Emerald sehr weit getrieben. Sogar abstrakte Datentypen sind als Objekte erster Klasse konzipiert [HR91], [Hut96]. Klassen und Vererbung finden in der Syntax von Emerald auch Platz. Da Klassen allerdings lediglich auf Erzeugerobjekte abgebildet werden und als syntaktisches Element entbehrlich sind, bezeichnen die Entwickler die Sprache nicht als objektorientiert sondern objektbasiert.

4.1.1 Systemarchitektur

Das Emerald System besteht aus Compiler und Interpreter [Hut96]. Die Funktionalität für Mobilität ist voll in die Sprache integriert und wird von Compiler und Interpreter erbracht, den eng miteinander gekoppelt sind. In [BHJL86] wird jedoch eine prototypische Implementation angeführt, deren Compiler Maschinencode erzeugt. Dort wird das Laufzeitsystem nicht als Interpreter sondern als Laufzeitkern bezeichnet.

Die Interpreter oder Laufzeitkerne fungieren als Ausführungsorte, in der Emerald-Terminologie als Knoten bezeichnet. Jeder Knoten ist durch Rechneradresse und TCP-Portnummer adressiert. Diese Adressierung spielt allerdings nur beim Start eines Knotens eine Rolle. Knoten kooperieren in Gruppen. Jeder neue Knoten muß sich bei einem beliebigen Knoten einer Gruppe anmelden.

Für die ausgeführten Programme werden Knoten durch vordefinierte Objekte repräsentiert. In jedem Knoten existiert eine Liste anderer bekannter Knoten. Mit der oben genannten Rechneradressen und TCP-Nummern kommen Emerald Programme nicht in Berührung.

4.1.2 Handhabung mobiler Objekte

Migrationseinheit in Emerald ist das programmiersprachliche Objekt. Emerald unterstützt ein uniformes Objektmodell, das heißt daß es auf programmiersprachlicher Ebene keine Unterscheidung zwischen mobilen und immobilen Objekten gibt.

Objekte bekommen bei Ihrer Erzeugung einen eindeutigen Namen zugewiesen. Außerdem können Objektreferenzen Konstanten, globalen Variablen oder Attributen anderer Objekte zugewiesen werden. All diese Emerald-Konstrukte können gleichermassen zum Zugriff auf Objekte verwendet werden.

Die Migration von Objekten kann durch Operatoren kontrolliert werden. Die Operatoren `move` und `fix` veranlassen die Migration von Objekten. Als Operanden erhalten

sie das zu migrierende Objekt sowie ein Zielangabe. Als Zielangabe kommen dabei beliebige Objektreferenzen in Betracht, insbesondere natürlich auch Knoten-Objekte. Der `move`-Operator ist dabei nur als Hinweis an das System zu verstehen. Ob und wann die Migration durchgeführt wird, obliegt dem System. Der `fix`-Operator hingegen erzwingt eine Migration des Objektes zu dem angegebenen Ziel. Außerdem bewirkt er, daß das Objekt danach nicht mehr weitermigriert werden kann. Dies wird erst durch Anwendung des `unfix`-Operators wieder möglich. Die atomare Kombination der Operatoren `fix` und `unfix` ist der `refix`-Operator [HR91].

Neben den besprochenen Operatoren können auch Methodenaufrufe eine Migration der Parameterobjekte bewirken. Normalerweise werden bei entfernten Methodenaufrufen nur Objektreferenzen übertragen. Steht bei dem Methodenaufruf jedoch das Schlüsselwort `move` vor einem Parameter, so wird das entsprechende Objekt zum aufgerufen Objekt migriert. Diese Variante der Migration dient vor allem der Optimierung durch die Zusammenfassung von Kommunikationsvorgängen für den entfernten Methodenaufruf und die Objektmigration.

Ein Emerald-Objekt kann einen eigenen Aktivitätsträger besitzen. Dadurch können mehrere Aktivitätsträger gleichzeitig in Ausführungsorten und auch in Objekten aktiv sein. Die Migration von Objekten hat auf Aktivitätsträger keinen Einfluß. Sie werden nahtlos fortgeführt.

4.1.3 Durchführung der Migration

Da Emerald Objektbasiert ist, ist jedes Objekt und somit jede Migrationseinheit mit genau einem Codefragment assoziierbar. Ausführungszustände werden transparent zerlegt und mit Objekten mitmigriert.

4.1.3.1 Mobiler Code

Emerald verwendet als Code Bytecode oder Maschinencode, der vom Compiler aus Emerald-Quellcode generiert wird. Dadurch funktionierte das System ursprünglich nur in homogenen Umgebungen. In [SJ95] ist jedoch eine Erweiterung, vorgesehen, die Maschinencode für verschiedene Rechner zur Verfügung stellt und damit zu einem gewissen Grad den Einsatz in heterogenen Systemen erlaubt.

Die Einheit der Codefragmentierung ist das Objekt. Codefragmente werden intern sogar wiederum als Objekte repräsentiert und referenziert.

4.1.3.2 Datenzustände

Der Datenzustand von Objekten besteht aus primitiven Daten und Referenzen auf andere Objekte. Das genaue Übertragungsformat der Datenzustände ist in den untersuchten Quellen nicht beschrieben. Detailliert beschrieben ist hingegen die Repräsentation von Objektzuständen im Arbeitsspeicher. Fest steht, daß die primitiven Daten dank der homogenen Umgebung kopiert werden können, während Objektreferenzen modifiziert werden müssen. Alle für das Objekt benötigten Daten werden einer Nachricht übermittelt.

4.1.3.3 Objektidentität und Referenzierung

Intern unterscheidet Emerald drei Arten von Objekten, insbesondere nach der Art wie sie referenziert werden. Es gibt globale Objekte, die frei bewegt werden können. Daneben gibt es sogenannte lokale Objekte, die nur als mit anderen Objekten mit diesem mitmigrieren können. Schließlich gibt es noch direkte Werte, die direkt in andere Objekte eingebettet sind. Die Unterscheidung zwischen globalen und lokalen Objekten trifft der Compiler bei der Generierung des Byte- oder Maschinencodes zum Zweck der Optimierung. Diese Entscheidung kann vom Programmierer dahingehend beeinflußt werden, das Objekte durch Verwendung des Schlüsselwortes `attached` als lokal deklariert werden.

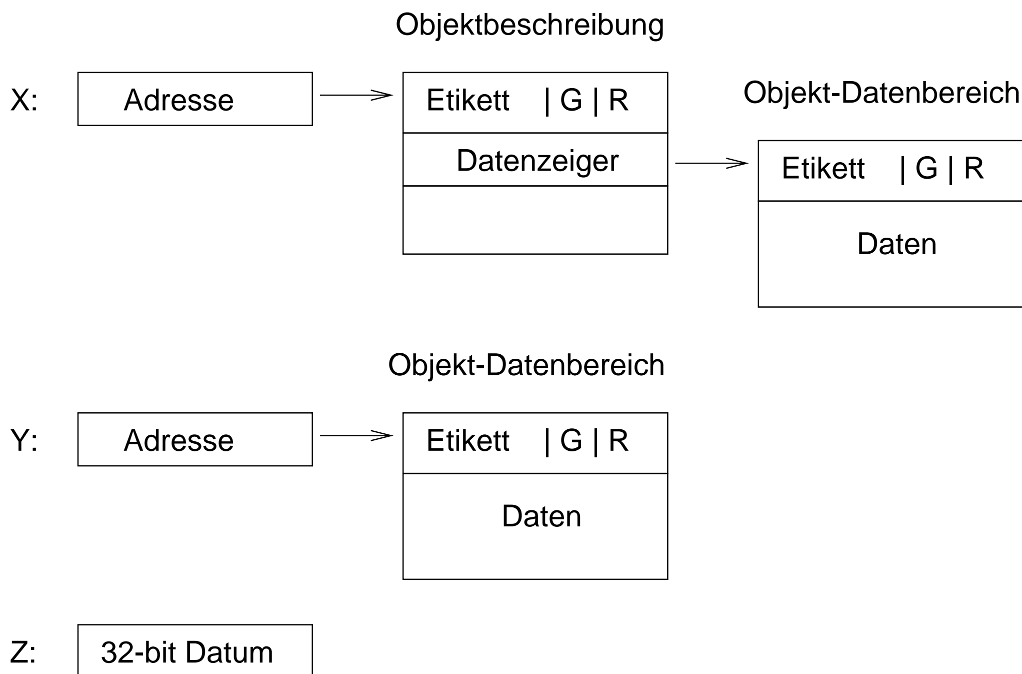

Abbildung 4.2: Emerald Adressierungsstrukturen (nach [JLHB88])

Abbildung 4.2 zeigt schematisch die Implementationen der eben angesprochenen Adressierungsvarianten. Dabei sind X, Y und Z jeweils Variablen, die ein globales, ein lokales und ein direktes Objekt referenzieren. Die Etiketten dienen zur Unterscheidung der Fälle. Außerdem enthalten sie Bits, die signalisieren ob hier ein globales Objekt referenziert wird (das G Bit) sowie ob das Objekt in der lokalen Ausführungsumgebung residiert (das R Bit).

Jede Objektbeschreibung enthält zudem einen global eindeutigen Objektidentifizierer (OID). Falls das R-Bit gesetzt ist, enthält die Objektbeschreibung wie in Abbildung 4.2 gezeigt einen Zeiger auf den Datenbereich. Ansonsten steht hier eine Weiterleitungsadresse, die einen Zeitstempel enthält und auf den nächsten bekannten Knoten des Objektes verweist. Jeder Knoten verfügt über Listen, die OIDs Objektbeschreibungen zuordnet. Die Zeitstempel in den jeweiligen Objektbeschreibungen zeigen dem Suchalgorithmus an, ob die jeweilige Weiterleitungsinformation noch aktuell ist. Objekte, die nicht durch Weiterleitungsobjekte auffindbar, werden durch globale Anfrage an alle Ausführungsorte lokalisiert.

Wird ein globales Objekt migriert so, wird an den Anfang der zu übertragenden Datenpaketes sein Datenbereich kopiert. Im Datenbereich enthaltene Zeiger auf andere Objekte werden durch deren OID ersetzt. Zeiger auf lokale Objekte werden durch deren Datenbereich ersetzt.

4.1.3.4 Ausführungszustände

Ausführungszustände speichert Emerald als Stapel von Aktivierungssätzen. Für jedes Objekt wird Information geführt, die angibt welche Aktivierungssätze seine Methoden aktivieren. Wird Objekt, das durch einen oder mehrere Steuerflüsse aktiviert ist, migriert, so werden alle betroffenen Ausführungszustände in mehrere Teile aufgespalten, und die entsprechenden Teile mit dem Objekt mitmigriert [JLHB88]. Die Situation nach der Migration ist so, als wäre das Objekt von vornherein am Zielort gewesen und die Aktivierungen hätten durch entfernte Methodenaufrufe stattgefunden, für die jeweils ein eigener Steuerfluß erzeugt worden wäre.

4.2 Java

Literatur: [Sun97], [Sun98], [LY96], [Sun], [GM96], [KJS96], [MHM96], [Gri98b], [Eck97], [Fla96], [MF96]

Java ist eine von der Firma Sun Microsystems entwickelte objektorientierte Programmiersprache. Neben Klassen mit Einfachvererbung kennt sie auch abstrakte Klassen und Schnittstellen (siehe Abschnitt 2.1.1). Java bietet plattformunabhängigen mobilen Code und seit der Version 1.1 [Eck97] die sogenannte Serialisierung von Objektgraphen an. Mit dem ebenfalls seit Version 1.1 eingeführten entfernten Methodenaufruf RMI (Remote Method Invocation) stellt Java ein verteiltes objektorientiertes System dar. Darüberhinaus bietet es ein aufwendiges Sicherheitskonzept für die sichere Ausführung mobilen Codes zum Beispiel in Form von sogenannten Applets (siehe zum Beispiel [Fla96]), die besonders für den Einsatz im Internet konzipiert wurden.

Die genannten Eigenschaften zusammen mit der weiten Verbreitung dieser Programmiersprache haben dazu geführt, daß sie als Grundlage für viele mobile Objektsysteme verwendet wird. Obwohl Java keine Objektmigration unter Erhaltung der Identität erlaubt und somit kein mobiles Objektsystem im Sinne der Definition aus Abschnitt 3, Seite 17 ist, sollen hier einige Aspekte von Java beschrieben werden, auf denen andere mobile Objektsysteme aufbauen.

4.2.1 Systemarchitektur

4.2.1.1 Ansatzpunkte

Kernstück von Java ist neben der Sprachsyntax, ein Compiler, der Quellcode in sogenannten Bytecode umsetzt, und eine Laufzeitumgebung , die diesen Bytecode in der Regel

interpretativ ausführt (siehe Abbildung 4.3). Beim Einsatz von RMI kommt ein zusätzlicher Compiler zum Einsatz, der für die Erzeugung des Codes für Stellvertreterobjekte verantwortlich ist [Sun97].

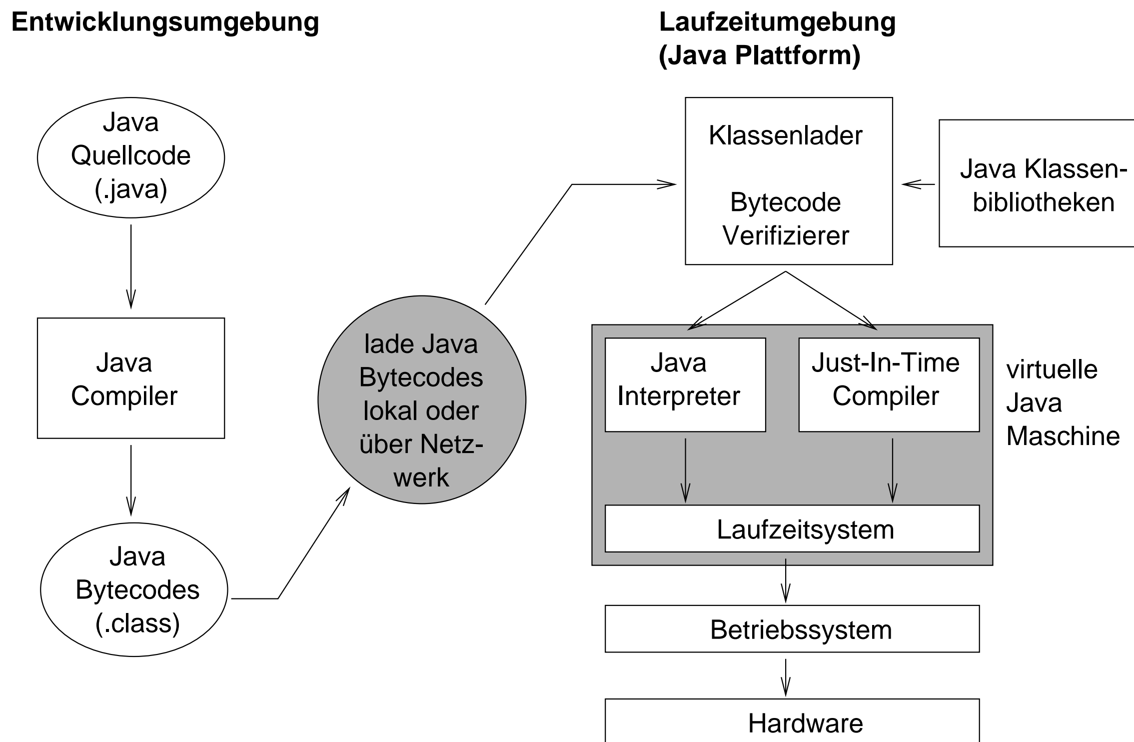

Abbildung 4.3: Java Architektur (aus [KJS96])

4.2.1.2 Ausführungsorte

Die Laufzeitumgebung, *Java Platform* genannt, besteht aus der sogenannten *virtuellen Maschine* und einer Programmierschnittstelle, der *Java API* [KJS96]. Die virtuelle Maschine stellt einen standardisierten abstrakten Rechner dar. Sie kann in Hard- oder Software realisiert werden, definiert das Format des von ihr ausgeführten Bytecodes und übernimmt Aufgaben wie zum Beispiel Speicherverwaltung. Der auszuführende Code kann von der lokalen Platte oder aus dem Netz geladen werden.

Abbildung 4.4 zeigt diverse Möglichkeiten für die Einbettung der Java Plattform in verschiedene Umgebungen.[1] Die Einbettung der Laufzeitumgebung in viele verbreitete Betriebssysteme insbesondere als Teil von WWW-Browsern macht Java quasi plattformunabhängig und hat sicherlich am meisten zur Verbreitung von Java beigetragen.

Die Java Programmierschnittstelle ist der für den Anwendungsentwickler sichtbare Teil des Systems. Dabei handelt es sich um Klassenbibliotheken, die unter anderem Funktionalität wie zum Beispiel Ein-/Ausgabe, Netzwerkzugang oder graphische Benutzerschnittstellen zur Verfügung stellen.

[1] In [MHM96] ist die virtuelle Maschine nicht oberhalb von JavaOS angesiedelt, sondern sie ist Grundlage für den Großteil der JavaOS-Funktionalität.

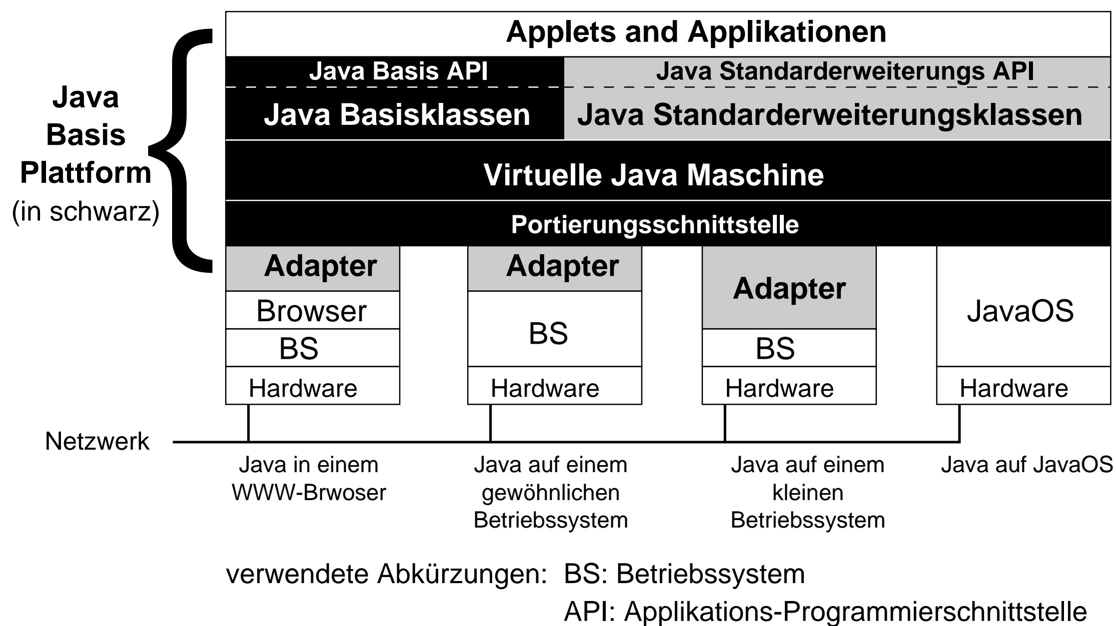

Abbildung 4.4: Java Platform (aus [KJS96])

4.2.1.3 Infrastruktur

Neben der Java Plattform selbst können *http* oder *ftp* Server zum Einsatz kommen, da für den Transport mobilen Codes die Übertragungsprotokolle http und ftp verwendet werden. Dazu muß der zu ladende Code auf entsprechenden http oder ftp Servern bereitliegen. Die Codefragmente werden durch sogenannte URLs[2] adressiert.

Für den entfernten Methodenaufruf kommen auch Namensserver, sogenannte RMIRegistry Server zum Einsatz. Bei ihnen können Netzreferenzen auf Objekte hinterlegt und mit einem Namen assoziiert werden. Andere Anwendungen können solche Objektreferenzen abholen. Die bei Namensservern hinterlegten Objektreferenzen werden ebenfalls durch URLs identifiziert (zum Beispiel `rmi://nameserver:portnr/objektname`).

4.2.2 Handhabung von Java-Objekten

Im Rahmen des entfernten Methodenaufrufes können Objekte als Parameter übergeben werden. Dabei wird der vom entsprechenden Objekt ausgehende Objektgraph serialisiert und übertragen. Eine Entfernung der übertragenen Objekte aus dem Adreßraum des Aufrufers findet nicht statt.

Alle zu übertragenden Objekte müssen Instanzen von Klassen sein, die die Schnittstellen `java.io.Serializable` oder `java.io.Externalizable` aus dem Ein-/Ausgabe Modul der Java Programmierschnittstelle implementieren (siehe [Sun98]). Bei der `Serializable` Schnittstelle handelt es sich dabei nicht wie sonst üblich um die

[2] Eine URL (Unary Ressource Locator) besteht zumindest aus drei oder vier Teilen, die ein Übertragungsprotokoll, einen Rechner, gegebenfalls eine TCP-Portnummer und einen Pfad angeben z.B. http://host:port/pfad . Durch URLs werden z.B. Dateien auf http oder ftp Servern referenziert.

Festlegung von zu implementierenden Methoden. Die Schnittstellendefinition selbst ist leer. Stattdessen wird sie wie ein Schlüsselwort verwendet, das eine entsprechende Klasse als serialisierbar markiert.[3]

Objekte, die die genannten Schnittstellen nicht implementieren, können nicht übertragen werden. Der Versuch, sie in entfernten Methodenaufrufen als Parameter zu übergeben, führt zum Abbruch des Aufrufs und einer Fehlermeldung. Auch Objekte, die für entfernte Methodenaufrufe zugänglich sind, also Netzreferenzen zur Verfügung stellen, können nicht kopiert werden. Hier werden beim entfernten Methodenaufruf statt der Objekte selbst die Stellvertreterobjekte als Parameter übertragen [Sun97].

Die Lebensdauer von Java-Objekten ergibt sich aus ihrer lokalen und entfernten Referenzierung. Sobald ein Objekt nicht mehr referenziert wird, ist es zur garbage collection freigegeben. Eine explizite Kontrolle der Lebensdauer findet nicht statt.

4.2.3 Durchführung der Migration

Die Übertragung von Code und Datenzuständen findet bei Java getrennt statt. Die entsprechenden Mechanismen lassen sich unabhängig voneinander nutzen. Symbolische Information in den serialisierten Objekten identifiziert die benötigten Codefragmente. Eine Übertragung von Ausführungszuständen findet nicht statt.

4.2.3.1 Mobiler Code

Codeformat Java verwendet zur Übertragung und Ausführung sogenannten Bytecode, eine dem Pascal-P-Code [Nor81] verwandte maschinencodeähnliche Sprache für Maschinen mit Stapel-Architektur (0-Adreß-Maschine) [RP97]. In der Regel wird Bytecode von der interpretiert. Etliche Implementationen der Java Plattform übersetzen Bytecode jedoch zunächst in Maschinencode für den entsprechenden Rechner und führen diesen dann aus. Diese Strategie wird Just-In-Time Compiling genannt (siehe Abbildung 4.3). Ein neueres Verfahren kombiniert Interpretation und Übersetzung von Code. Zunächst wird der geladene Code interpretiert, um Anfangsverzögerungen zu vermeiden. Während der Abarbeitung wird der Code dann besonders auf häufig durchlaufene Teile untersucht und diese werden gezielt und hochoptimiert nach Maschinencode übersetzt [Gri98b].

Codefragmente Einheit der Codefragmentierung ist die Klasse. Beim Compilieren wird für jede Klasse und jede Schnittstelle eine eigene Datei mit dem entsprechenden Bytecode angelegt. Neben dem Code an sich enthalten diese Dateien auch symbolische Informationen. Dies sind zum Beispiel Klassennamen, die Namen der implementierten Schnittstellen, außerdem Methodennamen sowie Namen von Attributen [LY96]. Dies erlaubt eine Verzögerung des Bindens bis zum Zeitpunkt der Ausführung.

Codereferenzierung Klassen sind in ein hierarchisches, baumartig strukturiertes Modulkonzept eingebettet. Die Zugehörigkeit einer Klasse zu einem Modul – bei Java *package* genannt – läßt sich an Ihrem vollen Namen (englisch: fully qualified name) ablesen. Die

[3] Die Verwendung von Schnittstellen statt Schlüsselwörtern zur Markierung von Objekteigenschaften, die nicht direkt die Implementation von Methoden betreffen, ist umstritten [Eck97].

Modulhierarchie wird auf Dateibäume abgebildet. Jedem package ist ein Verzeichnis zugeordnet, das die Dateien der dazugehörigen Klassen enthält.

Beispiel: Der Bytecode der Klasse mit dem vollen Namen: `bei.spiel.Klasse` befindet sich in einer Datei mit dem Pfad: `bei/spiel/Klasse.class`.

Codeverwaltung Der Code von Klassen wird erst dann in die Ausführungsumgebung geladen, wenn er zur Ausführung benötigt wird. Für das Laden von Klassen sind sogenannte Klassenlader (englisch: *ClassLoader*, siehe Abbildung 4.3) verantwortlich. Dies sind Instanzen von Klassen, die von `java.lang.ClassLoader` abgeleitet sind. Sie implementieren unter anderem eine Methode namens `loadClass(String name)`, die die Klasse gewünschten Namens als Instanz der Klasse `java.lang.Class` das heißt als Metaobjekt zurückgibt. Dieser Rückgabewert läßt sich dazu verwenden, entsprechende Objekte zu instantiieren.

Normalerweise kommt der Anwendungsprogrammierer mit ClassLoadern nicht direkt in Berührung. Diese werden von der Java Plattform selbst benutzt, um Objekte zu instantiieren [Sun]. Dabei können mehrere ClassLoader gleichzeitig aktiv sein. Referenziert eine Klasse eine Methode oder einen Konstruktor einer anderen Klasse, so wird die referenzierte Klasse mit Hilfe des gleichen ClassLoaders geladen, wie die referenzierende. Will der Anwendungsprogrammierer Klassen laden, die nicht mit den vorhandenen ClassLoadern geladen werden können, so muß er einen neuen ClassLoader instantiieren und diesen explizit für das Laden einer neuen Klasse verwenden. Auf diese Weise entsteht für jede Quelle von Klassen ein eigener Namensraum neben dem immer gegebenen Namensraum der Klassen, die von der lokalen Platte geladen werden [GM96].

Konkrete Aufgabe für ClassLoader ist das Finden und Laden der Datei für die gesuchte Klasse. Der Anwendungsprogrammierer kann dies selbst implementieren oder aber auf einen von der Java API bereits zur Verfügung gestellten ClassLoader zurückgreifen. Dies sind der Standard ClassLoader, der Klassen von der lokalen Festplatte lädt, der AppletClassLoader, der die Klassen von Applets von http oder ftp Servern lädt sowie der RMIClassLoader, der Klassen für Stellvertreterobjekte oder Parameter ebenfalls mit Hilfe von http oder ftp laden kann [Sun97].

4.2.3.2 Datenzustand

Datenformat Für die Übertragung von Datenzuständen bietet Java die sogenannte Serialisierung von Objektgraphen an. Dabei wird die Zustandsinformation in eine flache Transferdarstellung gebracht (siehe Abschnitt 3.4.4) und einem Ausgabestrom übergeben.

Kontrolle der Serialisierung Ein- und Ausgabevorgänge für Dateien, Netzwerkverbindungen, Bildschirmausgaben, Tastatureingaben und so weiter werden in Java durch sogenannte Ströme erledigt. Dementsprechend wird auch der Export und Import von Objektgraphen durch entsprechende Ströme realisiert. Objektgraphen werden durch Übergabe an die Methode `writeObject(Object)` einer Instanz der Klasse `java.io.ObjectOutputStream` exportiert und durch `readObject(Object)` der Klasse `java.io.ObjectInputStream` wieder importiert. Neben den erwähnten

Methoden stellen diese Klassen auch Funktionalität zur Behandlung von primitiven Datentypen zur Verfügung. Im Rahmen des entfernten Methodenaufrufs geschieht die Serialisierung und Deserialisierung von Parametern und Rückgabewerten transparent.

Für die Serialisierung eines Objektes wird die Klasse des Objektes herangezogen, ebenso wie deren serialisierbare Basisklassen. Beginnend mit der höchsten (grundlegendsten) Basisklasse werden die Attribute, die von den jeweiligen Klassen deklariert werden, in den Ausgabestrom geschrieben. Zur Wiederherstellung von Objekten aus einem serialisierten Strom muß die erste nicht serialisierbare Basisklasse einen parameterlosen Konstruktor zur Verfügung stellen. Die Attribute, die von den serialisierbaren Klassen deklariert wurden, werden aus dem serialisierten Datenstrom wiederhergestellt.

Der Anwendungsprogrammierer kann den Serialisierungsvorgang für einzelne Klassen beeinflussen, indem er entsprechende `writeObject()` und `readObject()` Methoden implementiert.[4] Dabei kann er sich entweder damit begnügen, zusätzliche Informationen im Datenstrom unterzubringen, oder er kann die Serialisierung vollständig selbst gestalten. Dabei stehen ihm die bereits erwähnten Methoden der Klassen `ObjectOutputStream` und `ObjectInputStream` für primitive Daten zur Verfügung.

Alternativ zur Verwendung der `Serializable` Schnittstelle können Klassen die `java.io.Externalizable` Schnittstelle implementieren. Dabei liegt die Verantwortung für die Serialisierung und Deserialisierung vollständig beim Programmierer. Darüberhinaus kann man eigene Ströme für Objektserialisierung implementieren.

Repräsentierung des Objektzustands Die zur Instantiierung benötigten Klassen werden in Form spezieller Objekte der Klasse `java.io.ObjectStreamClass` repräsentiert. Im Datenstrom werden der volle Name der Klasse gespeichert, ebenso wie Anzahl, Namen und Datentypen der Attribute. Darüberhinaus wird für jede Klasse ein eindeutiger Identifizierer in Form einer acht Byte langen Zahl (Java Datentyp `long`) zur Unterscheidung von anderen Versionen oder zufällig gleichnamigen Klassen in den Datenstrom ausgegeben. Sofern diese Zahl nicht als Attribut der Klasse durch den Programmierer vorgegeben ist, wird sie durch eine Hashfunktion aus dem Namen der Klasse, den Namen der implementierten Schnittstellen, sowie den Methoden und Attributen berechnet.

Der Zustand eines Java Objektes besteht aus primitiven Daten wie Zahlen oder Wahrheitswerten und Referenzen auf andere Objekte. Aggregationen von Objekten gibt es nicht. Attribute, die von der standardmässigen Serialisierung nicht erfaßt werden sollen, werden mit dem Schlüsselwort `transient` versehen.

Das Schlüsselwortes `transient` wirkt sich auch auf die Abgrenzung von Objektgraphen aus. Jedes Objekt wird genau einmal im Datenstrom aufgenommen und mit einem innerhalb des Datenstroms gültigen Identifizierer (engl.: handle) versehen. Diese Art der Referenzierung trifft auch auf Objekte zu, die Klassen repräsentieren. Für eine detaillierte Beschreibung der Grammatik für serialisierte Datenströme siehe [Sun98].

[4] Die Methoden `writeObject()` und `readObject()` müssen dabei als `private` deklariert sein. Dennoch können sie offensichtlich z.B. von `ObjectOutputStream` aus – also von außerhalb der Klasse – aufgerufen werden.

4.2.4 Sicherheitskonzepte

Bei der Entwicklung von Java war man darum bemüht, die Java Laufzeitumgebung vor Angriffen durch Code unbekannter Herkunft zu schützen. Die Sicherheitsmechanismen erstrecken sich über mehrere Ebenen [RP97], nämlich das Design der Sprache sowie statische Überprüfung und dynamische Kontrolle von geladenem Code.

- Zunächst einmal sind in der Sprache Java keine direkten Speicherzugriffe, wie es sie etwa in C oder C++ gibt, vorgesehen. Durch strenge Typisierung wird eine indirekte unerlaubte Manipulation von Speicherinhalten – etwa durch unerlaubte Typumwandlungen – unterbunden. Die Speicherverwaltung wird vollständig von der Laufzeitumgebung übernommen.

- Geladener Code wird zunächst einmal durch einen sogenannten Bytecode Verifier untersucht. Dieser soll garantieren, daß die auf Sprachebene definierten Eigenschaften von Java eingehalten werden. D.h. er überprüft Code auf unerlaubte Speicherzugriffe sowie auf die Einhaltung von Datentypen und Zugriffsrechten für Objektattribute [GM96].
  Die genannten Überprüfungen werden auch vom Java Compiler beim Erzeugen des Bytecode vorgenommen . Gründe für die Notwendigkeit des Bytecode Verifier sind der semantische Unterschied zwischen der Sprache Java und ihrem Bytecode und die Tatsache daß beim Empfang von Bytecode nicht garantiert werden kann, daß der Bytecode auch von einem vertrauenswürdigen Compiler erzeugt wurde. Die semantischen Unterschiede zwischen Java-Quellcode und Bytecode machen die Aufgabe des Bytecode Verifier zudem aufwendig und somit fehleranfällig [MF96]. Es wurden etliche Beispiele bekannt in denen es gelang die Sicherheitsmechanismen von Java aufgrund von Fehlern des Bytecode Verifiers auszuhebeln.

- Überprüfungen, die nicht statisch durchgeführt werden können, wie zum Beispiel die Einhaltung von Indexgrenzen beim Zugriff auf Felder werden zur Laufzeit durchgeführt.

- Klassen werden je nach ihrer Herkunft in verschiedenen Namensräumen verwaltet (siehe Abschnitt 4.2.3.1). Jede benötigte Klasse wird zunächst auf der lokalen Festplatte gesucht und dann im Namensraum der referenzierenden Klasse. Auf diese Weise wird verhindert, daß geladene Klassen sicherheitsrelevante Systemklassen, die sich auf der lokalen Festplatte befinden, ersetzen. Der Zugriff auf sicherheitsempfindliche Ressourcen, wie zum Beispiel auf Dateien oder das Netzwerk sind nur über Systemklassen möglich.

- Der Zugriff auf die sicherheitsempfindlichen Ressourcen wird durch einen sogenannten Security Manager geregelt. Überprüft wird zum Beispiel der Zugriff auf Dateien, auf das Netzwerk, die Installation von ClassLoadern und so weiter. Der Security Manager wird von den Systemklassen, die entsprechende Zugriffe durchführen, konsultiert. Rechte werden pro Aktivitätsträger (Thread) zugeteilt. Java Applikationen können einen Security Manager einsetzen, der ihre spezielle Sicherheitspolitik implementiert. Dies ist jedoch nur einmal während der Abarbeitung

einer Applikation möglich [Sun], [RP97]. Java Applets, die von WWW-Browsern ausgeführt werden können, werden von einem durch den jeweiligen Browser vorgegebenen und sehr restriktiven Security Manager kontrolliert. So können Applets beispielsweise nicht auf das Dateisystem zugreifen und dürfen nur Netzwerkverbindungen zu dem Rechner aufbauen, von dem ihr Code geladen wurde.

4.3 Sumatra (Java basiert)

Literatur: [ARS96], [RASS97]

Bei Sumatra handelt es sich um eine Erweiterung der Programmiersprache Java. Ziel der Entwickler an der University of Maryland, USA war der Entwurf eines Systems, das mit Hilfe von Programmmobilität auf Änderungen in der Verfügbarkeit von Ressourcen wie zum Beispiel Netzwerkbandbreiten reagieren kann. Die Beobachtung von Ressourcen durch Einbindung von Monitoren war somit neben der Mobilität ein weiteres wichtiges Anliegen bei der Realisierung von Sumatra.

4.3.1 Systemarchitektur

Die Funktionalität von Sumatra wird durch eine Erweiterung der Java-Klassenbibliothek und durch eine Modifikation des Java-Interpreters realisiert. Die Schnittstelle der Java Platform ebenso wie die Java Programmiersprache werden durch diese Modifikation nicht beeinträchtigt, so daß die modifizierten Interpreter weiterhin normale Java Programme ausführen können [RASS97]. Andererseits sind Programme, die die erweiterte Funktionalität nutzen, in ihrer Portabilität auf die modifizierten Interpreter eingeschränkt.

Die Ausführungsorte von Sumatra sind modifizierte Java Interpreter, die als sogenannte Ausführungsmaschinen (englisch: Execution-engines) fungieren. Der Zugriff auf die Migrationsfunktionalität der lokalen ebenso wie entfernter Ausführungsmaschinen erfolgt über Instanzen einer Klasse namens `Engine` (siehe Abbildung 4.5). Jede Ausführungsmaschine kann lokal oder entfernt durch Verwendung des Konstruktors von `Engine` unter Angabe einer Rechneradresse und einer TCP-Nummer neue Ausführungsmaschinen erzeugen, für die dann entsprechende neue Interpreter gestartet werden. Das konkrete Starten wird von speziellen Dämon-Prozessen durchgeführt, die auf den entsprechenden Rechnern unter einer bekannten TCP-Nummer erreichbar sind. Die Ausführungsmaschinen ihrerseits sind durch ihre Rechner-Adresse und TCP-Nummer identifiziert.[5]

4.3.2 Handhabung mobiler Objekte

Sumatra bietet als Migrationseinheiten Objektgruppen und Aktivitätsträger an. Diese werden auf höchst unterschiedliche Weise verwendet.

[5] Aus den verfügbaren Quellen geht nicht klar hervor, wie ein Programm Instanzen der Klasse Engine erhält, die nicht die lokale oder vom Programm selbst erzeugte Ausführungsorte repräsentieren.

```
public final class Engine {
   public String hostname;
   public int port;

   public Engine(String hname, int portno);
   public void downloadClass(String Classname);
   public void go();
   public native void rexec(String classname,
                            String[] args);
}
```

Abbildung 4.5: Schnittstelle der Sumatra Ausführungsmaschine

Objektgruppen

Objektgruppen werden durch Instanzen der Klasse `ObjGroup` repräsentiert. Objekte werden einzeln zu einer Gruppe hinzugefügt oder wieder daraus entfernt. Objekte können mehreren Objektgruppen gleichzeitig angehören. Was allerdings geschieht, wenn Objektgruppen, die gemeinsame Objekte enthalten, getrennt werden, ist nicht angegeben.

Thread Objekte, die in Java Schnittstellen für Aktivitätsträger darstellen, können nicht zu Objektgruppen hinzugefügt werden. Andere Objekte, die stark mit dem lokalen Ausführungsort verbunden sind, wie zum Beispiel Ein-/Ausgabe Objekte können nicht migriert werden. Stattdessen werden sie bei der Migration zurückgesetzt. Andere Einschränkungen sind nicht angegeben.

Jede Gruppe ist mit einem Namen assoziiert, der zu ihrer Identifizierung verwendet werden kann. Die Eindeutigkeit der verwendeten Namen wird vom System nur lokal überprüft und erzwungen. Jedes Objekt bleibt nach der Migration für sich adressierbar. Die Umwandlung von lokalen in entfernte Referenzen geschieht für den Benutzer transparent. Objekte, die zu Objektgruppen gehören, sind von der garbage collection ausgenommen.

Die Migration von Objektgruppen wird durch Aufruf der Methode `moveTo(Engine engine)` ausgelöst. Als Parameter erhält die Funktion das Objekt, das den entsprechenden Zielort repräsentiert. Migriert werden alle Objekte, die zur Objektgruppe gehören und nur diese. Ob Objekte die Migration ihrer eigenen Gruppe auslösen können, ist unklar. Eine Migration von Aktivitätsträgern oder gar Ausführungszuständen findet nicht statt.

Aktivitätsträger (Threads)

Aktivitätsträger finden keine spezielle Repräsentation durch Objekte oder Klassen. Zur Migrationseinheit gehören alle Objekte, die Teil des Ausführungszustands sind (siehe Abschnitt 3.4.5). Ausgenommen sind Objektgruppen und darin enthaltene Objekte.

Die Migration von Aktivitätsträgern kann nur aktiv durchgeführt werden. Dazu wird die `go()`-Methode der `Engine`-Instanz aufgerufen, die das Ziel repräsentiert. Migriert wird

der Aktivitätsträger, der die `go()` Methode aufruft. Zurückgelassene Objekte bleiben referenzierbar. Die Ausführung des Aktivitätsträgers wird an der nächsten Anweisung nach dem Migrationskommando fortgesetzt. Der Ausführungszustand wird mitmigriert.

Gemeinsamkeiten

Schlägt der Migrationsvorgang fehl, so wird dies durch Ausnahmen signalisiert und der Programmierer kann entsprechend reagieren. Für Objektgruppen kann der aktuelle Ausführungort lokal und entfernt abgefragt werden. Migrierende Aktivitätsträger können stets lokal ihren aktuellen Ausführungsort ermitteln.

4.3.3 Durchführung der Migration

Bei der Migration von Objektgruppen werden nur Datenzustände übertragen. Bei Aktivitätsträgern die Datenzustände der mitmigrierenden Objekte sowie der Ausführungszustand.

4.3.3.1 Mobiler Code

Für den Transport von Code ist der Anwendungprogrammierer grundsätzlich selbst verantwortlich. Die Klasse `Engine` stellt dafür eine entsprechende Methode zur Übertragung von Java-Klassen zur Verfügung. Zum Einsatz kommt sicherlich Java Bytecode (siehe Abschnitt 4.2.3.1).

4.3.3.2 Datenzustand

Eine Verwendung des Java Serialisierungsmechanismus liegt nicht vor. Stattdessen wurde ein eigener Mechanismus in C entwickelt.

4.3.3.3 Ausführungszustand

Sumatra implementiert parallel neben dem üblichen Stapel, der die Objekte enthält, auch einen zweiten Stapel, der Typinformationen zu diesen Objekten enthält. Übertragen werden auch Programmzählerwerte.

4.3.3.4 Objektidentität und Referenzierung

Objekte, die sich im gleichen Ausführungsort befinden, interagieren über gewöhnliche Java-Referenzen. Werden Objektgruppen migriert, so werden alle lokalen Referenzen auf wegmigrierende Objekte transparent durch Stellvertreterobjekte ersetzt. Ebenso werden Objektreferenzen von migrierenden Aktivitätsträgern auf zurückgelassene Objekte durch Stellvertreterobjekte ersetzt. Was hingegen mit Referenzen in der jeweils umgekehrten Richtung geschieht wird nicht explizit erklärt.

Außer der Umwandlung lokaler Referenzen in Stellvertreterobjekte findet keine weitere Verfolgung von Objekten statt. Aufrufe an Objekte, deren Stellvertreterobjekt nicht mehr die aktuelle Adresse enthalten, werden mit einer Ausnahme beantwortet. Diese enthält die Information über den nächsten bekannten Ausführungsort des entsprechenden Objektes. Es liegt beim Programmierer, diese Information für einen erneuten Aufruf zu nutzen.

4.4 Voyager (Java basiert)

Literatur: [Obj97b], [Obj98b], [Obj98a], [Obj97a], [Obj97c]

Voyager wurde von der Firma Objectspace als rein Java-basiertes verteiltes Objektsystem konzipiert [Obj98b]. Obwohl Voyager zu großen Teilen konform zum CORBA Standard [COR95] für verteilte Systeme ist, entspricht die Handhabung der Objektmobilität nicht der Spezifikation für Objektlebenszyklen (LifeCycleService, siehe [Obj96, Kapitel 6]), in der auch Schnittstellen für das entfernte Kopieren und Migrieren von Objekten vorgesehen sind. Darüberhinaus war die Handhabung der Objektmobilität bei der Entwicklung der hier betrachteten Versionen 1.0.0, 2.0beta und 2.0.0 signifikanten Veränderungen unterworfen. Die hier vorliegende Beschreibung des Systems versucht allen Versionen gerecht zu werden. Merkmale, die sich zwischen den Versionen stark unterscheiden, werden entsprechend versionsabhängig beschrieben.

4.4.1 Systemarchitektur

Die Funktionalität von Voyager steckt zum größten Teil in einer Java-Klassenbibliothek. Java Applikationen und Applets können durch Aufruf einer entsprechenden Funktion eine Voyager Ausführungsumgebung starten. Will man keine eigene Applikation schreiben, die einen Ausführungsort startet, so kann man auf eine vorgefertigte Java-Applikation zurückgreifen, die nichts anderes tut als auch eingehende Methodenaufrufe oder ähnliches zu warten. Pro Applikation und somit pro Java-Interpreter ist der Betrieb eines Ausführungsortes möglich. Jeder Ausführungsort wird durch seine Rechneradresse und eine TCP-Nummer adressiert und identifiziert. Die Auswahl der Rechneradresse[6] und Portnummer kann beim Start von Voyager angegeben werden oder dem System überlassen werden.

Daneben gibt es je nach verwendeter Version Werkzeuge, die für die Bearbeitung von Code vorgesehen sind. In Version 1.0.0 kam ein Werkzeug namens `vcc` zum Einsatz, das für gegebene Java-Klassen in Form von Quell- oder Bytecode den Code für Stellvertreterobjekte generiert [Obj97b]. Ein wesentlicher Teil der Funktionalität sowohl für entfernte Methodenaufrufe als auch für Objektmobilität basiert auf diesen Stellvertreterobjekten. In neueren Versionen ist dieses Werkzeug durch die zur Laufzeit mögliche Untersuchung von Klassendefinitionen in Java (Java Core Reflection) überflüssig geworden.

Andere Werkzeuge befassen sich mit der Erstellung und Umwandlung von Schnittstellendefinitionen. Schnittstellen spielen bei der Handhabung von Objektverteilung und Objektmobilität eine wichtige Rolle.

[6] Dies gilt nur, falls der lokale Rechner unter mehreren Adressen erreichbar ist.

4.4.2 Handhabung mobiler Objekte

4.4.2.1 Migrationseinheiten

Objektgraphen sind die Migrationseinheiten von Voyager. Dabei vereinigt stets ein Objekt, das gleichzeitig die „Wurzel“ des Objektgraphen darstellt, die Steuerung der Migration sowie die entfernte Referenzierbarkeit auf sich. Alle Objekte, die Teil einer Migrationseinheit sind, müssen die `java.io.Serializable`-Schnittstelle (siehe Abschnitt 4.2) implementieren.

4.4.2.2 Mobile und immobile Objekte

Voyager differenziert zwischen etlichen Stufen der Objektmobilität. Es gibt die nicht Java-serialisierbaren immobilen Objekte, Objekte, die Teil einer Migrationseinheit sein können, und die nach dem Sprachgebrauch der Voyager-Dokumentation *mobilen Objekte*, die Ankerpunkt einer Migrationseinheit sein können. Im Verlauf der weiteren Betrachtung von Voyager möge das Wort *mobiles Objekt* im eben beschriebenen Sinne verstanden werden. Bei den *mobilen Objekten* kann man noch einmal unterscheiden zwischen sogenannten *Agenten,* die aktiv migrieren können, und lediglich passiv migrierbaren Objekten.

Die Erzeugung *mobiler Objekte* hat sich über die diversen Versionen von Voyager stark verändert. In Version 1.0.0 müssen Klassen, deren Instanzen mobil sein sollen, zunächst dem oben erwähnten `vcc`-Werkzeug übergeben werden. Dieses erstellt Klassen, die von einer Klasse namens `VObject` abgeleitet sind und deren Instanzen als Stellvertreterobjekte fungieren. Der Name einer generierten Klasse ergibt sich aus dem Namen der bearbeiteten Klasse durch eine Voranstellung des Buchstabens V. Eine Applikation erzeugt die Instanz eines *mobilen Objektes* nicht direkt sondern ein Stellvertreterobjekt, das seinerseits das zugrundeliegende Objekt je nach Wunsch in dem lokalen oder einem entfernten Ausführungsort instantiiert. Normale Java-Objekte können auch nachträglich durch Aufruf der Klassenmethode `VObject.forObject(Object o)` mit einem Stellvertreterobjekt und somit mit Migrationsfähigkeit ausgestattet werden. Agenten müssen zudem noch von einer speziellen Basisklasse erben, um so selbst Zugriff auf die Migrationsfunktionalität zu erhalten. Stellvertreterobjekte für Agenten sind von `VAgent` abgeleitet. (siehe [Obj97b])
Die Version 2.0beta2 verlagert die Generierung der Klassen für Stellvertreterobjekte auf den Zeitpunkt der Objekterzeugung. Objekte werden nicht wie üblich durch Verwendung des `new`-Operators von Java erzeugt sondern durch Aufruf einer Klassenmethode `construct()` der Klasse `Voyager`. Diese erhält als Parameter den Namen einer Klasse, sowie optional Parameter für deren Konstruktor. Die nachträgliche Erzeugung der Migrationsfähigkeit erfolgt durch Aufruf der Klassenmethode `Lifecycle.of(Object o)`. Auch hier müssen Agenten von der Basisklasse `Agent` erben [Obj98a].
Version 2.0.0 bietet die eben beschriebene Vorgehensweise in leicht modifizierter Form ebenfalls an. Die entsprechende Methode heißt jetzt `create()` und gehört zur Klasse `Factory` [Obj98b]. Zusätzlich ist es jetzt möglich, bereits erzeugte gewöhnliche Java-Objekte nachträglich zu *Agenten* zu machen. Das Konzept, das dies ermöglicht, heißt dy-

namische Aggregation [Obj98b]. Sie erweitert die Funktionalität eines bestehenden Objektes durch die Assoziierung mit sogenannten Facetten.

Der Zugriff auf *mobile Objekte* findet in der Regel über Stellvertreterobjekte statt, auch wenn dies umgangen werden kann. Der Zugriff auf die Stellvertreterobjekte wiederum findet stets über Schnittstellen statt. Das in [Obj97b] vorgeschlagene Vorgehen ist wie folgt: Der Programmierer entwickelt zunächst eine Klasse, die anwendungsspezifische Aufgaben erledigt. Anschließend definiert er eine Schnittstelle (Java-Interface), über die ein Zugriff auf die Methoden seiner Klasse möglich ist. Dies kann auch automatisch durch ein Voyager-Werkzeug erledigt werden. Der Name der Schnittstelle ergibt sich aus dem Namen der Klasse durch Voranstellung des Buchstaben I. Diese Schnittstellen werden verwendet, um Variablen, die mobile Objekte referenzieren sollen, zu deklarieren. Auf diese Weise bietet das Stellvertreterobjekt die gleiche Schnittstelle an, wie die zugrundeliegende Klasse. Daneben implementieren Stellvertreterobjekte auch weitere Schnittstellen, insbesondere auch eine Schnittstelle namens `IMobility`, die den Zugriff auf die Migrationsfunktionalität regelt.

Die Steuerung der Lebensdauer von Voyager-Objekten war über die Versionen hinweg größeren Veränderungen unterworfen. In der Version 1.0.0 konnte zwischen der expliziten Angabe beliebiger Zeiten für die Lebensdauer und einer herkömmlichen Steuerung der Lebensdauer durch Referenzierung gewählt werden. Außerdem existierte eine explizite Methode zur Zerstörung eines Objektes [Obj97b]. In der Version 2.0.0 ist hingegen nur noch eine Methode für Agenten vorgesehen, mit der die Erfassung des Objektes durch die automatische referenzenbasierte Speicherbereinigung (garbage collection) an- und abgeschaltet werden kann [Obj98b].

4.4.2.3 Migrationsanweisungen

Passive Migration wird durch Aufruf der Methode `moveTo()` des Stellvertreters des entsprechenden *mobilen Objektes* ausgelöst. Der Zielort kann entweder durch Angabe der Adresse eines Ausführungsortes oder aber durch die Referenz auf ein anderes mobiles Objekt festgelegt werden. Die Migration wird erst durchgeführt wird, wenn alle im Objekt aktiven synchronen[7] Methoden abgearbeitet sind. Neu eingehende Aufrufe werden in der Zwischenzeit in eine Warteschlange eingereiht. Die Dokumentation [Obj97b], [Obj98b] weist darauf hin, daß die Migration von Objekten, auf die neben Voyager-Stellvertreterobjekten auch normale Java-Referenzen verweisen, nicht sicher ist. Ebenso dürfen nur Methodenaufrufe im Objekt aktiv sein, die über Voyager-Stellvertreterobjekte getätigt wurden.

Agenten können in Version 1.0.0 und 1.0beta2 ihre eigene `moveTo()` Methode aufrufen. In Version 2.0.0 können Objekte sich selbst mit einer Agenten-Facette assoziieren, und anschließend deren `moveTo()`-Methode aufrufen. Dabei kann als optionaler Parameter der Name einer Methode angegeben werden, die nach der Migration ausgeführt werden soll[8]. Anweisungen, die hinter dem Aufruf der `moveTo()`-Methode stehen, werden im

[7] Siehe z.B. [Fla96] für eine Erläuterung des Schlüsselwortes `synchronized` in Java.

[8] Die Angabe eines Funktionsnamens scheint nach [Obj98b] in Version 2.0.0 nicht mehr optional sondern zwingend zu sein.

Falle erfolgreicher Migration nicht mehr ausgeführt, da der aktuelle Aktivitätsträger beendet wird. Hier lohnt sich lediglich die Plazierung von Code für die Ausnahmebehandlung im Falle eines abgebrochenen Migrationsvorgangs.

Auch die Benachrichtigung von Objekten über Migrationsvorgänge hat sich über die Versionen hinweg verändert. Die Versionen 1.0.0 und 2.0beta2 generieren Ereignisse, um Objekten bevorstehende Migrationen die Ankunft in einem neuen Ausführungsort zu signalisieren (siehe [Obj97b], [Obj98a]). Version 2.0.0 verwendet hingegen Rückruf-Funktionen (englisch: Callback), um diese Vorgänge zu signalisieren. Die angezeigten Phasen der Migration sind dabei (siehe [Obj98b]):

- Vor der Migration: Dies wird dem mobilen Objekt am Ausgangsort angezeigt. Der Fortgang des Migrationsvorgangs kann durch Erzeugung einer Ausnahme gestoppt werden.
- Vor der Ankunft: Dies wird der Kopie des mobilen Objekt am Zielort angezeigt. Zu diesem Zeitpunkt ist noch die Instanz am Ausgangsort das gültige Objekt. Auch hier kann die Migration noch abgebrochen werden.
- Nach der Ankunft: Dies wird ebenfalls der Kopie des mobilen Objekt am Zielort angezeigt. Zu diesem Zeitpunkt ist die Instanz am Zielort das gültige Objekt. Der Migrationsvorgang ist abgeschlossen und kann nicht mehr abgebrochen werden.
- Nach der Migration: Dies wird dem nunmehr veralteten Objekt am Ausgangsort angezeigt und soll genutzt werden, um letzte Aufräumarbeiten zu erledigen.

4.4.3 Durchführung der Migration

4.4.3.1 Mobiler Code

Klassen werden zum Zeitpunkt der Migration von einem von Voyager installierten Java-ClassLoader (siehe Abschnitt 4.2.3.1) geladen [Obj97b, Seite 166]. Ausführungsorte können optional als *http*-Server fungieren und benötigte Klassen zur Verfügung stellen [Obj98b]. Dabei können sie auch als indirekter Klassenlader für Ausführungsorte fungieren, die als Java-Applets in WWW-Browser eingebettet sind. Java-Applets unterliegen nämlich aus Gründen der Ausführungssicherheit der Beschränkung, nur Netzwerkverbindungen zu dem Rechner aufzubauen von dem der ausführende WWW-Browser ihren Code geladen hat. Somit ist auch das Laden von Klassen anderer Herkunft nicht ohne Indirektion möglich.

4.4.3.2 Datenzustand

Für die Übertragung von Datenzuständen wird die Java-Serialisierung (siehe Abschnitt 4.2.3.2) verwendet. Da es in Java keine Möglichkeit gibt, Objekte explizit zu zerstören, bleibt der Objektgraph trotz Migration ganz oder teilweise im ursprünglichen Ausführungsort bestehen, wenn es außer der Referenzierung durch Stellvertreterobjekte noch gewöhnliche Java-Referenzen auf Objekte aus der Migrationseinheit gibt.

4.4.3.3 Objektidentität und Referenzierung

Wie bereits erwähnt, verwendet Voyager Stellvertreterobjekte für die Referenzierung *mobiler Objekte*. Diese sind ebenso wie die *mobilen Objekte* selbst eigenständige Objekte im Sinne der Java Programmiersprache. Der Zugriff auf die mobilen Objekte kann auch über normale Java Referenzen erfolgen und so Methodenaufrufe erheblich beschleunigen. Die Migrationssemantik bezüglich der Objektidentität wird dadurch jedoch erheblich beeinträchtigt, weil Objektgraphen so ganz oder teilweise kopiert werden, anstatt wirklich zu migrieren (siehe vorigen Abschnitt).

Voyager 1.0.0 generiert bei der Erzeugung eines *mobilen Objektes* eine 16 Byte lange Zufallszahl und gibt dem Benutzer auf Wunsch auch Zugang dazu [Obj97b, Seite 55]. Diese Zahl kann dazu verwendet werden Stellvertreterobjekte für das entsprechende Objekt zu erzeugen. Intern wird diese Zahl verwendet, um beispielsweise zu Überprüfen, ob zwei Stellvertreterobjekte dasselbe Objekt referenzieren.

Alternativ wird die Identifizierung von Objekten durch Namensdienste empfohlen. Jeder Ausführungsort unterhält einen Namensdienst, bei dem Objekte angemeldet und mit einem Namen versehen werden können. Um so eine Objektreferenz zu erhalten muß man allerdings erst wissen wo es erzeugt wurde.

Die Lokalisierung von Migrationseinheiten und die Zustellung von Nachrichten erfolgt mit Hilfe von Weiterleitungsobjekten, die *mobile Objekte* an allen Ausführungsorten zurücklassen. Der Aufrufer erhält mit dem Ergebnis auch die neue Adresse des Objektes. Dies geschieht transparent für den Anwender. Weiterleitungsobjekte haben die gleiche Lebensdauer wie das referenzierte Objekt. Das Hinterlassen von Weiterleitungsobjekten kann pro *mobilem Objekt* abgeschaltet werden [Obj97b, Seite 173].

Die Referenzierung wird zur Bestimmung der Lebensdauer von Objekten verwendet, sofern nicht abgeschaltet ist. In Version 1.0.0 senden Stellvertreterobjekte alle 60 Sekunden sogenannte heartbeat-Nachrichten, um das Objekt am Leben zu halten. In Version 2.0.0 schickt jeder Ausführungsort alle zwei Minuten Nachrichten über neu erstellte und zerstörte Stellvertreterobjekte an die Ausführungsorte, die referenzierte Objekte enthalten.

4.4.4 Sicherheitskonzepte

Voyager erbt die Sicherheitsmechanismen von Java (siehe Abschnitt 4.2.4). Ausführungsumgebungen, die als Java-Applets gestartet werden, müssen sich den restriktiven Beschränkungen des Security Managers für Applets unterwerfen. Java-Applikationen können wahlweise ohne Security Manager arbeiten, einen eigenen SecurityManager installieren oder aber den `VoyagerSecurityManager` verwenden. Dieser unterscheidet bei Objekten, ob deren Klassen von der lokalen Festplatte oder über das Netz geladen wurden. Die ersteren Objekte werden gar nicht eingeschränkt, während die Beschränkungen für Objekte mit „fremden“ Klassen ähnlich sind wie Applets. Die Unterschiede liegen darin, daß der `VoyagerSecurityManager` mehr Netzwerkfunktionalität erlaubt aber die Manipulation von Aktivitätsträgern verbietet. Außerdem werden die Aufrufe bestimmter Methoden der Voyager-Systemklassen kontrolliert [Obj98b].

4.5 IBM Aglets (Java basiert)

Literatur: [OKO98], [LA97], [LC96], [Lan97], [IBM]

Das Aglets-System, auch Aglets-Workbench genannt [LC96], wird von IBM-Forschungszentren in Japan und der Schweiz entwickelt. Es ist als Agentensystem konzipiert und basiert vollständig auf Java. Die Namensgebung erinnert nicht zufällig an die Java-Applets, will doch IBM nach der von den Applets etablierten Codemobilität nun mit Aglets die Objektmobilität etablieren [Lan97]. IBM versucht einerseits seine Konzepte für Agententransfer als Standard der Object Management Group zu etablieren und andererseits mit den mitgelieferten visuellen Werkzeugen zur Verwaltung und Steuerung von Aglets eine benutzerfreundliche Bedienungsschnittstelle vergleichbar den WWW-Browsern mit Java-Unterstützung anzubieten. Die folgende Beschreibung bezieht sich auf Version 1.1 der Aglets-Workbench.

4.5.1 Systemarchitektur

Die Funktionalität von Aglets wird in Form von Klassenbibliotheken zur Verfügung gestellt. Dabei wird zwischen zwei aufeinander aufbauenden Schichten unterschieden. Die untere Schicht ist zuständig für die Implementierung von Kommunikationsprotokollen während die darauf aufbauende Schicht, die Verwaltung und Steuerung von Aglets realisiert [OKO98].

Die Ausführungsumgebungen von Aglets sind Java Applikationen. Der Benutzer kann einen Ausführungsort in seine eigene Applikation einbetten, indem er die entsprechende Laufzeitumgebung durch Aufruf der Methode `AgletRuntime.init()` initialisiert. Alternativ kann er eine mit der Aglets-Workbench mitgelieferte Java-Applikation namens `tahiti` starten, die eine grafische Benutzeroberfläche zur Verwaltung und Steuerung von Aglets bietet.

Die Ausführungsorte von Aglets heißen AgletContext. Dabei können mehrere Kontexte in einer Java-Applikation aktiv sein [OKO98]. Die Adresse eines Kontextes besteht aus einer Rechneradresse, einer TCP-Nummer, die die Ausführungsumgebung identifiziert und einem Namen, der den Kontext innerhalb der Ausführungsumgebung identifiziert.

4.5.2 Handhabung mobiler Objekte

4.5.2.1 Migrationseinheiten

Migrationseinheiten, hier Aglets genannt, sind Objektgraphen, deren Anker ein von der Klasse Aglets abgeleitetes Objekt sein muß. Dieses Ankerobjekt konzentriert die Steuerung der Migration sowie die entfernte Referenzierbarkeit auf sich. Alle Objekte, die Teil einer Migrationseinheit sind, müssen die Schnittstelle `java.io.Serializable` oder `java.io.Externalizable` (siehe Abschnitt 4.2) implementieren. Die Klasse `Aglets` vererbt die Implementation von `java.io.Serializable` automatisch.

4.5.2.2 Mobile und immobile Objekte

Wie schon bei Voyager ergibt sich eine dreistufige Differenzierung der Objektmobilität (siehe 4.4). Man unterscheidet zwischen Aglets, den Ankerobjekten, an denen insbesondere die Identität der Migrationseinheiten festgemacht wird, serialisierbaren Objekten, die Teil von Migrationseinheiten sein können und immobilen Objekten.

Aglets werden durch Anforderung an einen lokalen oder entfernten Aglet-Kontext erzeugt. Als Parameter werden der Name der zu instantiierende Klasse angegeben sowie eine URL, die angibt wo die entsprechende Klasse gefunden werden kann. Weiterer Parameter ist ein Objekt , das an die Initialisierungsfunktion des Aglets übergeben wird. Bei entfernter Erzeugung muß dem Laufzeitsystem auch die Adresse eines Kontextes angegeben werden. Alternativ kann auch ein vorhandenes Aglet dupliziert werden. Da nicht der gewöhnliche Java-Mechanismus zur Objekterzeugung zum Einsatz kommt, wird in [OKO98] von der Implementation eines Java-Konstruktors abgeraten. Stattdessen soll eine Methode Namens `onCreation(Object init)` implementiert werden, der der oben erwähnte Parameter übergeben wird.

Neben der eben genannten Methode gibt es noch drei weitere, deren Implementation für die Realisierung des gewünschten Verhaltens wichtig sind. Eine `run()` Methode, die nach der Objektinitialisierung und bei jeder Aktivierung, also auch nach Migrationen, aufgerufen wird. Weiterhin wichtig ist eine Methode namens `handleMessage(Message msg)`. Die Zustellung von Nachrichten auf diesem Wege ersetzt die klassischen Methodenaufrufe. Der Empfänger untersucht die eingehende Nachricht und trifft entsprechende Fallunterscheidungen. Diese Vorgehensweise widerspricht allerdings dem Prinzip der Objektorientierung, Implementationen zu verbergen und stattdessen Schnittstellen bekanntzumachen. Letztlich gibt es noch die Methode `onDisposing()` die unmittelbar vor der Zerstörung eines Objektes aktiviert wird. Aglets sind von der normalen Speicherbereinigung ausgenommen und müssen explizit zerstört werden.

Der Zugriff auf Aglets erfolgt normalerweise über Stellvertreterobjekte, die bei der Erzeugung eines Aglets zurückgegeben werden und die eine Schnittstelle namens `AgletProxy` implementieren. Stellvertreterobjekte für bereits existierende Aglets lassen sich nachträglich erwerben. Dazu kann man sich vom lokalen oder einem entfernten Kontext eine Liste von Stellvertreterobjekten für alle im jeweiligen Kontext residierenden Aglets geben lassen. Auch eine individuelle Identifizierung von Aglets ist möglich, sofern die sogenannte `AgletID` (siehe unten) eines Aglets verfügbar sowie sein aktueller Kontext bekannt ist.

4.5.2.3 Migrationsanweisungen

Aglets unterscheidet bei der Migration zwischen Verschickung und Rückholung. Das Kommando für das Verschicken Aglets heißt `dispatch()` . Dabei handelt es sich um eine Methode der Klasse `Aglets`. Sie kann sowohl für aktive wie auch für passive Migration verwendet werden, da sie dem Aglet selbst zur Verfügung steht und auch von außen über ein entsprechendes Stellvertreterobjekt aufgerufen werden kann. Als Parameter erhält die Funktion dies Adresse des Zielkontextes in Form einer URL. Alternativ

können auch sogenannte Tickets verwendet werden. Sie spezifizieren neben dem Zielort auch Details über das zu verwendende Übertragungsprotokoll.

Die Rückholung eines Aglets wird durch Aufruf der Methode `AgletContext.retractAglet()` bewirkt [IBM]. Als Parameter wird wird die Adresse des aktuellen Ausführungsortes, sowie die entsprechende `AgletID` angegeben. Fraglich ist, ob es sich hier um ein redundantes Kommando handelt, oder ob die oben genannten `dispatch()` Methoden nur lokal ausgeführt werden.

Vor der Ausführung einer Migration wird dem Aglet ein Ereignis zugestellt, ebenso wie unmittelbar nach seiner Ankunft in einem neuen Kontext. Bei den Ereignissen, die eine bevorstehende Migration anzeigen wird unterschieden zwischen einem Wegschicken und dem Zurückholen eines Aglets.

Für den Empfang von solchen Ereignissen muß allerdings ein entsprechender Empfänger registriert werden. Solch ein Ereignisempfänger ist ein Java-Objekt, das eine bestimmte Schnittstelle implementieren muß. Als vom Ankerobjekt direkt oder indirekt referenziertes Objekt ist es Teil des Aglets.

Nach der Ankunft in einem neuen Kontext und der Zustellung und Verarbeitung der entsprechenden Ereignisse wird die `run()` Methode eines Aglets ausgeführt.

4.5.3 Durchführung der Migration

Das Aglets System ist so konzipiert, daß es Kommunikationsmechanismen verschiedener Standards als Grundlage verwenden kann [OKO98]. Vorgesehen ist die Verwendung der Protokolle von JavaRMI, CORBA und das von IBM neu entwickelte Agent Transfer Protocol ATP (siehe [LA97]).

Das ATP Protokoll kommt normalerweise zum Einsatz. Es ist möglichst allgemein gehalten um auch von anderen mobilen Agentensystemen verwendet werden zu können. Es verwendet ein URL-Schema für die Identifizierung von Ausführungsorten, Agenten und Klassen. Für die Interaktion von Ausführungsorten sind vier Vorgänge vorgesehen, nämlich das Verschicken und Zurückholen von Agenten, die Anforderung von Code und die Zustellung von Nachrichten. Die konkrete Codierung der Inhalte, also der Agenten und Klassen ist nicht Gegenstand des Protokolls, sondern nur die Signalisierung der verwendeten Codierungen und Formate.

Aglets überträgt Datenzustand und Code von Migrationseinheiten größtenteils zusammen. Werden Objekte im Rahmen der Zustellung von Nachrichten übertragen so findet kein Codetransfer statt.

4.5.3.1 Mobiler Code

Jedem Aglet ist eine eigener `AgletClassLoader` zugeordnet [OKO98] (siehe Abschnitt 4.2.3.1 zum Thema Java-Klassenlader). Wie im Abschnitt 4.5.2 bereits angegeben, muß bei der Erzeugung eines Aglets eine Quelle angegeben werden, von der der entsprechende Code geladen werden kann. Dies kann beispielsweise die URL einer von

einem http-Server zur Verfügung gestellten Datei sein oder eine ATP-URL für Klassen, die bis auf die Spezifizierung des verwendeten Protokolls genauso aufgebaut ist wie eine http-URL.

Aglets teilt Klassen in vier Kategorien ein:

1. Archivierte Klassen: Dies sind Klassen, die in einem Java Klassen Archiv (ähnlich einem durch den Komprimierer ZIP erzeugten Archiv) enthalten sind. Ist ein Klassenarchiv als Codequelle für ein Aglet angegeben, so wird stets das ganz Archiv bei der Migration eines Aglets mitübertragen.

2. Codebasis Klassen: Dies sind Klassen, die ebenfalls durch eine URL ladbar sind, jedoch nicht in Archiven zusammengefaßt sind. Die Menge der zu übertragenden Klassen wird bei der Serialisierung des Aglet-Objektgraphen ermittelt, und mit dem Aglet während der Migration übertragen. Eine Analyse des Codes findet nicht statt. Dadurch kann es vorkommen, daß ein Aglet später Klassen instantiiert, die während der Migration nicht übertragen wurden. Diese können, sofern noch eine Verbindung zur Codequelle besteht, nachgeladen werden.

3. System Klassen: Dies sind Klassen, die von der lokalen Festplatte der jeweiligen Ausführungsumgebung geladen werden können, wie beispielsweise Java-Systemklassen. Solche Klassen werden nicht übertragen. Daher sollten alle Ausführungsumgebungen über die gleiche Menge an Systemklassen verfügen.

4. Sonstige Klassen: Dies sind Klassen, die weder im lokalen System noch in der Codebasis (Codequelle) eines Aglets auffindbar sind. Jedes Aglet hat ist für seine gesamte Lebensdauer mit genau einer Codequelle assoziiert. Sollte ein Aglet etwa im Rahmen einer Nachrichtenzustellung ein Objekt erhalten, das Instanz einer Klasse ist, die von einer anderen als der eigenen Codequelle stammt, so kann es dies Objekt nicht verwenden. Dieser Fall wird dem Aglet durch eine Sicherheitsausnahme signalisiert.

4.5.3.2 Datenzustand

Für die Übertragung von Datenzuständen wird der Java-Serialisierungmechanismus verwendet. Enthält ein Aglet Stellvertreterobjekte für den Zugriff auf andere Aglets, so behalten diese auch am Migrationsziel ihre Gültigkeit. Die referenzierten Aglets werden nicht mitmigriert.

4.5.3.3 Objektidentität und Referenzierung

Wie in Abschnitt 4.5.2 bereits erwähnt, findet der Zugriff auf Aglets stets über Stellvertreterobjekte statt. Dies kann auch umgangen werden, da das System auf Wunsch auch normale Java-Referenzen auf lokal vorhandene Aglets-Ankerobjekte zurückgibt. Existieren solche Java-Referenzen zum Zeitpunkt der Aglet-Migration, so kommt es zu einer ganzen oder teilweisen Duplikation der Migrationseinheit [OKO98].

Das Aglets Agentensystem kann nach in der Version 1.0 Migrationseinheiten nur über einen Migrationsschritt verfolgen. Nach [AO98] sind in neueren Versionen alle in Abschnitt 3.4.6.2, Seite 3.4.6.2 beschriebenen Strategien nämlich Weiterleitung, Registrierung und Suche möglich, wobei der Benutzer für jedes Stellvertreterobjekt die Strategie festlegen kann. Die Suche nach Objekten muß er zum Teil sogar selber implementieren. Die Dokumentation der aktuell vertriebenen Version 1.1b [IBM] der Aglets-Workbench kann die Aussagen aus [AO98] jedoch noch nicht bestätigen.

Die Identität eines Aglets wird intern durch alphanumerische Zeichenkette repräsentiert. Der Anwender kann dieses unveränderliche Attribut eines Aglets durch dessen Methode `getAgletID()` erfragen und erhält es in Form eines Objektes der Klasse `AgletID`. Während Version 1.0 von Aglets noch Konstruktoren für `AgletID`s vorsah, ist nun keine Erzeugung solcher Werte durch den Anwender mehr vorgesehen. `AgletID`s können zusammen mit der Kenntnis über den Aufenthaltsortes eines Objektes dazu verwendet werden die Referenzierung eines Aglets nach einer Unterbrechung wiederherzustellen.

4.5.4 Sicherheitskonzepte

Aglets verwendet das Java SecurityManager-Konzept zur Realisierung eigener Sicherheitsmechanismen. Die Rechte für den Zugriff auf Dateisystem, Netzwerk, grafische Benutzerschnittstelle, Drucksystem und weitere sicherheitskritische Ressourcen werden geregelt. Ebenso wird die Manipulation von Ausführungsorten, die Steuerung anderer Aglets und die entfernte Kommunikation kontrolliert. Die Zuteilung von Rechten basiert dabei auf der Herkunft der Klassen von Migrationseinheiten.

Jede Ausführungsumgebung gehört einem Besitzer, der sich beim Start mit Benutzername und Passwort anmeldet. Die Klassen und Migrationseinheiten, die von einer Ausführungsumgebung exportiert werden, gehören automatisch demselben Besitzer. Die Identität des Besitzers wird bei der Zuteilung von Rechten mit berücksichtigt.

Weitergehende Konzepte, wie Verschlüsselungsverfahren und Signierung von Klassen sind geplant aber noch nicht implementiert [OKO98].

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung

Die vorliegende Arbeit beschreibt und kommentiert für Objektmobilität verwendete Konzepte. Dabei beschränkt sich die Auswahl dieser Konzepte auf solche, die einen unmittelbaren Bezug zu der Handhabung und Durchführung von Objektmigration haben.

Da es auf diesem Gebiet noch wenig umfassende Standardliteratur gibt, ergeben sich die in Kapitel 3 herausgearbeiteten Themen und Alternativen vor allem aus der Betrachtung von bestehenden Systemen, die Objektmobilität implementieren. Die so zusammengetragenen Aspekte von Objektmobilität werden schließlich dazu verwendet, die implementierten Systeme in möglichst einheitlicher und somit vergleichbarer Form zu untersuchen.

Kernthemen dieser Arbeit sind besonders die für Anwendungsentwickler sichtbaren Konstrukte zur Handhabung von Migration (Abschnitt 3.3), sowie die Strategien für die systeminterne Realisierung von Migrationen (Abschnitt 3.4). Außerdem wird die Gestaltung der Ausführungs- und Entwicklungsumgebungen mobiler Objektsysteme besprochen (Abschnitt 3.2) und ein Überblick über die Sicherheitsproblematik im Zusammenhang mit Objekt- und besonders Codemobilität gegeben.

Eine angemessene Eingrenzung der zu besprechenden Themen erwies sich als schwierig. Etliche der betrachteten Systeme sind mit Hinblick auf bestimmte Anwendungsgebiete entwickelt worden, die meisten als mobile Agentensysteme. Dadurch ergeben sich dort Schwerpunkte, die nur mittelbar mit Objektmobilität zu tun haben. Etliche Arbeiten befassen sich beispielsweise mit Kommunikationsmechanismen, die der Dynamik von mobilen Einheiten, die sich a priori gegenseitig nicht bekannt sind, besonders gerecht werden. Diese Arbeit befaßt sich mit Kommunikationsmechanismen jedoch nur soweit es den Migrationsvorgang selbst betrifft und mit der Realisierung von Methodenaufrufen im Rahmen der Referenzierung von Migrationseinheiten.

Die Untersuchung der mobilen Objektsysteme zeigt, daß Objektorientierung und Mobilität wie erwartet miteinander verträglich sind. Insbesondere stellt auch die Vererbung kein grundsätzliches Problem dar, wie es etwa bei den Vererbungsanomalien im Zusammenhang mit Nebenläufigkeit oft der Fall ist.

Einzelne Objekte werden jedoch selten als Migrationseinheiten verwendet. Größere Migrationseinheiten wie Objektgraphen werden oft bevorzugt. Dies hat vor allem zwei Gründe.

Zum einen enthalten einzelne programmiersprachliche Objekte in der Regel wenig Funktionalität. Der Aufwand für die Migration steht dann in einem schlechten Verhältnis zum Umfang der Arbeit, die das Objekt zwischen Migrationsschritten erledigen kann. Die Funktionalität von einzelnen Objekten aufzublähen, widerspräche jedoch den Zielen der Objektorientierten Programmierung.

Zum anderen sind Referenzierung und somit auch Kommunikationsmechanismen über die Grenzen von Migrationseinheiten hinweg anders implementiert als innerhalb von Einheiten. Oft sind Methodenaufrufe zwischen verschiedenen Migrationseinheiten um Größenordnungen langsamer als herkömmliche Aufrufe, selbst wenn sie nur innerhalb eines Ausführungsortes durchgeführt werden. Daß dieser große Unterschied nicht zwangsläufig hingenommen werden muß, beweist das Emerald System.

5.2 Ausblick

Mechanismen für die Migration von Objekten werden schon seit Ende der 80er Jahre untersucht und realisiert. Große Beachtung fand das Thema jedoch erst nach Erscheinen der Programmiersprache Java mit ihrem maschinenunabhängigen mobilen Bytecode. Die Zahl der mobilen Objektsysteme, die Java als Grundlage verwenden, spricht für sich.

Obwohl die Technologie für Objektmobilität also mittlerweile vorhanden ist, gibt es noch kaum Anwendungen, die ihre Vorteile nutzen. Der Einsatz der Mobilität als neues Programmierparadigma und der Vergleich zu herkömmlichen verteilten Systemen sind Gegenstand der aktuellen Forschung. Anwendungen, insbesondere im Bereich von Telekommunikation, Netzwerkmanagement und für mobile Rechner werden realisiert und untersucht.

Für die bislang geringe Verwendung mobiler Objekttechnologie insbesondere im kommerziellen Bereich lassen sich einige Gründe anführen. Insbesondere für den Einsatz in offenen Netzen wie dem Internet, scheinen die Sicherheitskonzepte noch nicht ausgereift genug zu sein.

Die mobilen Objektsysteme sind zudem sowohl auf der technischen Ebene als auch auf der Ebene der Handhabung untereinander inkompatibel . Bislang hat sich noch keines als Standard durchsetzen können. Im Bereich der mobilen Agenten findet inszwischen eine Standardisierung durch die Object Management Group (OMG) statt, die die Kooperation verschiedener Agentensysteme regeln soll [MBB+98]. Dabei werden jedoch lediglich Schnittstellen und Abläufe für den Austausch und die Verwaltung von mobilen Agenten, sowie einige Begriffe über die Architekturen mobiler Agentensysteme festgelegt. Eine Festlegung auf Programmiersprachen, Codeformate oder Formate für die Übertragung von Objektzuständen ist jedoch nicht Gegenstand der Standardisierung, um nicht mit vorhandenen Systemen zu konkurrieren. Dies stellt jedoch, anders als bei immobilen Objekten, einen recht kleinen gemeinsamen Nenner dar. Bei immobilen Objekten hingegen

reicht die Standardisierung von Schnittstellen, entfernten Objektreferenzen und entfernten Methodenaufrufen für die Kooperation heterogener Systeme aus.

Neben der Problematik der konkreten Implementierung und Nutzung von mobilen Objektsystemen ist auch die formale Erfassung und Beschreibung von Objektmobilität Gegenstand der Forschung. Hier geht es darum, Programme formal zu spezifizieren oder ihre Eigenschaften zu verifizieren. Während dies beispielsweise für funktionale Sprachen recht weit fortgeschritten ist, steht die Objektmobilität hier erst am Anfang.

Literaturverzeichnis

[ACM86] ACM: *First ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, SIGPLAN Notices, New York, September 1986.

[AO98] ARIDOR, YANIV und OSHIMA, M.: *Infrastructure for Mobile Agents: Requirements and Design*. In: ROTHERMEL, KURT und FRITZ HOHL [RH98], Seiten 38–49.

[ARS96] ACHARYA, ANARUG, RANGANATHAN, M. und SALTZ, J.: *Sumatra: A Language for Resource-Aware Mobile Programs*. In: VITEK, JAN und CHRISTIAN TSCHUDIN [VT96], Seiten 111–130.

[AS96] ACHARYA, ANURAG und SALTZ, J.: *Dynamic Linking for Mobile Programs*. In: VITEK, JAN und CHRISTIAN TSCHUDIN [VT96], Seiten 245–262.

[Ben87] BENNETT, JOHN K.: *The Design and Implementation of Distributed Smalltalk*. In: MEYROWITZ, NORMAN [Mey87], Seiten 318–330.

[BHJL86] BLACK, ANDREW, HUTCHINSON, N., JUL, E. und LEVY, H.: *Object Structure in the Emerald System*. In: *First ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)* [ACM86], Seiten 78–85.

[BNOW95] BIRRELL, ANDREW, NELSON, G., OWICKI, S. und WOBBER, E.: *Network Objects*. Software–Practice and Experience, 25(S4):87–130, Dezember 1995. Auch verfügbar als Forschungsbericht 115 des Digital Systems Research Center.

[Boo94] BOOCH, GRADY: *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 2. Auflage, 1994.

[BR98] BAUMANN, JOACHIM und ROTHERMEL, K.: *The Shadow Approach: An Orphan Detection Protocol for Mobile Agents*. In: ROTHERMEL, KURT und FRITZ HOHL [RH98], Seiten 2–13.

[Car95] CARDELLI, LUCA: *Obliq, A Language with Distributed Scope.* Computing Systems, 8:27–59, 1995.

[CGP96] CUGOLA, GIANPAOLO, GHEZZI, C. und PICCO, G. P.: *Analyzing Mobile Code Languages*. In: VITEK, JAN und CHRISTIAN TSCHUDIN [VT96], Seiten 93–110.

[COR95] OBJECT MANAGEMENT GROUP: *CORBA: Architecture and Specification*, August 1995.

[DK87] DIGEL, WERNER und KWIATKOWSKI, G. (Herausgeber): *Meyers grosses Taschenlexikon in 24 Bänden*, Band 10. BI-Taschenbuchverlag, 1987.

[Eck97] ECKEL, BRUCE: *Thinking in Java*. August 1997. http://www.EckelObjects.com.

[Fla96] FLANAGAN, DAVID: *Java in a Nutshell*. O'Reilly & Associates, Inc., 1. Auflage, Februar 1996.

[Fün98] FÜNFROCKEN, STEFAN: *Transparent Migration of Java-Based Mobile Agents : Capturing and Reestablishing the State of Java Programs*. In: ROTHERMEL, KURT und FRITZ HOHL [RH98], Seiten 26–37.

[Fra96] FRANZ, MICHAEL: *Adaptive Compression of Syntax Trees and Iterative Dynamic Code Optimization: Two Basic Technologies for Mobile Object Systems*. In: VITEK, JAN und CHRISTIAN TSCHUDIN [VT96], Seiten 263–276.

[Gei96] GEIER, MARTIN: *Untersuchung von Koordinierungsmechanismen in objektorientierten, nebenläufigen Programmiersystemen*. Diplomarbeit, FAU Erlangen-Nürnberg, IMMD IV, April 1996.

[Gena] GENERAL MAGIC: *Introduction to the Odyssey API*. http://www.genmacig.com/agents.

[Genb] GENERAL MAGIC: *Mobile Agents White Paper*. http://www.genmacig.com/agents.

[GM96] GOSLING, JAMES und MCGILTON, H.: *The Java Language Environment, A White Paper*. Sun Microsystems, Inc., Mai 1996.

[GR85] GOLDBERG, ADELE und ROBSON, D.: *Smalltalk 80: the language and its implementation*. Computer Science. Addison Wesley, 1985.

[Gri98a] GRIFFEL, FRANK (Herausgeber): *Componentware: Konzepte und Techniken eines Softwareparadigmas*. dpunkt-Verlag, 1998.

[Gri98b] GRISWOLD, DAVID: *The Java HotSpot Virtual Machine Architecture*. Sun Microsystems, Inc., März 1998.

[HBS97] HOLDER, OPHIR und BEN-SHAUL, I.: *A Reflective Model for Mobile Software Objects*. In: *The 17th International Conference on Distributed Computing Systems*, Seiten 339–346. IEEE Computer Society Press, Mai 1997.

[HR91] HUTCHINSON, NORMAN C. und RAJ, R. K.: *The Emerald Programming Language*. Technischer Bericht, Department of Computer Science, University of Bristish Columbia, Oktober 1991.

[Hut96] HUTCHINSON, NORMAN C.: *An Emerald Primer*. Technischer Bericht, Department of Computer Science, University of British Columbia, September 1996.

[IBM] IBM TOKYO RESEARCH LABORATORY: *Aglet API Documentation (ver 1.1b)*.

[JLHB88] JUL, ERIC, LEVY, H., HUTCHINSON, N. und BLACK, A.: *Fine-Grained Mobility in the Emerald System*. ACM Trans. on Computer Systems, 6(1):109–133, Februar 1988.

[Kat96] KATO, KAZUHIKO: *Safe and Secure Execution Mechanisms for Mobile Objects*. In: VITEK, JAN und CHRISTIAN TSCHUDIN [VT96], Seiten 201–211.

[KC86] KOSHAFIAN, SETRAG N. und COPELAND, G. P.: *Object Identity*. In: *First ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)* [ACM86], Seiten 406–416.

[KJS96] KRAMER, DOUGLAS, JOY, B. und SPENHOFF, D.: *The Java Platform, A White Paper*. Sun Microsystems, Inc., Mai 1996.

[Kle98] KLEINÖDER, JÜRGEN: *Object-Oriented Concepts in Distributed Systems*. Friedrich Alexander Universität, SS 1998. Vorlesungsskript.

[Kna96] KNABE, FREDERICK: *Performance-Oriented Implementation Strategies for a Mobile Agent Language*. In: VITEK, JAN und CHRISTIAN TSCHUDIN [VT96], Seiten 229–243.

[KTM+96] KATO, KAZUHIKO, TOUMURA, K., MATSUBARA, K., AIKAWA, S., YOSHIDA, J., KONO, K., TAURA, K. und SEKIGUCHI, T.: *Protected and Secure Mobile Object Computing in PLANET*. In: *2nd ECOOP Workshop on Mobile Object Systems*, Linz, Austria, Juli 1996.

[LA97] LANGE, DANNY B. und ARIDOR, Y.: *Agent Transfer Protocol – ATP/0.1*. IBM Tokyo Research Laboratory, März 1997.

[Lan97] LANGE, DANNY B.: *Java Aglet Application Programming Interface (J-AAPI) White Paper - Draft 2*. IBM Tokyo Research Laboratory, Februar 1997.

[LC96] LANGE, DANNY B. und CHANG, D. T.: *IBM Aglets Workbench: Programming Mobile Agents in Java, A White Paper Draft*. IBM Corporation, September 1996.

[LY96] LINDHOLM, TIM und YELLIN, F.: *The Java Virtual Machine Specification*. Sun Microsystems, Inc., September 1996.

[MBB+98] MILOJICIC, DEJAN, BREUGST, M., BUSSE, I., CAMPBELL, J., COVACI, S., FRIEDMAN, B., KOSAKA, K., LANGE, D., ONO, K., OSHIMA, M., THAM, C., VIRDHAGRISWARAN, S. und WHITE, J.: *MASIF: The OMG Mobile Agent System Interoperability Facility*. In: ROTHERMEL, KURT und FRITZ HOHL [RH98], Seiten 38–49.

[Mey87] MEYROWITZ, NORMAN (Herausgeber): *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, Band 22 der Reihe *SIGPLAN Notices*, Orlando, Florida, 4–8 Oktober 1987.

[MF96] MCGRAW, GARY und FELTEN, E.: *Java Security : hostile applets, holes & antidotes*. Wiley Computer Publishing, 1996.

[MGW97] MILOJICIC, DEJAN S., GUDAY, S. und WHEELER, R.: *Old Wine in New Bottles: Applying OS Process Migration Technology to Mobile Agents*. In: *Mobile Object Systems ECOOP Workshop*, 1997.

[MHM96] MADANY, PETER W., HAMILTON, G. und MITCHELL, J.: *JavaOS: A Standalone Java Environment, A White Paper*. Sun Microsystems, Inc., Mai 1996.

[Nor81] NORI, K. V.: *The Pascal P-code compiler: implementation notes*. In: BARRON, D. W. (Herausgeber): *Pascal - The language and its implementation*. John Wiley & Sons, New York, 1981.

[Obj96] OBJECT MANAGEMENT GROUP: *CORBAservices: Common Object Services Specification*, November 1996.

[Obj97a] OBJECTSPACE, INC.: *ObjectSpace Voyager Core Package Technical Overview, Version 1.0*, Dezember 1997.

[Obj97b] OBJECTSPACE, INC.: *ObjectSpace Voyager: Core Technology User Guide, Version 1.0.0*, 1. Auflage, 1997.

[Obj97c] OBJECTSPACE, INC.: *Voyager and Agent Platforms Comparison*, Dezember 1997.

[Obj98a] OBJECTSPACE, INC.: *Core Technology Version 2.0 Beta 2 Documentation*, 1998.

[Obj98b] OBJECTSPACE, INC.: *ObjectSpace Voyager: Core Technology User Guide, Version 2.0.0*, 1. Auflage, 1998.

[OKO98] OSHIMA, MITSURU, KARJOTH, G. und ONO, K.: *Aglets Specification (Version 1.1) Draft*, September 1998. http://www.trl.ibm.co.jp/aglets/.

[Pap89] PAPATHOMAS, MICHAEL: *Concurrency Issues in Object-Oriented Programming Languages*, Seiten 207–245. Centre Universitaire d'Informatique, University of Geneva, Juli 1989. http://cuiwww.unige.ch/OSG/publications/OO-articles/concurrency.ps.Z.

[Pic98] PICCO, GIAN PIETRO: *Understanding, Evaluating, Formalizing and Exploiting Code Mobility*. Doktorarbeit, Politecnico di Torino, Dipartimento di Automatica e Informatica, 1998.

[PRM97] PICCO, GIAN PIETRO, ROMAN, G.-C. und MCCANN, P. J.: *Expressing Code Mobility in Mobile UNITY*. Technischer Bericht WUCS-97-02, Washington University, St. Louis, Mo., Januar 1997.

[RASS97] RANGANATHAN, MUDUMBAI, ACHARYA, A., SHARMA, S. und SALTZ, J.: *Network-Aware Mobile Programs*. In: *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, Cal., Januar 1997. Version available as University of Maryland Department of Computer Science Technical Report CS-TR-3659.

[RH98] ROTHERMEL, KURT und HOHL, F. (Herausgeber): *Mobile Agents: Second International Workshop, MA'98*, Band 1477 der Reihe *LNCS*. Springer, September 1998.

[RP97] RECHENBERG, PETER und POMBERGER, G. (Herausgeber): *Informatik-Handbuch*. Hanser, 1997.

[RPZ97] ROTHERMEL, KURT und POPESCU-ZELETIN, R. (Herausgeber): *Mobile Agents : First International Workshop, MA 97*, Band 1219 der Reihe *LNCS*. Springer, April 1997.

[SBH96] STRASSER, MARKUS, BAUMANN, J. und HOHL, F.: *Mole – A Java Based Mobile Agent System*. In: *2nd ECOOP Workshop on Mobile Object Systems*, Seiten 28–35, Linz, Austria, Juli 1996.

[SGM89] SHAPIRO, MARC, GAUTRON, P. und MOSSERI, L.: *Persistence and Migration for C++ Objects*. In: COOK, STEPHEN (Herausgeber): *ECOOP'89, Proc. of the Third European Conf. on Object-Oriented Programming*, British Computer Society Workshop Series, Seiten 191–204, Nottingham (GB), Juli 1989. The British Computer Society, Cambridge University Society.

[SJ95] STEENSGAARD, BJARNE und JUL, E.: *Object and Native Code Thread Mobility Among Heterogeneous Computers*. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Seiten 68–78, Copper Mountain, Co., Dezember 1995.

[Sun] SUN MICROSYSTEMS, INC.: *Java Application Programming Interface 1.1, Users Guide*.

[Sun97] SUN MICROSYSTEMS, INC.: *Java Remote Method Invocation Specification*, Oktober 1997. Revision 1.42, JDK 1.2 Beta1.

[Sun98] SUN MICROSYSTEMS, INC.: *Java Object Serialization Specification*, November 1998. Revision 1.43, JDK 1.2.

[Tan92] TANENBAUM, ANDREW S.: *Modern Operating Systems*. Prentice Hall, 1992.

[Tsc96a] TSCHUDIN, CHRISTIAN: *Instruction-Based Communications*. In: VITEK, JAN und CHRISTIAN TSCHUDIN [VT96], Seiten 67–90.

[Tsc96b] TSCHUDIN, CHRISTIAN: *The Messenger Environment M0 – A Condensed Description*. In: VITEK, JAN und CHRISTIAN TSCHUDIN [VT96], Seiten 149–156.

[Veg86] VEGDAHL, STEVEN R.: *Moving Structures between Smalltalk Images*. In: *First ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)* [ACM86], Seiten 466–471.

[VST96] VITEK, JAN, SERRANO, M. und THANOS, D.: *Security and Communication in Mobile Object Systems*. In: VITEK, JAN und CHRISTIAN TSCHUDIN [VT96], Seiten 177–193.

[VT96] VITEK, JAN und TSCHUDIN, C. (Herausgeber): *Mobile Object Systems : Towards the Programmable Internet*, Band 1222 der Reihe *LNCS*. Springer, 1996.

[WBDF98] WICKE, CHRISTIAN, BIC, L. F., DILLENCOURT, M. B. und FUKUDA, M.: *Automatic State Capture of Self-Migrating Computations in MESSENGERS*. In: ROTHERMEL, KURT und FRITZ HOHL [RH98], Seiten 69–79.

[Weg87] WEGNER, PETER: *Dimensions of Object-Based Language Design*. In: MEYROWITZ, NORMAN [Mey87], Seiten 168–182.

[WLAG93] WAHBE, R., LUCCO, S., ANDERSON, T. E. und GRAHAM, S. L.: *Efficient software-based fault isolation*. In: *Proc. of the 14th ACM Symp. on Operating System Principles*, Seiten 203–216, 1993.

[WPW+97] WONG, DAVID, PACIOREK, N., WALSH, T., DICELIE, J., YOUNG, M. und PEET, B.: Concordia: *An Infrastructure for Collaborating Mobile Agents*. In: ROTHERMEL, KURT und RADU POPESCU-ZELETIN [RPZ97], Seiten 86–97.