Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme

Studienarbeit

Eine bidirektionale typisierte Kommunikation zwischen zwei reflexiven, objektorientierten Systemen

Design und prototypische Implementierung eines Java-Tycoon-2-Gateways

eingereicht bei:
Prof. Dr. Florian Matthes
Arbeitsbereich Softwaresysteme
TU Hamburg Harburg

vorgelegt von:
Daniel Schneider
Marktstraße 6
20357 Hamburg
Tel.: 040 - 432 24 00

Hamburg, den 28. Januar 1998

Zusammenfassung

Die Arbeit beschreibt das Design und eine prototypische Implementierung einer typisierten Kommunikation zwischen Tycoon-2 und Java. Hierbei wird der Schwerpunkt darauf gelegt, dem Tycoon-System die Funktionalität von Java zugänglich zu machen, um beispielsweise grafische Frontends für Tycoon-Programme mit Hilfe des “Java Abstract Windowing Toolkit” (AWT) erzeugen zu können.

Diese Anbindung wird durch eine Form des entfernten Methodenaufrufs realisiert, der die Verteilung größtenteils vom Anwender “versteckt”. Entfernte Methodenaufrufe unterscheiden sich aus Sicht des Benutzers also nicht von lokalen Aufrufen.

Durch die Reflexionsmechanismen, über die beide Sprachen verfügen, ist es möglich das Laufzeitsystem sehr generisch zu implementieren und damit die Codereplikation gering zu halten.

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Ziel und Gliederung der Arbeit 4

2 Analyse **5**
2.1 Verteilungsmodell 5
2.2 Performance 6
2.3 Programmierschnittstelle 6
2.4 Multi-Threading 7

3 Systemdesign **8**
3.1 Überblick 8
3.2 Schichtenmodell 9
3.2.1 Stub/Skeleton Layer 11
3.2.2 Marshal Layer 12
3.2.3 Transport Layer 14
3.3 Multi-Threading 14
3.4 Abbildung der Datentypen 16
3.4.1 Basisdatentypen 17
3.4.2 Arrays 18
3.5 Renaming 18
3.6 Basisklassen der erzeugten Stubklassen 19
3.7 Abbildung von Java-Sprachkonstrukten 20
3.7.1 Interfaces 20
3.7.2 Packages 22
3.7.3 Overloading 22
3.7.4 Callbacks 22
3.7.5 Vererbung, Klassenmethoden und Konstruktoren . . . 26
3.7.6 Slots 28
3.7.7 Exceptions 28
3.8 Optimierungen 29
3.8.1 Void-wertige Methoden mit leerer `throws`-Klausel . . 29

3.8.2 Konstruktoren mit leerer `throws`-Klausel 30
3.8.3 Andere Methoden 30
3.9 Distributed Garbage Collection 30

4 Implementierung **32**
4.1 Transport Layer 32
4.2 Marshal Layer 33
4.2.1 Transfersyntax 33
4.2.2 Kommunikationsprotokoll 34
4.3 Stub/Skeleton Layer 36

5 Zusammenfassung und Ausblick **41**

Kapitel 1

Einleitung

Tycoon hat bereits als “aktiver” WWW-Server seine Praxistauglichkeit bewiesen. Ein weiterer Schritt zur Etablierung des Systems wäre die Implementierung von komplexen Informationssystemen. Für deren Realisierung wird allerdings eine grafische Benutzeroberfläche benötigt, über die Tycoon bisher noch nicht verfügt. Kriterien bei der Wahl bzw. beim Entwurf einer solchen Oberfläche sind u. a. die folgenden:

- **Portabilität:** Tycoon existiert bereits für diverse Plattformen (Solaris, HP-UX, Windows-NT, Linux). Die zugehörige Benutzeroberfläche sollte zum einen (natürlich auch) auf diesen Plattformen zur Verfügung stehen und zum anderen sollte das API auf den unterschiedlichen Plattformen identisch sein, um den Portierungsaufwand so gering wie möglich zu halten.

- **Interaktivität:** Moderne benutzerfreundliche Informationssysteme haben hohe Anforderungen an die interaktiven Möglichkeiten ihrer Benutzeroberfläche. Um beispielsweise eine sogenannte “Bubblehelp” über einem beliebigen Grafikelement darzustellen, wenn sich der Mauszeiger darüber befindet, ist ein sehr genaues Monitoring der Mouse-Events nötig.

- **Einfache Anwendung:** Nicht zuletzt sollte die Verwendung der grafischen Oberfläche nicht zu kompliziert sein. Sie sollte sich gut in das OO-Konzept von Tycoon einbinden lassen.

Unter Berücksichtigung dieser Punkte ist die Wahl auf eine Anbindung von Java an Tycoon gefallen. Seit der raschen Entwicklung des WWW ist Java für praktisch jede Plattform verfügbar. Obendrein ist Java durch die Integration der Java-Vitual-Machine (JVM) in diverse WWW-Browser heute

auch praktisch auf jedem Rechner bereits vorhanden. Eine auf Java basierende Oberfläche bedeutet also, daß fast jeder an das WWW angeschlossene Rechner ohne zusätzlichen Aufwand als “Tycoon-Termial” verwendet werden kann.

Mit dem Java-AWT existiert hier bereits eine sehr mächtige GUI-Bibliothek, die den heutigen Anforderungen an interaktive Systeme durchaus gerecht werden kann. Außerdem bietet eine generische, nicht auf bestimmte Klassen festgelegte Anbindung von Java an Tycoon noch andere Möglichkeiten, die über die bloße Nutzung von GUI-Diensten weit hinausgehen.

Es gilt also nun unter Berücksichtigung des dritten Punktes, der einfachen Benutzbarkeit, die Verwendung von Java-Klassen von Tycoon-System aus zu ermöglichen.

1.1 Ziel und Gliederung der Arbeit

Im Rahmen der vorliegenden Arbeit werden die folgende Punkte ausgeführt:

- In Abschnitt 2 wird analysiert, wie eine GUI-Oberfläche auf Java-Basis vom Tycoon-System aus sinnvoll eingesetzt werden kann und in welcher Form diese vom Programmierer verwendet werden kann.
- Im 3. Abschnitt wird ein objektorientiertes Design vorgestellt, das den in Kapitel 2 ermittelten Anforderungen genügt.
- In Abschnitt 4 wird die im Rahmen der Arbeit erstellte prototypische Implementierung beschrieben.

Kapitel 2

Analyse

Im Folgenden wird eine kurze Analyse der Anforderungen an einen Java-Tycoon-Kommunikation durchgeführt. Hierbei wird zuerst das Verteilungsmodell beschrieben, das zu unterstützen ist. Darauf folgen die Anforderungen an Performance und Benutzbarkeit des Systems.

2.1 Verteilungsmodell

Tycoon wird momentan hauptsächlich als Serverplattform im WWW eingesetzt, die über das HTTP mehrere Clients "gleichzeitig" bedient. Diese Clients kommunizieren mittels eines beliebigen HTML-Browsers mit dem Server. Der Zustand des Systems wird ausschießlich im Server verwaltet - der Client dient lediglich als "HTML-Terminal".

Dieses Verteilungsmodell soll im wesentlichen beibehalten werden, jedoch schafft diese Arbeit die Möglichkeit, stark interaktive (reaktive) Systeme zu entwerfen. Dabei besteht aber weiterhin die Möglichkeit, eine große Anzahl von Clients im WWW zu erreichen.

Dies führt zu dem Modell eines Tycoon-Systems als *multi-threaded* Server, der mehrere in einem Netzwerk verteilte Benutzer bedienen kann. Jeder vom Server angeboten Dienst besteht dann aus einer Serverkomponente, die im Tycoon-Store als separater Thread ausgeführt wird und einer, möglicherweise generischen, Clientkomponente. Die Serverkomponente steuert den Kontrollfluß des Dienstes, verarbeitet Events und hält die Anwendungsdaten, während die Clientkomponente lediglich für die Darstellung der Daten und für das Weiterleiten von Benutzer-Events zuständig ist. Betrachtet man das Modell aus der Sicht des klassischen Model-View-Controller-Paradigmas [Krasner, Pope 88], so ist also lediglich die View-Komponente auf Client-Seite zu finden, während Model und Controller sich auf der Server-Seite

befinden. Dieser Aufbau ermöglicht es, jegliche Transaktionsdaten allein im Server zu führen, und somit nicht auf 2-Phase-Commit [Lockemann, Schmidt 87] oder andere verteilte Transaktionsmechanismen angewiesen zu sein.

2.2 Performance

Da das System zum Erstellen von WWW-Anwendungen eingesetzt werden soll, müssen geringe Bandbreiten (30-60 Kbps) und insbesondere lange Paketlaufzeiten (100-1000 ms) toleriert werden.

Es gilt also nicht allein das Kommunikationsvolumen gering zu halten, sondern vor allem die Anzahl der Requests zu minimieren, die eine direkte Antwort erfordern, und damit den Serverthread für die Dauer der Paketlaufzeiten blockieren, die sogenannten *round trip requests* [Nye 90]. Abschnitt 3.8 beschäftigt sich mit diesem Problem und stellt eine Lösung vor.

2.3 Programmierschnittstelle

Das System soll flexibel und erweiterbar gehalten werden, d.h. insbesondere, daß prinzipiell nicht nur Java-GUI-Klassen (oder irgendeine anderer festgelegte Menge von Klassen) von Tycoon aus benutzt werden können, sondern beliebige Java-Klassen dem Tycoon-System potentiell zur Verfügung stehen.

Dies führt zu einem generischen System des entfernten Methodenaufrufes ähnlich dem Java-RMI [Sun 97], den Modula-3 Network Objects [Birell et al. 93; Birell et al. 95] oder Corba [Obj 96]. Allerdings muß das Systeme keine symmetrische Lösung bieten; eine asymmetrische Lösung, die den im folgenden genannten Punkten genügt, ist ausreichend und bedeutend einfacher zu realisieren, als eine vollkommen symmetrische Objektkommunikation zwischen zwei Spachen mit unterschiedlichem Objektmodell (zum Tycoon-Objektmodell s. [Wienberg 96], zum Java-Objektmodell s. [Gosling et al. 96]).

- Unterstützung von transparenten Methodenaufrufen von Tycoon-Objekten an Java-Objekte.
- Unterstützung von Callbacks von Java nach Tycoon.
- Der Unterschied zwischen einem lokalen Tycoon-Objekt und einem entfernten Java-Objekt soll für den Anwender weitgehenden verborgen bleiben.

- Die Benutzung des Systems sollte möglichst einfach gehalten werden. Der Anwender darf also nicht mit kryptischen oder zu langen Klassennamen konfrontiert werden. Insgesamt sollte es sich möglichst "harmonisch" in das Tycoon-System und dessen Sprachkonzepte einfügen (Metaklassen, Typisierung, etc).

Erfüllt das System diese Anforderungen, so ist es dem Anwender möglich, mit verhältnismäßig einfachen Mitteln ein Frontend für Tycoon-Applikationen zu erstellen.

2.4 Multi-Threading

Ein Server sollte natürlich mehrere Clients bedienen können. Nicht zwangsläufig notwendig ist es jedoch, daß eine Verbindung zwischen Client und Server von mehreren Threads verwendet werden kann. Rekursionen über zwei Addressräume (ein entfernter Methodenaufruf, der einen Callback erzeugt, der wiederum einen entfernter Methodenaufruf erzeugt) hingegen müssen unterstützt werden, um wirklich sinnvoll mit den System arbeiten zu können. Wie man sehen wird, stellen diese jedoch an das Laufzeitsystem fast dieselben Anforderungen, wie Multithreading. Der explizite Support von Multithreading erfordert nur minimale Designerweiterungen; daher wird das hier entwickelte Desing die nebenläufige Verwendung einer Verbindung von unterschiedlichen Threads ermöglichen.

Kapitel 3

Systemdesign

Im Folgenden wird das Systemdesign der Java-Tycoon-Kommunikation (des Java2Tycoon-Systems) beschrieben. Nach einem grober Überblick über das Gesamtsystem wird das Design des Laufzeitsystems näher beschrieben.

3.1 Überblick

Java2Tycoon wurde als System des entfernten Methodenaufrufs von Tycoon nach Java mit der Möglichkeit von Callbacks entworfen. Dies bedeutet also, daß für die verwendeten Klassen der einen Seite auf der jeweils anderen eine Stubklasse erzeugt werden muß, "gegen" die der Anwender dann Programmieren kann. Diese Klassen werden von einem Stubgenerator erzeugt, der als Eingabe Klassen des einen Sytems nimmt und für diese unter Berücksichtigung einiger Nebenbedingungen Stubklassen im anderen System erzeugt.

Der Aufruf einer Methode eines Stub-Objektes stellt sich dann folgendermaßen dar (s. Abb. 3.1):

1. Ein Tycoon-Prozeß ruft die Stubmethode eines Stub-Objektes auf.
2. Das Stub-Objekt überführt die Aktualparameter mit Hilfe des Java2Tycoon-Laufzeitsystems in eine serielle Repräsentation (dieser Vorgang heißt *marshalling*) und sendet diese zur Java-Seite.
3. Der Aufruf wird am Originalobjekt ausgeführt.
4. Der Rückgabewert wird nach erfolgtem *marshalling* zurückgesendet.
5. Das Stub-Objekt "packt" den Rückgabewert mit Hilfe des Laufzeitsystems wieder aus (*unmarshalling*).

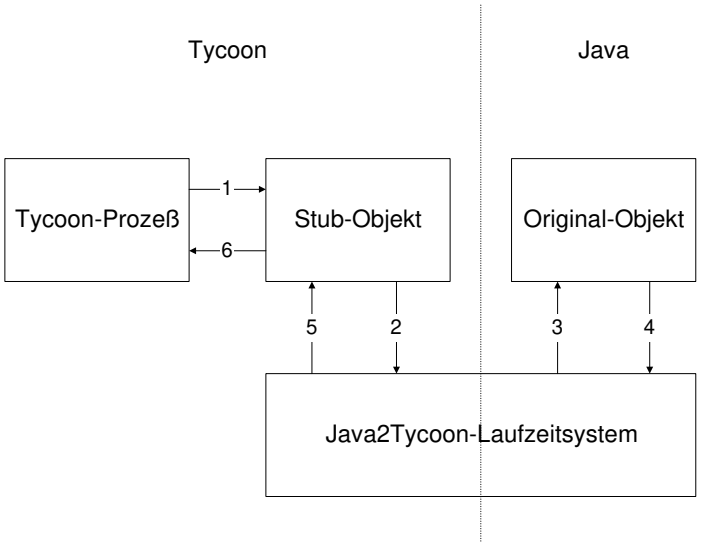

Abbildung 3.1: Verlauf eines entfernten Methodenaufrufs

6. Der Tycoon-Prozeß erhält einen Rückgabewert.

Eine detaillierte Beschreinbung des Verlaufs erfolgt in diesem Kapitel. Dabei wird das Design des Java2Tycoon-Laufzeitsytem dargestellt. Die Darstellung eines Designs des Stubgenerators wird hier nicht vorgenommen und ist nicht Gegenstand dieser Arbeit.

3.2 Schichtenmodell

Das Java2Tycoon-System besteht aus drei Schichten (layers) (s. Abb. 3.2): der Stub/Skeleton Layer, der Marshal-Layer und der Transport-Layer. Jede dieser Schichten ist durch ein Interface (bzw. eine abstrakte Klasse) spezifiziert. Sie ist somit unabhängig von den umgebenden Schichten und kann durch eine andere Implementierung ersetzt werden, ohne die anderen Schichten zu beeinflussen. Die aktuelle Implementierung der Transport-Layer basiert beispielsweise auf TCP, könnte aber zum Umgehen von Firewalls ohne Modifikation des umgebenden Systems durch eine HTTP-basierte Implementierung ersetzt werden.

Die Aufgaben der einzelnen Schichten sind wie folgt aufgeteilt:

- Stub/Skeleton Layer - auf der Client-Seite werden entfernte Objekte

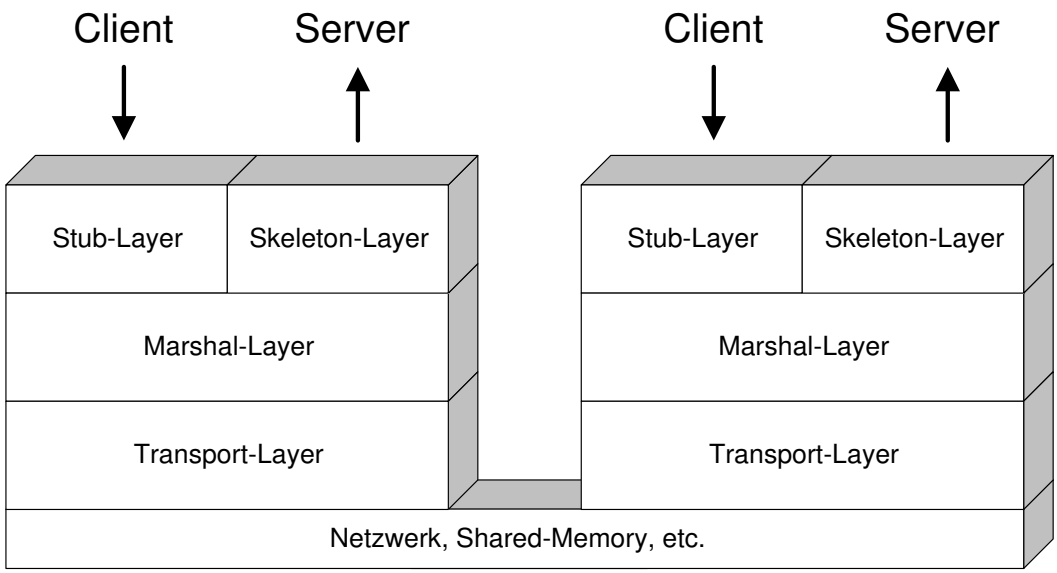

Abbildung 3.2: Das Java2Tycoon-Schichtenmodell

durch Stubs vertreten; die Methodenaufrufe am Server-Objekt werden durch die Skeleton Layer ausgeführt.

- Marshal Layer - sie serialisiert die Basisdatentypen und verwaltet die Objektreferenzen.
- Transport Layer - diese Schicht ist für den Aufbau einer streamorientierten Verbindung zwischen Client und Server zuständig.

Applikationen nutzen das System i. d. R. über Stub-Objekte. Lediglich zur Ausführung einiger Verwaltungsaufgaben kommuniziert eine Applikation direkt mit der Marshal-Layer ausgeführt (Verbindung beenden, Kommunikationsbuffer leeren, etc.)

Diese Schichtenaufteilung ist an die Architektur des Java-RMI [Sun 97] angelehnt, unterscheidet sich aber insbesondere in der zweiten Schicht. Sie übernimmt im Java-RMI-System weitreichende Aufgaben zur Ausfallsicherheit und Objektreplikation, die hier nicht benötigt werden.

3.2.1 Stub/Skeleton Layer

Die Stub/Skeleton-Layer besteht aus der Stub-Layer auf Client-Seite und der Skeleton-Layer auf Server-Seite (ein Prozeß wird im Laufe der Java2Tycoon-Kommmunikation normalerweise beide Seiten bekleiden).

Stub-Objekte sind Stellvertreter, die Methodenaufrufe an einem Objekt entgegennehmen und diese über das Netzwerk mit Hilfe der daruntergelegenen Schichten an das Orginal-Objekt weiterleiten. Der eigentliche Methodenaufruf wird dann von der gegenüberliegenden Skeleton-Layer übernommen.

Jeder Java-Klasse, die von Tycoon aus genutzt werden soll, entspricht eine Stub-Klasse innerhalb des Tycoon-Systems. Diese enthält für jede "relevante" Methode (der Begriff relevante Klasse/Methode wird in 3.7.2 näher erläutert) der Java-Klasse eine ggf. umbenannte Methode (s. hierzu 3.5) mit korrespondierender Signatur.

Über das bloße Weiterleiten von Methodenaufrufen hinaus wird in den Stub-Methoden außerdem entschieden, welche Optimierungen an dem zu sendenden Aufruf vorgenommen werden sollen (Optimierungen werden in Abschnitt 3.8 näher besprochen).

Eine Schnittstelle der Stub-Layer kann nicht angegeben werden, da diese der Schnittstelle des zu vertretenden Objektes entspricht.

Die Aufgabe der Skeleton-Layer ist es, von der Marshal Layer empfangene entfernte Methodenaufrufe auszuführen. Der Methodenselektor liegt hierbei als `String` vor, die Aktualparameter werden in einem `Array(Object)` (bzw. `java.lang.Object[]`) an die Skeleton-Layer übergeben.

Die Skleleton-Layer der Java-Seite hat die folgende Schnittstelle:

```
1   /** the SkeletonLayer is used by the MarshalLayer to perform
2      calls on local objects with local parameters
3   */
4   package hox.j2t;
5
6   public interface SkeletonLayer {
7
8     Object performVirtualCall (VirtualCall c)
9       throws SkeletonLayerException;
10
11    Object performStaticCall (StaticCall c)
12      throws SkeletonLayerException;
13
14  }
```

Ein `StaticCall` ist der Aufruf einer Java-Klassenmethode, ein `VirtualCall` ist der Aufruf einer Objektmethode. Beide Klassen enthalten keine Methoden. Sie fassen lediglich die Attribute eines Methodenaufrufes zusammen. Diese sind im Falle von `VirtualCall`

```
public String selector;      // der Methodenselektor
public Object[] params;      // die Parameter
public selfRef Object;       // das Objekt, in dessen Umgebung die Methode
                             // ausgeführt werden soll
```

`StaticCall` enthält anstelle von `selfRef` den Namen der Klasse, in der die statische Methode aufgerufen werden soll.

```
public String className;
```

Wird nun die Methode `performVitualCall` aufgerufen, so hat die Skeleton Layer die Aufgabe, am Object `selfRef` die Methode `selector` mit den Parametern `params` zu rufen. Sie kann dies mit Hilfe von Skeleton-Klassen (das Java-RMI [Sun 97] verfolgt diesen Ansatz) oder der reflexiven Möglichkeiten der Sprache generisch ausführen.

3.2.2 Marshal Layer

Ein Objekt vom Typ Marshal-Layer repräsentiert ein Verbindung zu einem Kommunikationspartner (Java bzw. Tycoon). Entsprechend werden verbindungsbezogene Methoden angeboten (`flush, disconnect`).

Die Hauptaufgabe der Marshal-Layer ist jedoch das Serialisieren der Basisdatentypen und das Verwalten von Objektreferenzen. Zudem verwaltet sie den Thread, der auf Callbacks von der Gegenseite wartet (`CallDispatcher`) und das Objekt, das die eintreffenden Rückgabewerte den entsprechenden wartenden Threads zuordnet (`ReplyDispatcher`).

Wird von der Stub-Layer mit `sendVirtualCall` die Ausführung einer Objektmethode angefordert, so überführt die Marshal-Layer die Aktualparameter abhängig von ihrem Laufzeittyp entweder in eine serielle Repräsentation (marshalling), wenn es sich um einen Basistyp (s. 3.4.1) handelt, oder versendet einen eindeutigen Objektschlüssel (`uid`), falls es sich um einen Objekttypen handelt.

Beim Empfang des Rückgabewertes wird aufgrund eines mitgesendeten Typecodes ermittelt, ob der Wert Instanz eines Basistyps ist. Ist dies der Fall, so wird er aus seiner serialisierten Form wieder in seine Objektform gebracht. Handelt es sich um einen Objekttypen, so wurde lediglich seine

`uid` empfangen. Die Marshal-Layer versucht nun, aus ihren Tabellen das zur `uid` gehörige Stub-Objekt zu finden und an die aufrufende Methode zurückzugeben. Ist das entsprechende Objekt nicht in den Tabellen enthalten, so fordert die Marshal-Layer das entsprechende Klassenobjekt auf, einen neuen Stub für die empfangene `uid` zu erzeugen. Der Name der verantwortlichen Stubklasse wird mit der `uid` versendet. Durch die reflexiven Möglichkeiten von Tycoon und Java kann aus diesem String das entsprechende Klassenobjekt ermittelt werden.

Das folgende Listing zeigt die Schnittstelle der Marshal-Layer auf Tycoon-Seite.

```
1   (*
2   The (abstract) interface of the Java-to-Tycoon marshal layer.
3   *)
4
5   class J2TMarshalLayer
6   super J2T
7   metaclass AbstractClass
8   public methods
9
10  flush()
11    (* flush all streams *)
12
13  disconnect()
14
15  startDeamons()
16    (* starts callback deamon, ReplyDispatcher, CallDispatcher *)
17
18  (* Methods invoked by Stubclasses *)
19  sendStaticCall (c :J2TStaticCall) :Object
20
21  sendVirtualCall (c :J2TVirtualCall) :Object
22
23  createUidFor(stub :J2TStub) :Int
24    (* creates a new uid and inserts Object into the J2TObjectTable *)
25
26  remoteAddress() :IPAddress
```

3.2.3 Transport Layer

Aufgabe der Tranport-Layer ist es, der Marshal-Layer zwei bidirektionale Kommunikationsströme zu einem Kommunikationspartner zur Verfügung zu stellen. Die aktuelle Implementierung realisiert dies auf der Basis von TCP, eine Implementierung mit HTTP (zur Überwindung von Firewalls) oder über shared memory (falls sich Client und Server auf dem selben Rechner befinden) ist aber ohne weiteres zu integrieren. Wichtig ist dabei lediglich, daß die o. g. Ströme zum transparenten Senden von Bytes bereitgestellt werden.

Die Streams `CallStream` und `ReplyStream` werden zur Versendung von Requests an das jeweils andere System und zum Empfangen von Return-Werten verwendet.

Die Streams `CallbackStream` und `CallbackReplyStream` werden zum Empfangen von "Upcalls" und zum Senden ihrer Return-Werten verwendet.

Das folgende Listing zeigt die Schnittstelle der Transport-Layer auf Java-Seite.

```
1    package hox.j2t;
2
3    import java.io.BufferedOutputStream;
4    import java.io.BufferedInputStream;
5
6    public interface TransportLayer {
7      public void connect (String host)
8        throws TransportLayerException;
9
10     public void disconnect ();
11
12     public BufferedInputStream getCallStream ();
13
14     public BufferedOutputStream getReplyStream ();
15
16     public BufferedInputStream getCallbackStream ();
17
18     public BufferedOutputStream getCallbackReplyStream ();
19   }
```

3.3 Multi-Threading

Die bereits in Abschnitt 3.2 erwähnten Kommunikationsströme sind schematisch in Abb. 3.3 dargestellt. Hierin sind die funktionalen Elemente

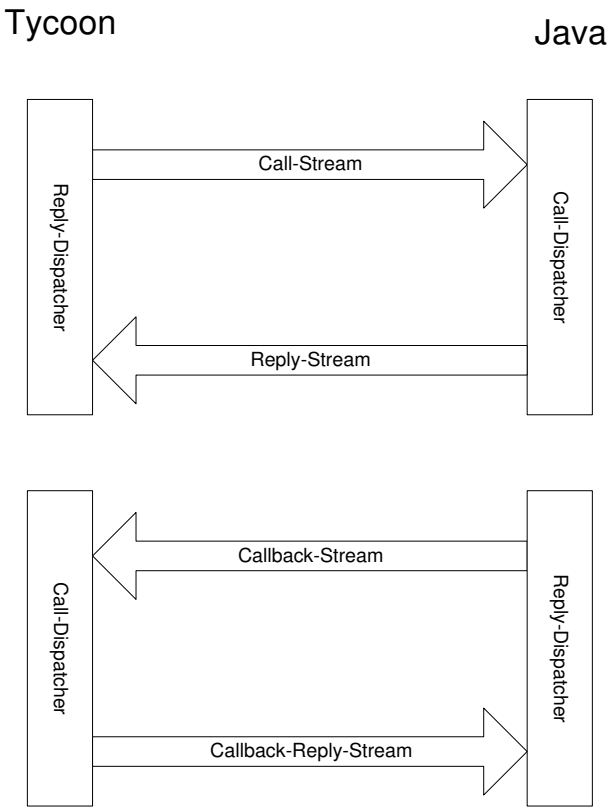

Abbildung 3.3: Kommunikationsströme aus der Sicht von Tycoon

`ReplyDispatcher` und `CallDispatcher` dargestellt. Beide sind notwendig, um Multithreading und Rekursionen über zwei Addressräume zu erlauben.

Aufgabe des `ReplyDispatchers` ist es zum einen, einen Call vor dem Versenden mit einer Thread-ID zu versehen, diesen durch den `CallStream` zur Gegenseite zu senden und den initiierenden Thread anzuhalten. Zum anderen empfängt er durch den `ReplyStream` Antworten von der Gegenseite, ordnet diese aufgrund der Thread-ID dem entsprechenden wartenden Thread zu und "weckt" diesen "auf". Ein direktes, blockierendes Lesen des initiierenden Threads vom `ReplyStream` verbietet sich, da hierdurch der Stream auch für andere Threads blockiert wäre.

Der `CallDispatcher` stellt das Gegenstück zum `ReplyDispatcher` dar. Er empfängt Calls (aus seiner Sicht) durch den `CallbackStream` und ordnet sie den entsprechenden Ausführungsqueues zu. Jede Queue wird von ei-

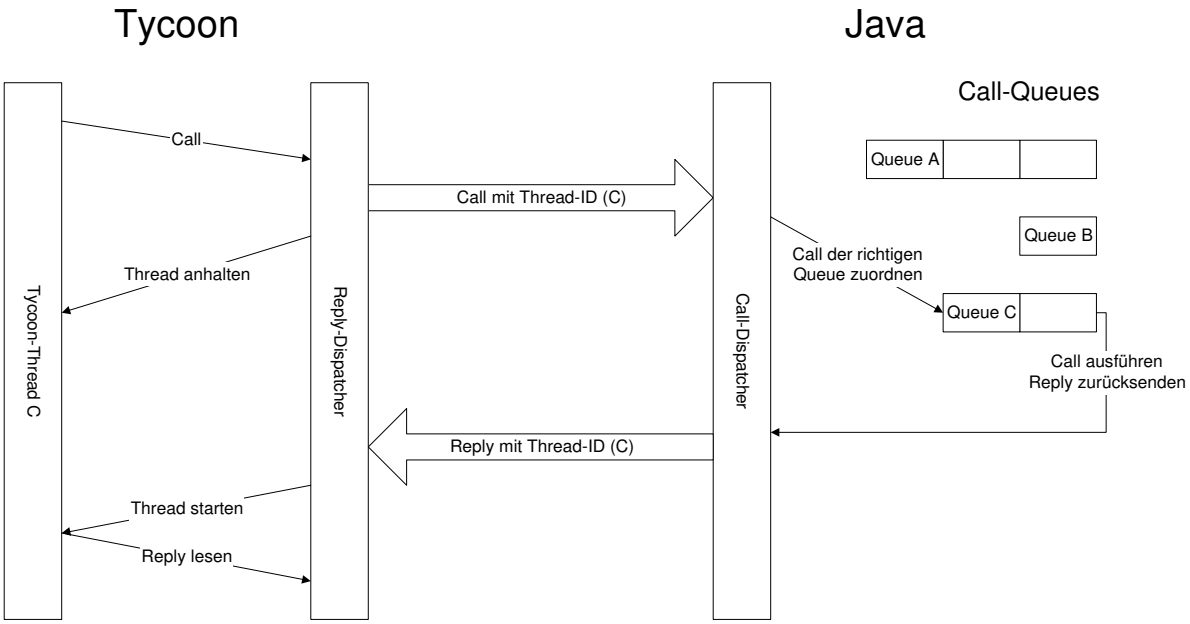

Abbildung 3.4: Arbeitsweise von `CallDispatcher` und `ReplyDispatcher`

nem eigenen Thread abgearbeitet. Vor den Senden einer Antwort wird diese vom `CallDispatcher` mit der ursprüglichen Thread-ID versehen, damit auf der Gegenseite der entsprechende wartende Thread fortgesetzt werden kann. Abbildung 3.4 zeigt den Verlauf eines Aufrufes.

Dieses Design erlaubt es, *one way calls* einzusetzten (s. Abschnitt 3.8) und Mutithreading zu unterstützen, beachtet dabei aber die Ausführungsreihenfolge innerhalb der einzelnen Threads.

3.4 Abbildung der Datentypen

Tritt ein Wert als Parameter oder Rückgabewert einer von Tycoon aufgerufenene Java-Methode auf, so entscheidet das System zur Laufzeit aufgrund des zugehöringen Typs, wie dieser Wert versendet werden soll. Handelt es sich um den Wert eines "Basisdatentypen", so wandelt das Laufzeitsystem den Wert zunächst in seinen Transfertyp, konvertiert diesen in seine lineare Form und versendet ihn dann.

Handelt es sich nicht um einen "Objektdatentyp", so wird lediglich eine Referenz auf das entsprechende Objekt zusammen mit dem Tycoon-

Java-Datentyp	Transfertyp	Tycoon-Datentyp
`byte`	`CHAR` (8 bit)	`Char` (8 bit)
`char`(16 bit)	`CHAR` (8 bit - Unicode wird mit aktueller Locale umgesetzt)	`Char`
`double`(64 bit)	`REAL`	`Real` (64 bit)
`float`(32 bit)	`REAL`	`Real` (64 bit)
`int`(32 bit)	`INT` (32 bit)	`Int` (32 bit)
`long`(64 bit)	`LONG` (64 bit)	`Long` (64 bit)
`short`(16 bit)	`INT` (32 bit)	`Int` (32 bit)
`boolean`	`BOOL`	`Bool`
`java.lang.String`	`STRING` (Unicode wird mit aktueller Locale umgesetzt)	`String`

Tabelle 3.1: Abbildung von Java-Basistypen auf Transfertypen und auf Tycoon-Typen

Name seiner Klasse (s. Renaming, Abschnitt 3.5) versendet. Existiert für die Klasse kein Tycoon-Name, so verwendet das Laufzeitsystem den Tycoon-Namen der Superklasse, usf. Für die versendete Referenz wird darauf auf der Empfängerseite ein Stellvertreterobjekt (Stub) erzeugt, das dieselbe Schnittstelle wie das Originalobjekt hat und konkrete Methodenaufrufe mit den Aktualparametern an diese weiterleitet.

3.4.1 Basisdatentypen

Hier werden jene Datentypen aufgelistet, die zur Übertragung serialisiert werden. Welcher Java-Typ letztlich in welchen Tycoon-Typ gewandelt wird geht aus Tabelle 3.1 hervor. Welcher Tycoon-Typ in welchen Java-Typ, geht aus Tabelle 3.2 hervor.

Die Konvertierung von Java `char` und `String` ist nicht verlustfrei, da ein `char` in Tycoon nur 8 bit, in Java aber 16 bit lang ist (Java verwendet Unicode-Zeichen). Daher werden `char`-Daten vor Übermittlung vom Java zum Tycoon-System der mit der aktuellen Locale (Sprach- und Ländereinstellung) in `CHAR` umgewandelt und erscheinen dann auf Tycoon-Seite als `Char`.

Tycoon-Datentyp	Transfertyp	Java-Datentyp
`Char`	`CHAR`	`char`
`Real`	`REAL`	`double`
`Int`	`INT`	`int`
`Long`	`LONG`	`long`
`True`	`BOOL`	`boolean`
`False`	`BOOL`	`boolean`
`String`	`STRING`	`java.lang.String`

Tabelle 3.2: Abbildung von Tycoon-"Basistypen" auf Transfertypen und auf Java-Typen

3.4.2 Arrays

Arrays stellen im Java2Tycoon-System Objekt-Typen dar. Ihre Versendung erfolgt also per Referenz. Auf Tycoon-Seite existiert eine generische Stubklasse `JavaArray(T <: Object)`, die Stub für `T[]` ist.

`JavaArray(T <: Object)` erbt von `MutableArray(T <: Object)` und überschreibt die Methoden `"[]"`, `"[]:="`, `size` und `equalityHash`. Die ersten drei werden in "Array Access Requests" umgesetzt; `equalityHash` liefert `size` als Ergebnis[1].

Ein String-Array `String[]` auf Java-Seite erscheint auf Tycoon-Seite also als `JavaArray(String)` und kann in Tycoon wie ein `MutableArray(String)` verwendet werden.

3.5 Renaming

Im Rahmen der vorliegenden Arbeit wurde eine Analyse der Java-AWT-Klassenstruktur vorgenommen, um beurteilen zu können, wie wichtig die Umsetzung einzelner Java-Sprachkonzepte im Hinblick auf eine Anbindung des Java-AWTs an Tycoon ist.

Hierbei wurde u. a. die Anzahl der von einer Klasse A abhängigen Klassen ermittelt. Abhängigkeit wurde dabei wie folgt definiert:

Eine Klasse A ist von B abhängig, wenn A-B in der transitiven Hülle der folgenden Relation enthalten ist:

- B ist Superklasse von A **oder**

[1] der `equalityHash` eines Objektes wird bei Hashtabellen verwendet. Es muß gelten, daß bei zwei Objekten, die als "gleich" zu betrachten sind, die Methode `equalityHash` den identischen Wert liefert (`x.equals(y)` $\Rightarrow$ `x.equalityHash = y.equalityHash`).

- B ist ein Interface und wird von A implementiert **oder**
- B ist der Typ einer öffentlichen Instanz- oder Klassenvariablen von A **oder**
- B ist in der Signatur einer Objekt- oder Klassenmethode von A enthalten **oder**
- B ist eine Exception und in der `throws` Klausel einer Objekt- oder Klassenmethode von A enthalten

Bei der Berechnung dieser Relation haben sich durchaus überraschende Ergebnisse ergeben. So ist die Klasse `java.awt.Frame` beispielsweise von 124 anderen Klassen abhängig, darunter Klassen wie `java.net.Url` und `java.lang.ClassLoader`.

Anstatt nun aufgrund einer gewünschten Klasse für 124 Java-Klassen Tycoon-Stub-Klassen zu erzeugen, nur um das volle Inteface der Klasse auf Tycoon-Seite anzubieten, wurde bei Java2Tycoon der entgegengesetzte Ansatz gewählt: Der Benutzer gibt an, welche Klassen der Java-Klassenbibliothek für seine Anwendung "relevant" sind. Das System beschneidet daraufhin die Schnittstellen der einzelnen Klassen so, daß keine relevante Klasse von einer nicht-relevanten abhängig ist. Dabei ist zu beachten, daß einen relevante Klassen nicht von einer nicht-relvanten Klassen erben darf.

Da Java-Klassen, wie in 3.7.2, beschrieben ohnehin eine anderen Namen im Tycoon-System erhalten müssen, wird die Benennung der relevanten Klassen und das Renaming von Java nach Tycoon in einem gemeinsamen Schritt ausgeführt.

Dies bedeutet also, daß nur jene Java-Klassen, die vom Benutzer mit einem Tycoon-Namen versehen wurden, als relevant angesehen werden und später dem Tycoon-System zur Verfügung stehen. Methoden, deren Signaturen eine nicht-relevante Klasse enthalten, werden nicht in das Interface der Tycoon-Stubklasse aufgenommen. Für nicht-relevante Klassen wird keine Stubklasse generiert.

3.6 Basisklassen der erzeugten Stubklassen

Die gemeinsame Basisklasse aller Stubklassen im Tycoon-System ist die Klasse `J2TStub`. Sie enthält einen Slot für die `uid` (der unique identifier dieses Objekts, der auf beiden Seiten als Referenz für dieses Objekt gilt) und einen

Slot für die Connection, die für dieses Objekt verantwortlich ist. Alle entfernten Methodenaufrufe an diesem Objekt werden an dieser Verbindung ausgeführt.

Von `J2TStub` erben `J2TInterface` und `J2TObject`, die lediglich eine Implementierung der abstrakten Methode `isInterface` aus `J2TStub` liefern. `J2TInterface` beerbt alle Stubs von Java-Interfaces, `J2TObject` beerbt alle Stubs von Java-Objekten.

Die Klasse `J2TStubClass(T <: J2TStub)` ist gemeinsame Basisklasse aller Metaklassen von Java-Objekten.

Die Methode `__createStubFor(uid :Int)` erzeugt einen neuen Stub vom Typ `T` und setzt dessen `uid` und `conn`.

`__createNewStub()` erzeugt einen neuen Stub vom Typ `T` und ermittelt selbst eine neue `uid` (diese Methode wird von den optimierten Konstruktoren benötigt; s. Abschnitt 3.8.2)

`new` ist eine Default-Implementierung eines Konstuktors mit leerer Parameterliste, die somit allen erbenden Metaklassen zur Verfügung steht. Sie ruft in der Umgebung der entsprechende Java-Klasse den Konstruktor mit leerer Parameterliste auf.

`javaClassName` liefert den Namen der zugehörigen Java-Klasse und muß von den Subklassen überschrieben werden.

`conn` gibt die mit dem aktuellen Thread assoziierte Connection zurück, die zum Aufruf von statischen Methoden verwendet wird.

Ein Diagramm der Basisklassen wird in Abb. 3.5 gezeigt. Der gestrichelte Pfeil bedeutet hier "hat als Metaklasse". Auf bisher nicht erwähnte Klassen wird im folgenden Abschnitt 3.7 eingegangen.

3.7 Abbildung von Java-Sprachkonstrukten

Die Sprachkonstrukte von Java lassen sich meist direkt in Tycoon-Sprachkonstrukte umsetzen. Ein Java-Objekt wird zu einem Tycoon-Objekt, eine Java-Methode zu einer Tycoon-Methode. Die Umsetzung der Java-Konstrukte, die sich nicht direkt in Tycoon wiederfinden, wird im folgenden einzeln beschrieben.

3.7.1 Interfaces

Java-Interfaces sind im Prinzip völlig abstrakte Klassen. Die Trennung zwischen Interface und abstrakter Klasse rührt daher, daß Java nur Einfach-Vererbung erlaubt. Da Tycoon dieser Beschränkung nicht unterworfen ist, werden Java-Interfaces zu abstrakten Tycoon-Klassen. Implementiert eine

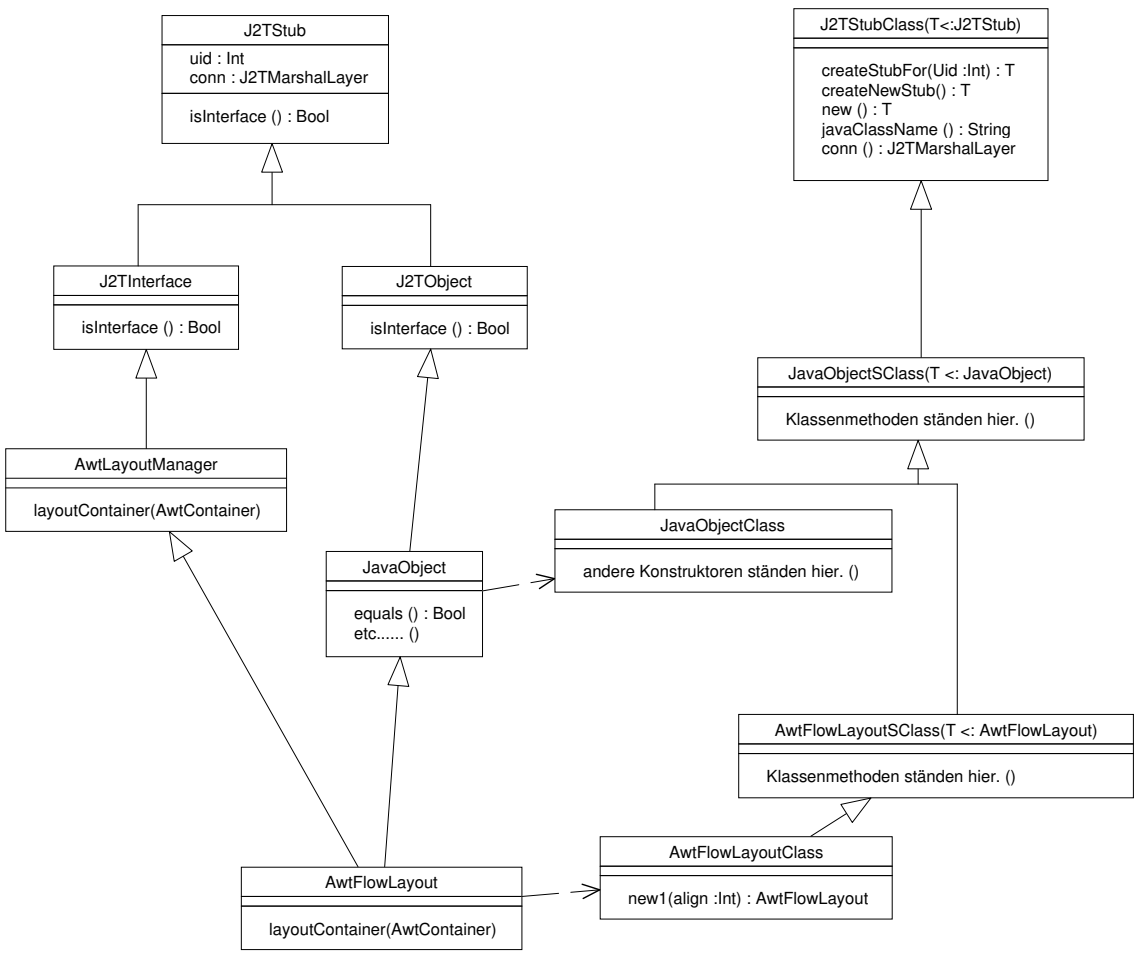

Abbildung 3.5: Basisklassen der Java2Tycoon-Stub-Klassen

Java-Klasse ein Interface, so erbt die korrespondierende Tycoon-Klasse von der zugehörigen abstrakten Tycoon-Klasse.

In Abb. 3.5 erbt `AwtFlowLayout` zuerst von `JavaObject` und dann von `AwtLayoutManager`. Dies entsprich dem Klassenaufbau der Originalklassen: `java.awt.FlowLayout` erbt von `java.lang.Object` und implementiert das Interface `java.awt.LayoutManager`

3.7.2 Packages

Java erlaubt die Zusammenfassung von Klassen zu Packages. Um eine Klasse voll zu qualifizieren, muß dem Klassenname der Packagename vorangestellt werden (`java.awt.Frame` benennt die Klasse `Frame` im Package `java.awt`).

Damit die entsprechende Tycoon-Klasse eindeutig zu benennen ist, und um keine überlangen Klassenname zu erhalten, wird von einer automatischen Zuordnung der Java-Namen zu ihren Tycoon-Namen abgesehen. Stattdessen erstellt der Benutzer eine Umsetzungstabelle, die einem Java-Namen den gewünschten Tycoon-Namen zuordnet (s. Renaming, Abschnitt 3.5). Dies führt zu kürzeren Klassenname und somit zu übersichtlicheren Programmtexten.

3.7.3 Overloading

Overloading in Java bedeutet, daß eine Methode innerhalb einer Klasse nicht allein durch ihren Selektor sondern erst durch Selektor **und** Typen der Formalparameter eindeutig bestimmt ist.

Tycoon unterstützt kein Overloading, daher müssen überladene Methoden je nach Signatur einen anderen Namen erhalten. Der Stubgenerator könnte hier eine automatische Umbenennung vorzunehmen, die jedoch entweder zu sehr langen oder zu kryptischen Methodennamen führen würde. Alternativ kann auch eine Umbenennung angegeben werden, die einer Methode mit einer bestimmten Folge von Formalparametertypen einen neuen Namen in der Stubklasse zuweist.

3.7.4 Callbacks

Java2Tycoon stellt die Funktionalität von Java dem Tycoon-System zur Verfügung. Ein Tycoon-Programm kann also beliebige (vorhandene) Methoden an einem Java-Objekt aufrufen. Als Antwort erhält die aufrufende Methoden den Rückgabewert der Java-Methode. Diese Form der Kommunikation ist insbesondere in ereignisgesteuerten GUI-Umgebungen zu ein-

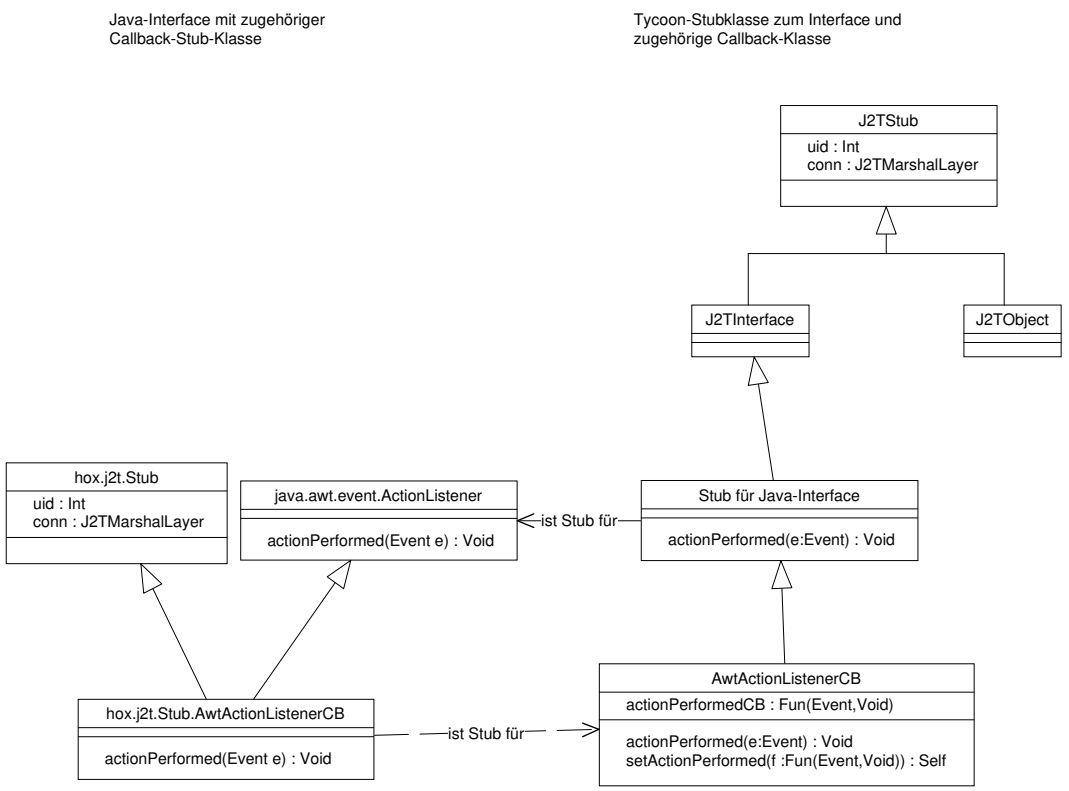

Abbildung 3.6: Der Callback-Mechanismus (Beispiel: `ActionListener`)

geschränkt. Es wird zusätzlich die Möglichkeit benötigt, bestimmte Tycoon-Objekte von Tycoon an Java zu übergeben. Werden an diesen Objekten Methoden aufgerufen, so wird die entsprechende Methode im Tycoon-Store ausgeführt. Dieser Mechanismus wird Callback genannt.

Bei der Stubgenerierung kann der Benutzer angeben, Objekte welcher Java-Interfaces er als Callback verwenden möchte. Für diese werden dann auf Tycoon-Seite Callback-Klassen erzeugt, die alle Methoden des Interfaces implementieren, indem der Call an eine entsprechende Funktion höherer Ordnung weitergeleitet wird.

Auf Java-Seite werden nun für diese Callback-Klassen Stubs erzeugt, die jeden Methodenaufruf an das Tycoon-Objekt weiterleiten. Dies wird allgemein in Abbildung 3.6 gezeigt.

Angenommen der Benutzer möchte ein Objekt der Klasse `java.awt.event.ActionListener` als Callback installieren. Dieses In-

terface bekommt den Tycoon-Namen `AwtActionListener`. Das System erzeugt nun eine ensprechende Callback-Klasse (`AwtActionListenerCB`). Für diese Klasse wird auf Java-Seite ein Stub erzeugt, dessen Name `hox.j2t.stubs.AwtActionListenerCB` lautet. Der Benutzer kann nun Funktionsobjekte bei dem `AwtActionListenerCB` registrieren, die gerufen werden, wenn eine der Methoden am Stub-Objekt aufgerufen wird.

Im folgenden Listing wird in Zeile 5 ein Callback-Objekt erzeugt. Mit der Methode `AwtActionListenerCB::setActionPerformed()` registriert der Benutzer eine Funktion, deren Aufruf erfolgt, wenn die Methode `ActionPerformed` am Stub-Objekt aufgerufen wird.

Im Weiteren kann das Callback-Objekt wie ein Objekt vom Typ AwtActionListener verwendet werden. Zum Beispiel wird in Zeile 23 das Objekt `cb` als Event-Listener von `button` eingetragen.

```
1    J2TProps.instance.nameServer.bindService("buttons",
2      fun (conn :J2TMarshalLayer) {
3
4        (* Create callback *)
5        let cb = AwtActionListenerCB.new,
6        cb.setActionPerformed(
7          fun( e :AwtActionEvent ) {
8            tycoon.stdout << e.getActionCommand() << "\n",
9            nil
10         }
11       ),
12
13       (* create a Frame *)
14       let f = AwtFrame.new1("Test"),
15       f.setSize(600,600),
16
17       (* set BorderLayout *)
18       let layout = AwtBorderLayout.new(),
19       f.setLayout(layout),
20
21       (* add a button *)
22       let button = AwtButton.new1("a button"),
23       button.addActionListener(cb),
24       f.addWithString("Center", button),
25
26       (* show it *)
27       f.show,
```

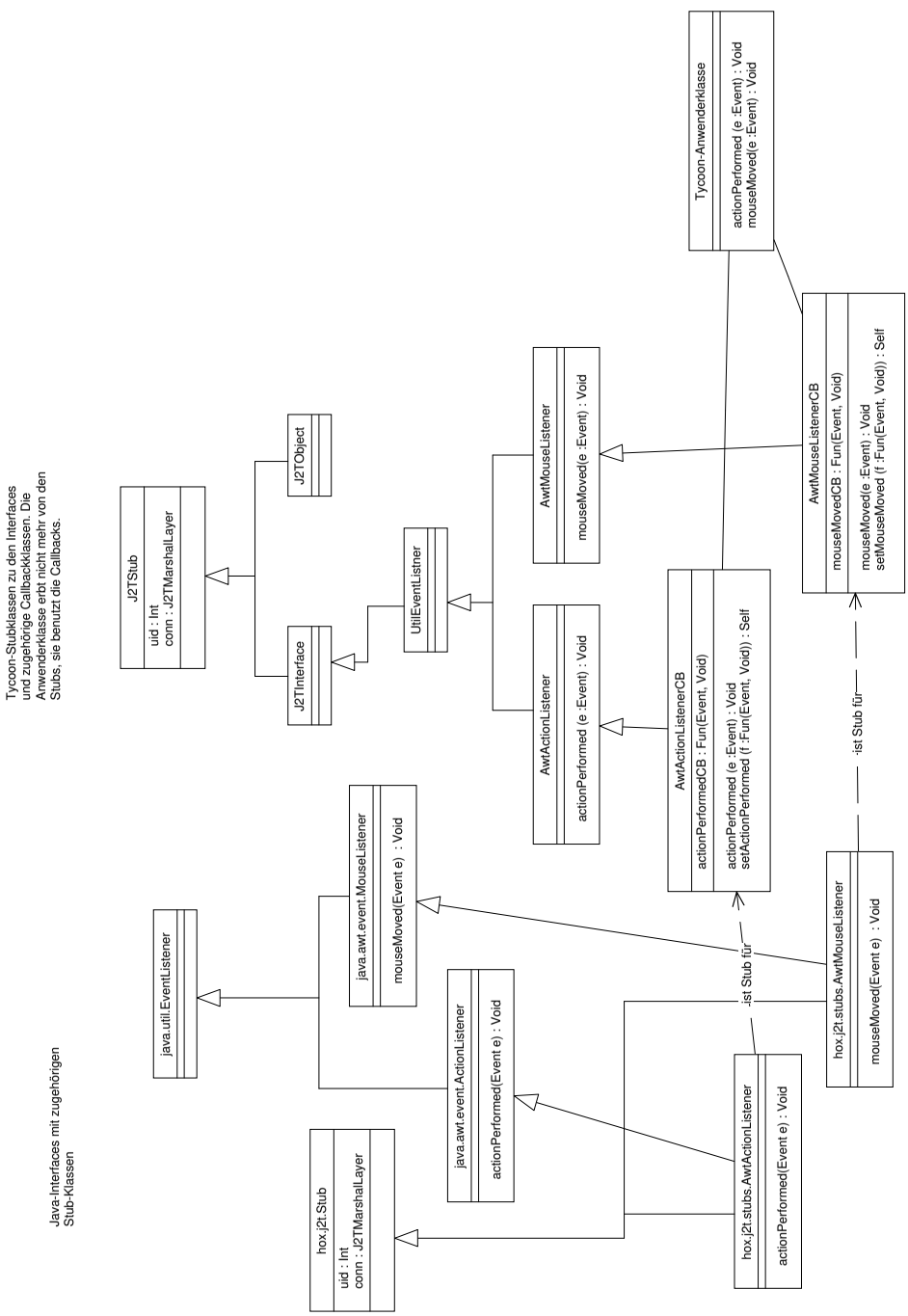

Abbildung 3.7: Eine Anwenderklasse benutzt zwei Callback-Klassen

```
28      }
29  )
```

Die Tatsache, daß hier die einzelnen Methoden über Funktionen höherer Ordnung angesprochen und nicht als zu überschreibende Methoden realisiert werden, liegt darin begründet, daß die Möglichkeit bestehen sollte, mit einem bestimmten Tycoon-Objekt mehrere Java-Callback-Intefaces zu implementieren. Würde dies über Vererbung realisiert, so hätte es zur Folge, daß einem Tycoon-Objekt mehrere Java-Objekt zugeordnet wären (s. Abb. 3.8), was weitreichende Probleme mit sich brächte. Bei einer Implementierung auf der Basis von Funktionen höherer Ordnung erzeugt sich das betreffende Tycoon-Objekt lediglich seine Callback-Objekte, erbt aber nicht von diesen. Im Ergebnis stehen also zwei Tycoon-Objekten ebensovielen Java-Objekte gegenüber (s. Abb. 3.7), womit die Systemintegrität gewahrt bleibt.

Dies entspricht der Diskussion "Inheritance vs. Object Composition" (Vererbung oder Objektkomposition) [Gamma et al. 95]. Auch in diesem Fall resultiert aus der größeren Flexibilität des Ansatzes der Objektkomposition eine unübersichtlichere Klassenstruktur.

3.7.5 Vererbung, Klassenmethoden und Konstruktoren

Für jede Java-Klasse werden drei Tycoon-Stub-Klassen erzeugt:

1. Die Objektklasse. Sie enthält alle öffentlichen Objektmethoden und Objektslots der Java-Klasse.

2. Die statische Metaklasse. Sie enthält alle öffentlichen statischen Methoden und Slots der Java-Klasse.

3. Die Konstruktor-Metaklasse. Sie enthält alle öffentlichen Konstruktoren der Java-Klasse.

Die Beziehungen zwischen den o. g. Klassen stell sich wie folgt dar (siehe Diagramm 3.9):

Gegeben sei eine Java-Klasse A mit Superklasse S. Die korrespodierenden Tycoon-Klassen heißen A-Object, A-Static und A-Constructor für die Klasse A und entsprechend S-Object, S-Static und S-Constructor für die Superklasse. Dann wird gelten:

- A-Object erbt von S-Object

- A-Object hat die Metaklasse A-Constructor

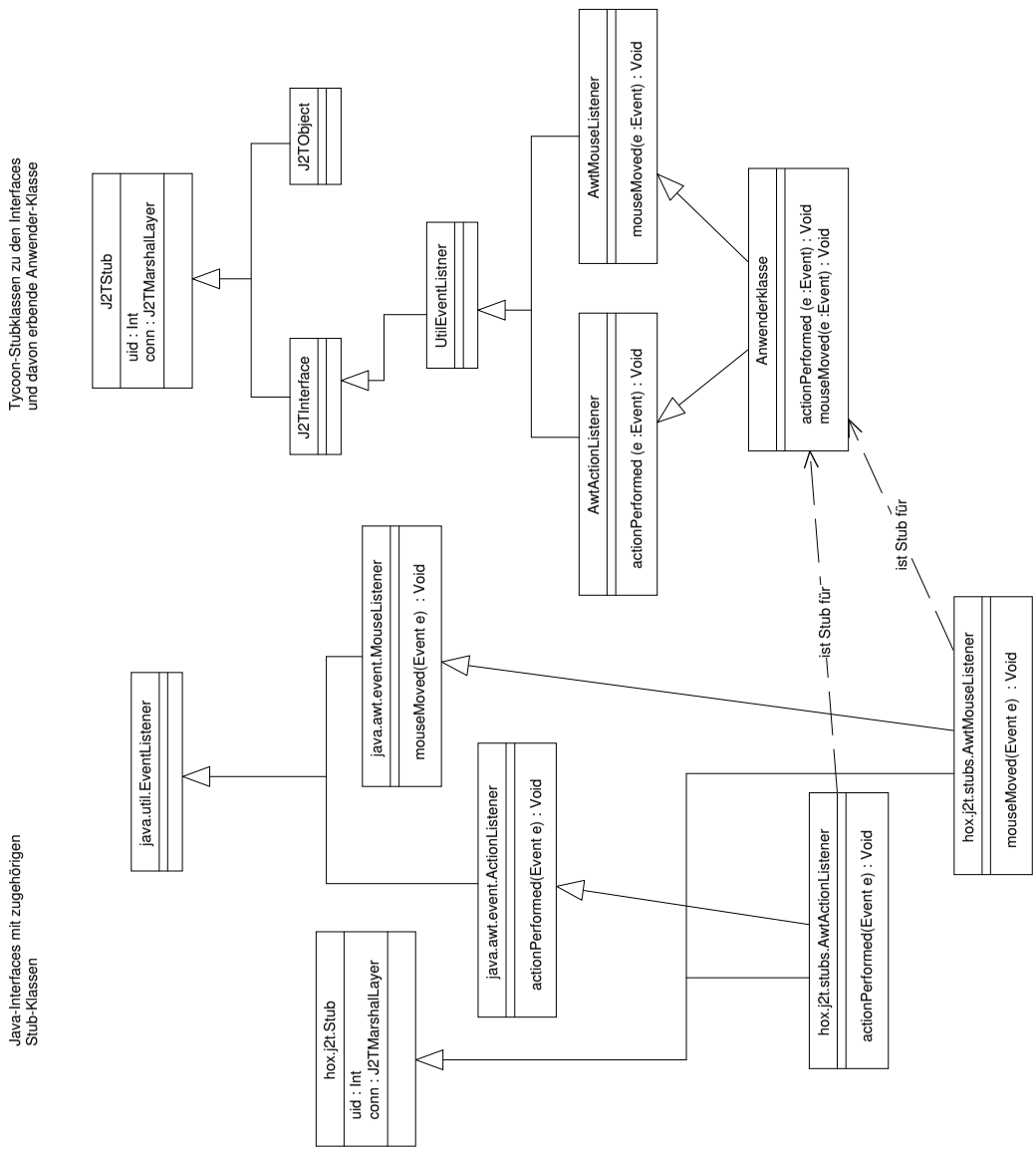

Abbildung 3.8: Das Modell aus Abb. 3.7 per Vererbung realisiert.

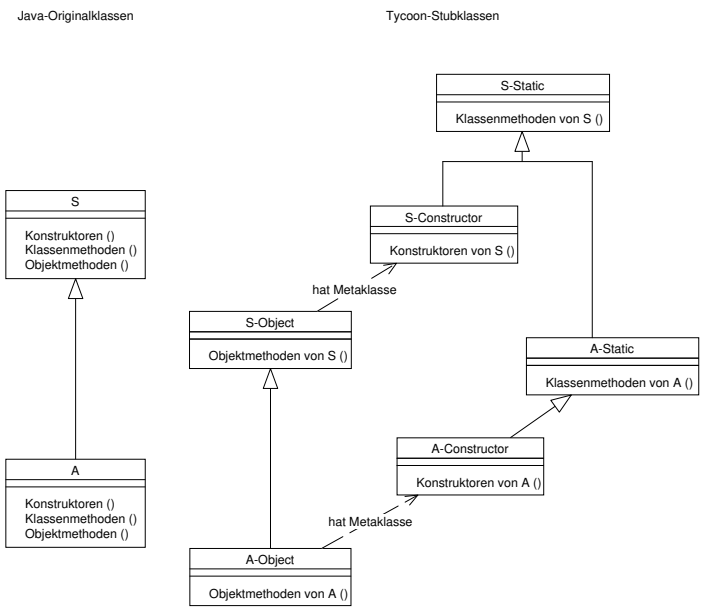

Abbildung 3.9: Umsetzung der Java-Vererbungsbeziehung

- A-Constructor erbt von A-Static
- A-Static erbt von S-Static

Diese Konstruktion ergibt sich aus der Situation, daß Klassenmethoden in Java zwar von Superklassen geerbt werden, Konstuktoren jedoch nicht.

Ein konkretes Beispiel wird in Abb. 3.5 gezeigt.

3.7.6 Slots

Für jeden öffentlichen Slot (`aSlot`) einer Java-Klasse erzeugt der Stubgenerator eine get-Methode (`"aSlot"`) und eine set-Methode (`"aSlot:="`). Diese senden dann einen “Slot Access Request” zur Java-Seite.

3.7.7 Exceptions

Die Klasse `JavaThrowable` (Stub von `java.lang.Throwable`) erbt von `J2TObject` und von `Exception`. Tritt bei der Bearbeitung eines Up-Calls[2]

[2] Aufruf einer Methode aufgrund eines Requests der jeweiligen Gegenseite

auf Java-Seite eine Exception auf, so wird diese an die Tycoon-Seite als Objekt übertragen. Das korrespondierende Tycoon-Objekt (ein Exemplar einer Subklasse von `Exception`) wird dann dort ausgelöst (raised).

3.8 Optimierungen

Da bei Java2Tycoon der Java-Client und der Tycoon-Server auf unterschiedlichen Rechnern ablaufen können und die Kommunikation zwischen den beiden Partnern möglicherweise über ein WAN mit geringer Bandbreite stattfindet, sollte die Anzahl der versendeten Kommunikationspakete möglichst gering gehalten werden. Dies wird im wesentlichen durch gepufferte Datenströme erreicht, die nur nach bestimmten Calls geleert (flushed) werden und deren Inhalt damit zur Gegenstelle gesendet wird.

Bezüglich der Optimierung werden zwei Grundtypen von Methodenaufrufen unterschieden:

1. **"round trip call":** Bei einem "round trip call" [Nye 90] muß der aufrufende Thread die Abarbeitung der Methode im Server abwarten, bevor die Programmausführung fortgesetzt wird; die Ausführung der Methode erfolgt also synchron. Ein "round trip call" führt somit immer zum Leeren des entsprechenden Kommunikationspuffers.

2. **"one way call":** Bei einem "one way call" erwartet der Stub keinen Rückgabewert von der Gegenseite und die Programmausführung kann somit gleich fortgesetzt werden; die letztliche Ausführung der Methode erfolgt also asynchron. Ein "one way call" führt i. d. R. nicht zum Leeren des entsprechenden Kommunikationspuffers.

Aufgrund ihrer Signaturen lassen sich die Methoden einer Java-Klasse wie folgt unterteilen:

- Void-wertige Methoden mit leerer `throws`-Klausel
- Konstruktoren mit leerer `throws`-Klausel
- Andere Methoden

3.8.1 Void-wertige Methoden mit leerer `throws`-Klausel

Dies sind jene Methoden einer Java-Klasse, deren Return-Typ `void` ist und deren `throws`-Klausel leer ist. Da der rufende Thread von dieser Methode keinen Rückgabewert erwartet und keine Exceptions ausgelöst werden, kann

dieser fortgesetzt werden, ohne die Ausführung bzw. Beendigung der gerufenen Methode abzuwarten.

Diese Methoden werden als “one way call” gerufen - die Kommunikationspuffer werden nach Plazierung des Aufrufs im Puffer nicht geleert.

3.8.2 Konstruktoren mit leerer `throws`-Klausel

Bei Java-Konstruktoren wird immer ein neues Objekt angelegt, dem zur Java2Tycoon-Kommunikation eine `uid` zugeteilt werden muß. Die Vergabe der `uid` geschieht normalerweise auf der Java-Seite, wenn das Objekt als Rückgabewert des Konstruktoraufrufs zur Tycoon-Seite übertragen werden soll.

Wird ein Konstruktor jedoch von Tycoon aus gerufen und ist seine `throws`-Klausel leer, so kann die `uid` des neuen Objektes bereits auf der Tycoon-Seite vergeben werden und dann im Aufruf an die Java-Seite weitergegeben werden. Der Rückgabewert der Konstruktormethode wird also bereits vom Aufrufer festgelegt.

Wie 3.8.1 werden auch diese Methoden als “one way call” gerufen - die Kommunikationspuffer werden nach Plazierung des Aufrufs im Puffer nicht geleert.

3.8.3 Andere Methoden

Alle Methoden, die nicht unter 3.8.1 oder 3.8.2 fallen, werden als “round trip call” gerufen - die Kommunikationspuffer werden also immer geleert und der aufrufende Thread muß die synchrone Abarbeitung der Methode abwarten.

3.9 Distributed Garbage Collection

Um die Benutzung so einfach wie möglich zu halten und um eine möglichst gute Integration in das Tycoon-System zu gewährleisten, sollte Java2Tycoon eine Distributed Garbage Collection (DGC) ausführen.

In [Birell et al. 93] wird ein allgemein verwendbarer Algorithmus zur Ausführung einer DGC vorgestellt. Dieser kann aufgrund des einfachen Aufbaus der Java2Tycoon-Kommunikation noch zusätzlich vereinfacht werden.

Die folgenden Vereinfachungen sind möglich:

- Da es sich bei Java2Tycoon lediglich um eine Kommunikation zwischen zwei Partnern handelt (Java-Seite, Tycoon-Seite), ist es möglich

jeglichen Datenaustausch zwischen den Partnern für die Dauer der DGC anzuhalten. Damit ist das Problem von *race conditions* erheblich entschärft.

- Referenzen auf entfernte Objekte sind nur im Kontext einer bestimmten Verbindung gültig. Ein "Weiterreichen" von Java-Objekten zwischen unterschiedlichen Verbindungen ist nicht erlaubt. Damit ist immer bekannt, von wo ein Objekt referenziert wird. Referenzen von unterschiedlichen Adressräumen (ein *dirty set*) müssen nicht verwaltet werden.

- Zwischen den Kommunikationspartnern wird eine sichere Verbindung vorausgesetzt. Ein Test, ob der jeweilige Partner abgestürzt ist, muß vom DGC nicht ausgeführt werden. Treten Störungen auf, so wird (analog zu einem X-Terminal) die Verbindung beendet und damit alle referenzierten Objekte freigegeben.

Der genannte Algorithmus erfordert allerdings *weak references*, die in der aktuellen Java-Version (JDK 1.1.5 vom Januar 1998) nicht vollständig zur Verfügung stehen. Ab der Version 1.2 wird Java über "echte" *weak references* verfügen, womit eine Implementation des geschilderten Algorithmus möglich wäre. Version 1.2 wurde für das 2. Quartal 98 angekündigt.

Kapitel 4

Implementierung

Im Rahmen der Arbeit wurde eine prototypische Implementierung erstellt, die zumindest teilweise dem beschriebenen Design entspricht. Dabei konnte teils aufgrund des gesteckten Zeitrahmens, teils wegen noch nicht zur Verfügungen stehender Features von Java oder Tycoon nicht der volle Funktionsumfang realisiert werden.

Im folgenden wird auf die einzelnen Komponenten der Implementierung eingegangen.

4.1 Transport Layer

Wie bereits eingangs beschrieben, basiert die aktuelle Implementierung der Transport Layer auf einer TCP-Socket-Kommunikation. Der Verbindungsaufbau findet durch den Client statt, der von einem Nameserver einen bestimmten Dienst anfordert.

Der Nameserver "horcht" auf einem festgelegten Port auf ankommende Anforderungen. "Möchte" ein Client einen Dienst anfordern, so baut er zunächst vier Socketverbindungen vom Client zum Nameserver auf. Durch jeden der Ströme wird als erstes Zeichen eine Kennung gesendet, die festlegt, als welcher der vier o. g. dieser in der folgenden Kommunikation zu verwenden ist. Es werden vier bidirektionale Ströme jeweils unidirektional verwendet, da in der aktuellen Implementierung der TCP-Sockets von Tycoon ein gleichzeitiges Lesen und Schreiben einer Socket nicht möglich ist.

Wurden die vier Streams aufgebaut, so erfragt der Java2Tycoon-Nameserver vom Client über die statische Methode `hox.j2t.Support.getRequestedService()` den gewünschten Dienst als `String`. Ein Dienst ist hierbei eine Funktion mit der Typisierung `:Fun1(MarshalLayer, Void)`. Eine solche Funktion kann beim Nameserver

per `bindService` an eine Service-Namen gebunden werden und steht somit den Clients zur Verfügung.

Das Starten des Services entspricht dann der Anwendung der aufgebauten Connection (bestehend aus den vier Einzelverbindungen) auf die registrierte Funktion. Hierfür wird ein neuer Thread gestartet, damit der Nameserver weitere Anforderungen bearbeiten kann.

Dieser Aufbau hat den Vorteil, daß Tycoon im Kommunikationsaufbau passiv ist, seine Dienste im Sinne eines Servers beliebigen Klienten im Netz auf Anforderung zur Verfügung stellt. Ein weiterer Vorteil ist der, daß die *Applet Security* [Java Security 97] (sofern WWW-Server und Tycoon-Strore auf dem selben Rechener ablaufen) nicht verletzt wird, da der Kommunikationsaufbau lediglich vom Client zum Server stattfindet, nicht jedoch umgekehrt.

4.2 Marshal Layer

4.2.1 Transfersyntax

Die Übertragung der Werte zwischen Java und Tycoon wird von den Klassen `J2TMarshalInput` und `J2TMarshalOutput` auf Tycoon-Seite bzw. `hox.j2t.MarshalInputStream` und `hox.j2t.MarshalOutputStream` auf Java-Seite übernommen. Diese Streams werden von der Marshal-Layer verwendet, um die Laufzeitwerte, entweder serialisiert oder per Referenz zur Gegenseite zu übertragen.

Das folgende Listing zeigt einen Teil der Schnittstelle der Klasse `J2TMarshalOutput`

```
1    (*
2    Decorates an output: can marshal Java base types
3    and Call-Objects
4    *)
5    class J2TMarshalOutput
6    super  OutputDecorator
7    metaclass J2TMarshalOutputClass
8
9    public methods
10
11   writeChar(v :Char)
12
13   writeReal(v :Real)
14
```

```
15   writeInt(v :Int)
16
17   writeLong(v :Long)
18
19   writeBool(v :Bool)
20
21   writeString(v :String)
22
23   writeGeneric(v :Object)
24     (* sends printOn-representation of an arbitrary object *)
25
26   writeObject(v :J2TStub)
27
28   writeNil()
29
30   writeAny(v :Object)
31   (*
32     Detects type, sends typecode and calls apropriate write-Method
33   *)
```

Die Streams bieten die Möglichkeit an, einen Wert mit oder ohne Typecode zu versenden. Ein Typecode gibt an, von welchem Transfertyp ein Wert ist. Die Methoden `write`*Basistyp* senden nur die serielle Repräsentation des betreffenden Werts, die Methode `writeAny` ermittelt aufgrund des Laufzeittyps den Transfertyp des zu übetragenden Objekts, sendet den entsprechenden Typecode und ruft dann die zugehörige `write`*Basistyp*-Methode auf.

Tabelle 4.2.1 zeigt alle Transfertypen mit ihren Typecodes (ein Typecode wird als `Char` übertragen). Der Transfertyp `ANY` hat keine eigenen Typecode; er stellt lediglich einen Wert dar, der seinen Typecode mit sich führt.

Zum Begriff des Transfertyps siehe Abschnitt 3.4.1. Die Spalte **Anzahl Transferbytes** gibt die Anzahl der für die Transferrepräsentation verwendeten Bytes an.

Die genaue Transferdarstellung der einzelnen Datentypen geht aus Tabelle 4.2 hervor. Dabei steht `byte[0]` für das erste Byte und `byte[n]` für das letzte Byte der Transferdarstellung .

4.2.2 Kommunikationsprotokoll

Die Nachrichten, die zwischen den Kommunikationspartnern versendet werden, gliedern sich in:

Transfertyp	Typecode	Anzahl Transferbytes
CHAR	'c'	1
REAL	'r'	?
INT	'i'	4
LONG	'l'	8
BOOL	'b'	1
STRING	's'	variabel
OBJECT	'o'	4
ANY		

Tabelle 4.1: Typecodes beim Datenaustausch zwischen Java und Tycoon

- **Requests:** Dies sind Anforderungen der Clientseite (Client ist der Verwender eines Stubs) an die Serverseite. Momentan existieren vier Request-Typen:

  1. Virtual Call Request: Aufruf einer Objektmethode
  2. Static Call Request: Aufruf einer Java-Klassenmethode (`static`)
  3. Array Access Request: Lesen oder Setzen eines Arrayelements (diese Request-Art wird von der momentanen Implementierung nicht unterstützt)
  4. Slot Access Request: Lesen oder Setzen eines Objektslots (diese Request-Art wird von der momentanen Implementierung nicht unterstützt)

- **Replies:** Dies sind Antworten auf die o. g. Requests. Abhängig von der eingesetzten Optimierung kann ein Request ohne Reply bleiben. Momentan existieren zwei Reply-Typen:

  1. Sucess Reply: Aufruf verlief erfolgreich. Die Nachricht enthält den Rückgabewert.
  2. Exception Reply: Während des Aufrufs wurde eine Exception ausgelöst. Die Nachricht enthält das Exception-Objekt (diese Reply-Art wird von der momentanen Implementierung nicht unterstützt).

Im Folgenden wir der Aufbau der einzelnen Nachrichten beschrieben.

Virtual Call Request

Ein Virtual Call Request stellt den Aufruf einer Objektmethode dar. Er darf über `CallStream` und `CallbackStream` versendet werden. Außer den Daten, die den Aufruf selbst betreffen, und somit an die Skeleton Layer weitergereicht werden, trägt ein Virtual Call Request einige Verwaltungsdaten. Die Thread-ID bestimmt die Ausfühungsqueue, in die der Request eingereiht werden soll. Außerdem wird über die Thread-ID eine Reply einem Request zugeordnet. Das Feld Optimierungen ist ein Integer-Wert, der angibt, wie der Request zu optimieren ist.

Der genaue Aufbau eines Virtual Call Requests ist in Tabelle 4.3 beschrieben.

Static Call Request

Ein Static Call Request ist wie ein Virtual Call Request aufgebaut. Statt SelfRef wird hier allerdings der Name der Klasse gesendet, in deren Kontext die Methode aufgerufen wird.

Der genaue Aufbau eines Static Call Requests ist in Tabelle 4.4 beschrieben.

Sucess Reply

Die Sucess Reply enthält den Rückgabewert eines Methodenaufrufs. Sie werden über `ReplyStream` und `CallbackReplyStream` versendet. Ist der Rückgabetyp einer Methode `void` (bzw. `Void`), so wird `null` bzw. `nil` als Wert versendet.

Der Aufbau der Sucess Reply ist in Tabelle 4.5 beschrieben.

4.3 Stub/Skeleton Layer

Die Skeleton Layer wurde generisch implementiert, d.h., daß im Gegensatz zum RMI-System [Sun 97] nicht für jede verwendete Klasse eine eigene Skeleton-Klasse erstellt wird, die dann aufgrund des Selektors im generischen Methodenaufruf an die Skeleton Layer (siehe Abschnitt 3.2.1) die entsprechende Methode aufruft.

In Java2Tycoon wird mit Hilfe der reflexiven Möglichkeiten von Java und Tycoon ohne die Verwendung von Skeletons die gewünschte Methode aufgerufen. In Tycoon geschieht dies mittels der Methode `Object::perform`, die die folgende Signatur hat:

```
(* perform method ``selector'' on receiver with
   arguments ``args'' and return the result *)

perform(selector :Symbol, args :Array(Object)) :Nil
```

In Java wird eine ähnliche Methode mit Hilfe des `java.lang.reflect`-Packages implementiert.

Die generische Realisierung der Skeletons erspart die Erzeugung einer Skeleton-Klasse für jede verwendete Anwendungsklasse.

Ein weiterer Schritt, um die Codereplikation gering zu halten, wäre die Verwendung von generischen Stubs auf Tycoon-Seite. Hierbei wird für jede verwendete Klasse vom Stubgenerator eine abstrakte Interface-Klasse erstellt, die also lediglich die Schnittstelle der entfernten Objektes abbildet. Das eigentliche Stub-Objekt ist dann eine Instanz einer parametrisierten Klasse `Stub(T)`. Wird eine Nachricht an ein Objekt dieser Klasse gesandt, für die keine Methode definiert wurde, so ruft die Tycoon-VM die Methode `_doesNotUnderstand` mit der folgenden Signatur auf:

```
_doesNotUnderstand(selector :Symbol, args :Array(Object)) :Object
```

Diese Methode wird in der Klasse `Stub` von einer Methode überschrieben, die den Selektor und die Argumente in einem *Virtual Call Request* zur Gegenseite sendet und dann ggf. den Inhalt der *Reply* als Ergebnis des Methodenaufrufes zurückgibt.

Um einem Objekt vom Typ `Stub(T)` alle Nachrichten des abstrakten Interfaces `T` senden zu können, und dabei weiterhin typkorrekte Programme zu schreiben, muß für den Tycoon-Typechecker folgende Subtypbeziehung gelten:

```
Stub(T) <: T
```

Die Implementierung dieser Regel im Tycoon-Typecheker ist aber noch nicht erfolgt, womit eine Verwendung der generischen Stubs noch nicht möglich ist. Die aktuelle Implementierung verwendet deshalb statische Stubs.

Da die Java-VM über keinen `_doesNotUnderstand`-Mechanismus verfügt, ist die Verwendung von generischen Stubs auf der Java-Seite ausgeschlossen.

Transfertyp	Position	Inhalt
CHAR	byte[0]	wird ohne jegliche Konvertierung übertragen
REAL		wird in der aktuellen Implementierung nicht unterstützt
INT	byte[0] byte[1] byte[2] byte[3]	MSB LSB
LONG	byte[0] byte[1] byte[2] byte[3] byte[4] byte[5] byte[6] byte[7]	MSB LSB
BOOL	byte[0]	x01 = true, x00 = false
STRING	byte[0] byte[1] byte[2] byte[3] byte[4] byte[.] byte[.] byte[n+3]	byte[0-3] enthalten die Länge n des Strings als INT. byte[4-(n+3)] enthalten die n Bytes des Strings als CHAR
OBJECT	byte[0] byte[.] byte[.] byte[x-1] byte[x] byte[x+1] byte[x+2] byte[x+3]	der Tycoon-Name der Klasse als STRING byte[x - x+3] enthalten die uid des Objektes als INT. Für Nil bzw. null wird der Wert 0 versendet.
ANY	byte[0] byte[1] byte[.] byte[.] byte[n]	der Typecode des betreffenden Transfertyps Darstellung des betreffenden Transfertyps

Tabelle 4.2: Darstellung der Transferdatentypen

Feld	**Transfertyp**	**Inhalt**
Requestkennung	`CHAR`	`'v'`
Selektor	`STRING`	Selektor des Aufrufs
ParamCount	`INT`	Zahl der Parameter
Param 1 .. Param n	`ANY` .. `ANY`	1. Parameter .. n. Parameter
Optimierungen	`INT`	Anzuwendende Optimierungen codiert als Bitmaske: 1. Bit: *one way call,* 2. Bit: Konstruktor-Optimierung
SelfRef	`OBJECT`	Objekt an dem der Aufruf auszuführen ist
Thread-ID	`INT`	
ReturnUID	`INT`	nur bei Konstuktor-Optimierung: UID des Returnwertes

Tabelle 4.3: Aufbau eines Virtual Call Requests

Feld	Transfertyp	Inhalt
Requestkennung	`CHAR`	`'s'`
Selektor	`STRING`	Selektor des Aufrufs
ParamCount	`INT`	Zahl der Parameter
Param 1 .. Param n	`ANY` .. `ANY`	1. Parameter .. n. Parameter
Optimierungen	`INT`	Anzuwendende Optimierungen codiert als Bitmaske: 1. Bit: *one way call*, 2. Bit: Konstruktor-Optimierung
ClassName	`STRING`	Name der Klasse, in deren Kontext der Aufruf auszuführen ist
Thread-ID	`INT`	
ReturnUID	`INT`	nur bei Konstuktor-Optimierung: UID des Returnwertes

Tabelle 4.4: Aufbau eines Static Call Requests

Feld	Transfertyp	Inhalt
Thread-ID	`INT`	Kennung des aufrufenden Threads
RetVal	`ANY`	Rückgabewert

Tabelle 4.5: Aufbau einer Sucess Reply

Kapitel 5

Zusammenfassung und Ausblick

Die vorliegende Arbeit beschreibt Analyse, Design und Implementierung einer Objekt-Kommunikation zwischen Java und Tycoon. Dabei wurde besonderer Wert auf die Möglichkeit der Verwendung der Java-GUI-Dienste von Tycoon aus gelegt.

Diese Entscheidung hatte u. a. die folgenden Auswirkungen:

1. Es entstand eine spezialisierte Form der Objektkommunikation zwischen zwei Kommunikationspartnern, die besonders GUI-Anwendungen im Inter- und Intranet unterstützt. Durch die verwendeten Optimierungsmethoden können hiermit deutlich mehr Calls pro Sekunde abgewickelt werden als z. B. mit Java-RMI oder auch Corba[1].

2. Die Kommunikation ist sehr einseitig angelegt. Im Gegensatz zu einer symmetrischen Kommunikation nimmt hier Tycoon lediglich die "Dienste" von Java in Anspruch.

Eine Ergänzung der Arbeit bestünde im Design eines Stubgenerators. Hierbei muß das Overloading von Java aufgelöst und Stub- sowie Callback-Klassen erstellt werden. Bei der Erzeugung von generischen Stub-Interfaces (s. Abschnitt 4.3) müßte ein anderer Mechanismus gefunden werden, die zu verwendende Optimierungsart an die aufgerufene Methode zu binden, da die Methode selbst, als Träger dieser Information, im generischen Stub nicht mehr vorhanden ist. Hier könnte ein Interface-Repository, wie es von Corba verwendet wird, zum Einsatz kommen.

[1] Corba unterstützt zwar asynchrone Requests, garantiert jedoch nicht ihre sequentielle Ausführung. Eine Konstruktor-Optimierung kann aufgrund des Fehlens von Konstruktoren in der IDL nicht vorgenommen werden.

Weiterhin ist die Frage interessant, inwieweit sich die verwendeten Optimierungen auf andere Objekt-Kommunikationen anwenden läßt, die nicht den o. g. Einschränkungen unterliegen.

Außerdem lassen sich nicht alle Anwendungsfälle befriedigend mit der Versendung von Basistypen und Objektreferenzen lösen. Hier wäre ein generisches Marshalling von beliebigen Objekten, wie es im Java-RMI-System enthalten ist, wünschenswert. Dabei werden jedoch schnell die Grenzen einer sinnvollen Abbildung zweier unterschiedlicher Objektmodelle aufeinander erreicht. Corba definiert daher die IDL und gibt somit ein verhältnismäßig einfaches (stark an C++ angelehntes) Objektmodell vor, das allerdings nur das Versenden von methodenlosen Strukturen, nicht aber von Objekten erlaubt[2]. Eine allgemeingültige Objekt-Kommunikation steht also noch aus.

[2] Die OMG arbeitet aber bereits an einem Standard zum Versenden von Objekten (es wurde ein *request for proposal* mit dem Titel *"Objects-by-value"* ausgeschrieben).

Literaturverzeichnis

Birell et al. 93: Birell, A., Evers, D., Nelson, G., Owicki, S., and Wobber, E. *Distributed Garbage Collection for Network Objects*. Technical Report 116, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, Dezember 1993.

Birell et al. 95: Birell, A., Nelson, G., Owicki, S., and Wobber, E. *Network Objects*. Technical Report 115, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, Dezember 1995.

Gamma et al. 95: Gamma, E., Helm, R., Johnson, R., and Vlissades, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

Gosling et al. 96: Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. Addison-Wesley Publishing Company, 1996.

Java Security 97: Frequently Asked Questions - Java Security . http://java.sun.com/sfaq/, 1997. Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, USA.

Krasner, Pope 88: Krasner, G. and Pope, S. *A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. Journal of Object-Oriented Programming, Jg. 1, 1988, Nr. 3.

Lockemann, Schmidt 87: Lockemann, P.C. and Schmidt, J.W. (Eds.). *Datenbank-Handbuch*. Springer-Verlag, 1987.

Nye 90: Nye, A. *X Protocol Reference Manual*. O'Reilly & Associates, Inc., 1990.

Obj 96: Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, Juli 1996.

Sun 97: Sun Microsystems. *Java Remote Method Invocation Specification*, 1.4 edition, Februar 1997.

Wienberg 96: Wienberg, A. *Bootstrap einer persistenten objektorientierten Programmierumgebung*. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, August 1996.