Verteilte Implementierung der objektorientierten Programmiersprache SELF

Kai Knubben

17. Dezember 2000

Diese Diplomarbeit entstand an der Technischen Universität Berlin im Fachbereich Informatik, Fachgebiet Formale Modelle, Logik und Programmierung, das von Herrn Prof. Mahr geleitet wird. Betreut wurde diese Arbeit von Herrn Dr. Robert Tolksdorf.

Inhaltsverzeichnis

Kapitel 1

Einführung und Motivation

Diese Diplomarbeit beschreibt die Programmiersprache dSelf, die einen verteilten Derivat zu SELF darstellt. SELF ist eine klassenlose, objektorientierte Programmiersprache, die zwischen 1987 und 1995 an der Stanford University in Zusammenarbeit mit Sun Microsystems Inc. entstand. Im Gegensatz zu klassenbasierten Sprachen, wie zum Beispiel Java oder Smalltalk, gibt es in SELF nur Objekte. Diese Objekte bestehen aus Slots, die Methoden- und Datenobjekte beinhalten können, und repräsentiern die Variablen und Methoden von klassenbasierten Sprachen. Diese Slots können zur Laufzeit durch Anwendung von primitiven Methoden beliebig manipuliert werden, indem man zum Beispiel vorhandene Slots löscht oder neue zum Objekt hinzufügt. Um Instanzen von Objekten zu bilden kopiert man sogenannte Prototypobjekte, indem man sie klont. Die Vererbung in SELF wird über sogenannte Parentslots realisiert, in denen sich Referenzen auf andere Objekte befinden. Diese Parentslots werden vom eingebauten Vererbungsalgorithmus benutzt, indem ein Objekt Nachrichten, für die es keinen passenden Slot besitzt, an seine Parents delegiert. Diese Parentslots können ebenfalls zur Laufzeit verändert werden, wodurch eine dynamische Vererbung realisiert wird. Da diese Referenz die einzige Verbindung zwischen einem erbenden Objekt und seinem Parent ist, besitzen Objekte in SELF einen hohen Grad an Autonomie.

In klassenbasierten Sprachen hingegen unterscheidet man zwischen einer Klasse und einem Objekt als seine Instanz. Die Klasse definiert das Verhalten für alle seine Instanzen und das Objekt den Zustand. Diese Trennung zwischen Verhalten und Zustand bedeutet eine starke Einschränkung in einer verteilten Umgebung, die am Beispiel Java verdeutlicht werden soll.

In Java gibt es bereits eine eingebaute Unterstützung für verteilte Systeme in Form von Java-RMI. RMI ermöglicht es, über die Grenzen einer Java Virtual Machine hinaus die Methoden eines entfernten Objekts aufzurufen. Dabei werden die Argumente dieses Methodenaufrufs serialisiert und im Kontext der entfernten VM wieder hergestellt. Um diese Argumentobjekte wiederherzustellen, muß man auch das Verhalten dieser Objekte, also ihre Klassen, zur Verfügung stellen, indem ihre Definitionen samt aller Oberklassen mitkopiert werden. Dabei wird auch all das Verhalten mitkopiert, welches auf der Gegenseite gar nicht benötigt wird, was zu einem hohen Aufwand führt.

Dieser Aufwand läßt sich mit einer klassenlosen Sprache wie SELF vermeiden, da hier ein Objekt Zustand und Verhalten in sich vereinen kann und zudem noch einen hohen Grad an Autonomie besitzt. Um diese Vorteile zu nutzen, wurde dSelf entwickelt. Es ermöglicht eine verteilte Vererbung über die Grenzen einer VM hinaus, indem es dem Benutzer zum Beispiel erlaubt ein lokales Objekt zu erzeugen, das von einem Objekt einer anderen VM erbt. Dadurch kann kann es durch den Vererbungsmechanismus von dem entfernten Objekt Verhalten in Form einer Methode anfordern, wobei nur ihr Code übertragen werden muß. Dies alles geschieht in einer für den Benutzer transparenten Art und Weise, indem er lediglich über einen einfachen Aufruf einer primitiven Methode eine Verbindung zu einer anderen dSelf Virtual Machine aufbaut. Von nun an kann er auf alle verteilten Objekte zugreifen, als seinen sie lokal vorhanden, da jegliche Verwaltung von dSelf übernommen wird.

In dieser Arbeit wird zuerst die Sprache SELF vorgestellt und im Anschluß daran ihr die Sprache dSelf gegenübergestellt, wobei die jeweiligen Unterschiede beschrieben und ihre Ursachen begründet werden. Danach erfolgt eine Beschreibung der konkreten Implementierung und ein abschließender Blick auf mögliche Erweiterungen für diese.

Kapitel 2

Einführung in SELF

Im folgenden Text wird die Sprache SELF beschrieben. Es wird versucht einen allgemeinen Überblick über die Sprache und ihre wesentlichen Bestandteile zu geben, ohne jedoch Details wie zum Beispiel alle zur Verfügung stehenden Escapesequenzen in Strings zu erwähnen. Solche Informationen finden sich in der Programmierreferenz [1], sowie auch eine formale Beschreibung des Vererbungsmechanismus.

2.1 Was ist SELF ?

SELF ist eine objektorientierte Programmiersprache, die im Zeitraum zwischen 1987 und 1995 an der Stanford University in Zusammenarbeit mit Sun Microsystems Inc. entstand. Ihre auffälligste Eigenschaft ist die Klassenlosigkeit, d.h. anstatt von Klassen und Objekten als deren Instanzen gibt es in SELF Objekte, die als Prototypen fungieren und die durch klonen, einer speziellen Art von kopieren (*shallow copy*), instanziert werden. Eine weitere Spezialität dieser Sprache sind die Slots, aus denen die Objekte bestehen. Sie beinhalten entweder Daten oder Methodenobjekte und sind zur Laufzeit mit entsprechenden primitiven Methoden (*_AddSlots:*, *_RemoveSlots:*, etc.) modifizierbar. Eine spezielle Art von Slots bilden die Parentslots, die die Vererbung in SELF realisieren, indem sie eine nichtverstandene Nachricht an die Parentslots weiterreichen. Dieses Verhalten wird delegieren genannt und wenn die Parentslots zur Laufzeit modifiziert werden, nennt man dies dynamische Vererbung.

2.2 Daten- und Methodenobjekte

Während man es in herkömmlichen objektorientierten Programmiersprachen gewohnt ist, daß es Klassen und als deren Instanzen die Objekte gibt, existieren in SELF nur Objekte. Man unterscheidet zwischen Daten- und Methodenobjekten. Datenobjekte bestehen aus einer Reihe von Slots, die durch Punkte seperiert werden. Also zum Beispiel:

```
(| a=1. b=2. c. |)
```

Hier werden drei Datenslots deklariert mit den Namen *a*, *b* und *c*, die jeweils die primitiven Datenobjekte 1, 2 und *nil* beinhalten. Die vertikalen Striche markieren den Anfang und das Ende einer Slotliste und die runden Klammern deklarieren diesen Ausdruck als ein SELF-Objekt. Methodenobjekte besitzen einen ähnlichen Aufbau, besitzen aber zusätzlich noch Code, der hinter der Slotliste angefügt wird. Also zum Beispiel:

```
(| a=1. b=2. c. | c: self a + self b. ^self c )
```

Hier wird die Summe von *a* und *b* dem Slot *c* zugewiesen und anschließend sein Inhalt mittels dem Operator "^" zurückgegeben. Der Punkt ist, wie zuvor bei den Slots, ein Seperator, der die einzelnen Ausdrücke voneinander trennt. Das reservierte Wort *self* bewirkt, daß die Nachrichten *a*, *b* oder *c* an das Objekt selbst geschickt und lokal ausgewertet werden. Da in SELF jede Berechnung durch das Versenden von Nachrichten realisiert wird, wovon viele an das Objekt selber geschickt werden, kann dies zu sehr langen Ausdrücken führen. Deshalb ist das *self*, wie einige andere sprachlichen Konstrukte optional. Darum ist folgende Notation semantisch äquivalent zur obigen:

```
(| a=1. b=2. c | c: a + b. c)
```

Desweiteren ist der letzte Punkt in einer Slotliste optional, wie auch das Retursymbol "^", da der letzte Ausdruck im Code immer als der Rückgabewert betrachtet wird. Wie Methodenobjekte oder die ihnen ähnlichen Blockobjekte in SELF im Detail behandelt werden, werden in einem späteren Abschnitt noch genauer Beschrieben. Um sie zu verstehen, ist es zunächst notwendig zu verstehen, wie die Slots im genauen aussehen.

2.3 Slots

Bisher wurde gesagt, daß Objekte aus Slots bestehen, doch es stellt sich die Frage, was Slots eigentlich genau sind. Slots (deutsch: Schlitz) stellen im Prinzip Behälter mit einem Namen dar, die entweder Daten- oder Methodenobjekte aufnehmen können.

2.3.1 Datenslots

Datenslots können in den Slotlisten auf drei verschiedene Weisen deklariert werden:

1. x = Datenobjekt.

2. x <- Datenobjekt.

3. x.

In der ersten Form wird ein Slot x deklariert, der das Datenobjekt, welches auf der rechten Seite vom "="-Operator erscheint, beinhaltet. Durch das "=" wird angegeben, daß es sich um einen nur lesbaren Slot handelt, der nicht überschrieben werden kann.

Die zweite Form weist dem x ebenfalls ein Datenobjekt zu, jedoch zeigt der Pfeil an, daß dieser Slot auch beschreibbar ist. Dazu wird zusätzlich ein Zuweisungsslot (*assignment slot*) hinzugefügt, der das überschreiben des alten Slotinhalts möglich macht. Also zum Beispiel:

```
(| x<- 1 |)
```

führt zu zwei neuen Slots, nämlich:

```
x = 1
```

und

```
x: <-
```

Die dritte Form deklariert einen Slot x mit dem Wert nil, da nicht initialisierte Slots diesen Wert immer zugewiesen bekommen, was verdeutlichen soll, daß es sich bei dem gegenwärtigen Inhalt um kein brauchbares Objekt handelt. Den Inhalt dieser Slots erhält man, indem man dem Objekt, das diese Slots besitzt, eine Nachricht sendet, die mit dem Namen des Slots übereinstimmt. Als Antwort wird sein Inhalt evaluiert und zurückgeschickt. Wenn es sich um einen überschreibbaren Slot handelt, kann durch ein angehängtes Doppelpunktzeichen an den Namen (*assignment primitive*) ein neues Objekt ihm zugewiesen werden. Es ist hier zu beachten, daß es in SELF im allgemeinen keine Typisierung gibt, es wird also nicht geprüft, ob zum Beispiel ein mit einem String initialisierter Slot mit einem Integer überschrieben wird. Mit "im allgemeinen" ist hier eine eingeschränkte Typisierung bei primitiven Methoden gemeint, die später im Detail noch besprochen werden.

2.3.2 Methodenslots

Methodenslots beinhalten Methodenobjekte und können in drei verschiedenen Formen deklariert werden:

1. unäre Methodenslots

2. binäre Methodenslots

3. keyword Methodenslots

2.3.2.1 Unäre Methodenslots

Unäre Methodenslots bestehen nur aus einem Namen (*identifier*), der mit einem kleinen Buchstaben oder Unterstrich beginnt und mit einer beliebigen Reihenfolge von kleinen oder großen Buchstaben, Zahlen oder Unterstrichen vervollständigt wird. Selbstdefinierte Name dürfen nicht mit einem Unterstrich beginnen, da es sich hierbei um primitive Methoden handelt, die von SELF vorgegeben sind. Einige Beispiele für unäre Methoden:

```
(| print = (_Print).
   succ = (self + 1).
   pred = (self - 1)
|)
```

Unäre Methoden werden immer von links nach rechts ausgewertet, also ein Ausdruck wie zum Beispiel:

```
x arg1 arg2 arg3
```

würde in folgender Reihenfolge ausgewertet werden:

```
((x arg1) arg2) arg3
```

Die Präzedenz ist größer als die von binären oder keyword-Methoden.

2.3.2.2 Binäre Methodenslots

Binäre Methodenslots bestehen aus einem Operator und einem Argument. Operatoren sind eine beliebige Reihenfolge von Operatorsymbolen. Folgende Symbole stehen in SELF dafür zur Verfügung:

! @ # $ % ^ & * - + = ~ / ? < > , ; | ' \

"^" und "|" dürfen allerdings nicht als alleinige Symbole benutzt werden, sondern müssen in Kombination mit anderen Operatorsymbolen auftreten, also zum Beispiel:

|>, ||, ^+, ^^, ^|

Zusätzlich muß ein Argument angegeben werden, wobei es dem Benutzer freigestellt ist, ob er den Argumentnamen direkt hinter den Operator schreibt oder stattdessen in der Slotliste aufführt. Die folgenden beiden Schreibweisen bedeuten also dasselbe:

```
(| + arg = (^self _IntAdd: arg) |)
(| + = (| :arg| ^self _IntAdd: arg) |)
```

In beiden Fällen wird ein Argument (ein Integer) übernommen und dieses mittels der primitiven Methode *_IndAdd:* auf *self* (ebenfalls ein Integer) hinzuaddiert. In der ersten Schreibweise wurde darauf verzichtet eine leere Slotliste in der Form von zwei vertikalen Strichen darzustellen, da leere Slotlisten in SELF optional sind, also nicht explizit angegeben werden müssen. Binäre Methoden mit gleichen Operatorsymbolen sind linksassoziativ, während unterschiedliche Operatorsymbole ohne Klammerung nicht nebeneinander auftreten dürfen. Der Ausdruck

```
1 + 2 + 3 + 4
```

wird zum Beispiel in folgender Reihenfolge ausgewertet:

```
((1 + 2) + 3) + 4
```

Ein Ausdruck wie zum Beispiel:

```
1 + 2 * 3
```

kann gar nicht ausgewertet werden, da er nicht erlaubt ist. Die Präzedenz ist größer als die von *keyword*-Methoden, aber kleiner als die Präzedenz von unären Methoden.

2.3.2.3 *keyword*-Methodenslots

keyword-Methodenslots beginnen immer mit einem großgeschriebenen Schlüsselwort (*keyword*) gefolgt von einem Argument und einer Sequenz von Paaren von kleingeschriebenen Schlüsselworten und Argumenten. Schlüsselwörter beginnen immer mit einem großgeschriebenen (*capital keyword*) oder einem kleingschriebenen Buchstaben (*small keyword*), gefolgt von einer Sequenz von Buchstaben, Zahlen oder Unterstrichen und enden mit einem Doppelpunkt. Wie auch bei unären Methodenslots gibt ein vorangesetzter Unterstrich an, daß es sich um eine primitive Methode handelt. Genau wie bei den binären Methodenslots gibt es auch hier zwei verschiedene Formen, die semantisch äquivalent sind:

```
(| ifTrue: False: = (| :trueBl. :falseBl| trueBl value) |)
(| ifTrue: trueBl False: falseBl = (trueBl value) |)
```

Keyword-Methoden sind rechtsassoziativ, d.h. ein Ausdruck wie

```
5 min: 6 min: 7 Max: 8 Max: 9 min: 10 Max: 11
```

wird so interpretiert:

```
5 min: (6 min: 7 Max: 8 Max: (9 min: 10 Max: 11))
```

Die Präzedenz ist kleiner als die von binären oder *keyword*-Methoden.

Es ist noch darauf hinzuweisen, daß durch *keyword*-Methoden Zuweisungsslots für Datenobjekte überschrieben werden können und umgekehrt. Ein Beispiel dafür ist:

```
(| x <- 'Datenobjekt' |) _AddSlots: (| x: arg = ( ^ arg) |)
```

In diesem Beispiel wird zunächst ein wiederbeschreibbarer Datenslot x erzeugt. Dadurch hat dieses zwei Slots, nämlich x um auf seinen Inhalt zuzugreifen und *x:* um ihn einen Wert zuzuweisen. Durch die anschließende primitive Nachricht *_AddSlots:* wird ein Methodenslot mit dem Namen *x:* hinzugefügt, was zur Konsequenz hat, daß der Zuweisungsslot überschrieben wird. Das resultierende Objekt hat also folgende Form:

```
(| x = 'Datenobjekt' . x: arg = ( ^ arg) |)
```

Wenn man hingegen zuerst eine Methode deklariert und anschließend einen zuweisbaren Datenslot hinzufügt wird die Methode nicht gelöscht, sondern durch den Zuweisungsslot des Datenobjekts verborgen. Daher führt folgendes Beispiel:

```
(| x: arg = ( ^ arg) |) _AddSlots: (| x <- 'Datenobjekt' |)
```

zu folgendem Ergebnis:

```
(| x<- 'Datenobjekt' . x: arg = ( ^ arg) |)
```

Der Methodenslot bleibt also weiterhin erhalten, jedoch ist ein Aufruf nicht möglich, da bei einem Aufruf wie *x: <argument>* der Zuweisungsslot des Datenobjekts aufgerufen wird. Um die Methode wieder benutzen zu können, muß man den Zuweisungsslot des Datenobjekts wieder entfernen, wie in diesem Beispiel:

```
(| x<- 'Datenobjekt' . x: arg = ( ^ arg) |) _RemoveSlot: 'x:'
```

Wenn man statt *_AddSlots:* die primitive Nachricht *_AddSlotsIfAbsent:* verwendet kommt man zu folgenden Ergebnissen:

```
(| x <- 'Datenobjekt' |) _AddSlotsIfAbsent: (| x: arg = ( ^ arg) |)
```

führt zu

```
(| x <- 'Datenobjekt' |)
```

und

```
(| x: arg = ( ^ arg) |) _AddSlotsIfAbsent: (| x <- 'Datenobjekt' |)
```

führt zu

```
(| x: arg <- ( ^ arg) . x = 'Datenobjekt' |)
```

Wie man sieht wird bei *_AddSlotsIfAbsent:* der Zuweisungsslot des Datenobjekts trotz der gleichnamigen Methode hinzugefügt. Die Methode bleibt wie bei *_AddSlots:* so lange verborgen, bis der Zuweisungsslot wieder entfernt wird.

2.3.3 Parentslots

Parentslots sind im Prinzip identisch zu den unären Slots,wird aber zusätzlich durch einen Sternsuffix deklariert. Dieses angehängte Sternsymbol ist kein Bestandteil des Slotnamens, sondern eine Anweisung an SELF, diesen Slot für den internen Vererbungsmechanismus zu verwenden. Einige Beispiele für Parentslots sind:

```
(| lobby* = lobby.
   super* = (|z|).
   parent* = 0.0
|)
```

Wie aus dem Beispiel ersichlich ist, gibt es in SELF Mehrfachvererbung. Wenn ein Objekt eine Nachricht erhält, versucht es zunächst einen gleichnamigen Slot in seiner slotliste zu finden. Sollte es keinen passenden Eintrag finden, sendet es an alle Parentslots die Nachricht weiter. Sollte bei diesen Objekten ebenfalls kein passender Eintrag gefunden werden, so wird eine Fehlermeldung ausgegeben. Wenn hingegen mehr als ein Eintrag gefunden wird, ist nicht klar, welcher Slot gemeint ist und es kommt ebenfalls zu einer Fehlermeldung. Wird genau ein passender Slot gefunden, wird sein Inhalt zu dem Objekt kopiert, das die Nachricht ursprünglich erhalten hat und dort evaluiert.

2.3.4 Anmerkungen für Slots

Durch Anmerkungen an den Slots (*annotations*) können in SELF zusätzliche Informationen an die Programmierumgebung weitergegeben werden. Dabei können entweder ein ganzes Objekt oder nur eine Teilmenge aller Slots dieses Objektes eine Anmerkung erhalten. Ein Menge von Slots läßt sich durch eine einleitende, geschweifte Klammer, gefolgt von einem String und einer abschließenden, geschweiften Klammer eingrenzen. Also zum Beispiel in folgender Form:

```
(|
  {'Category: ...'
     slot1 = (...).
```

```
      slot2 = (...)
    }
  |)
```

Um eine Anmerkung für ein ganzes Objekt hinzuzufügen, kann man zwei geschweiften Klammern einen String zuweisen, wie in folgendem Beispiel:

```
(|
  {} = 'An annotation for this object'
  slot1.
  slot2
|)
```

Anmerkungen können auch geschachtelt werden. In diesem Fall werden sie in der Reihenfolge, in der sie auftreten zusammengefügt.

2.4 Methoden- und Blockobjekte

Nachdem Methoden bisher nur oberflächlich beschrieben wurden, soll jetzt genauer darauf eingegangen werden, was Methodenobjekte eigentlich sind und wie sie aktiviert werden.

2.4.1 Blöcke

Blöcke werden durch eine eckige Klammer ("[") eingeleitet und umfassen alle Ausdrücke bis zu einer abschließenden eckigen Klammer ("]"). Blöcke sind in SELF als *Closures* implementiert worden, das heißt, sie stellen eine Schnittstelle zum umgebenden Kontext zur Verfügung. Ein Block besteht immer aus einem *block method object* und einem *block data object*.

Das *block method object* beinhaltet den Code und das ihn umgebende block data object besitzt einen Parentslot zu *traits block*, von dem es sein Standardverhalten erbt. Um auf Slots aus dem umgebenden Kontext zugreifen zu können, besitzt das *block method object* einen anonymen Parentslot zum *activation object* der Methode oder des Blockes, in dem es eingebettet wurde. Anonym bedeutet, daß dieser Slot keinen Namen besitzt und somit auch nicht explizit angesprochen werden kann.

Um einen Block zu aktivieren, wird ihm eine Nachricht namens *value* geschickt. Wenn zusätzlich ein Argument übergeben werden soll, kann dies durch eine Nachricht namens *value:* zum Ausdruck gebracht werden. Bei mehreren Argumenten werden diese durch anschließende *With:* angegeben. Ein Block mit drei Argumenten läßt sich zum Beispiel so aktivieren:

```
[|:arg1. :arg2. :arg3| ... ] value: 1 With: 2 With: 3
```

Die Anzahl der Argumente, die dem Block übergeben werden, muß größer oder gleich der Anzahl der Argumentslots sein. Wenn weniger Argumentslots deklariert werden, als Argumente übergeben werden, werden die restlichen Argumente ignoriert. Die Argumentslots werden, wie bei den Methoden, durch einen anführenden Doppelpunkt deklariert und bei der Aktivierung durch die Argumentobjekte initialisiert.

Die Blockevaluierung besteht aus zwei Phasen. In der ersten Phase wird der Block geklont und sein Parentslot auf den umgebenden Kontext gesetzt. In der zweiten Phase wird das *block method object* geklont und seine Argumentslots werden aufgefüllt, sein anonymer Parentslot auf den umgebenden Kontext gesetzt, der in der ersten Phase ermittelt wurde und zum Schluß dann der Code ausgeführt. Der Rückgabewert eines Blockes wird durch das Returnsymbol "^" bestimmt, oder durch den letzten Ausdruck im Block, da das Circumflex optional ist.

2.4.2 Methodenobjekte im Detail

Methoden bestehen, wie bereits erwähnt, aus einer optionalen Slotliste sowie einem daran anschließenden Codeteil. Die Slots stellen das Pendant zu den lokalen Variablen in anderen Sprachen (insbesondere Smalltalk) dar, das heißt, sie sind nur lokal sichtbar. Zusätzlich gibt es noch Argumentslots, deren Namen mit einem Doppelpunkt beginnen. Der Codeteil besteht aus einer Sequenz von Ausdrücken (*expressions*), die durch Punkte seperiert werden.

2.4.2.1 Ausdrücke

Ausdrücke werden in SELF durch das Versenden von Nachrichten (*messages*) an Empfänger (*receiver*) gebildet. Die zugrundeliegende Syntax ist an der von Smalltalk angelehnt. Wie im Abschnitt über Slots beschrieben gibt es drei verschiedene Arten von Nachrichten (unäre, binäre, keyword), mit unterschiedlichen syntaktischen Regeln. Desweiteren unterscheidet man, je nach Empfänger, zwischen expliziten und impliziten Empfängernachrichten (*implicit/explicit-receiver messages*). Nachrichten, die an einen expliziten Empfänger gerichtet werden sind zum Beispiel:

```
5 factorial.
2 + 3.
expr IfTrue: [...].
```

Wenn die zu versendende Nachricht sich an das Objekt selber wendet, handelt es sich um eine implizite Nachricht. Einige Beispiele wären:

```
factorial.
+ 3.
1 + power: 3
```

Zu beachten ist, daß es einen Unterschied macht, ob man eine Nachricht implizit an das Empfängerobjekt sendet oder explizit an *self*. Während eine implizite Nachricht innerhalb einer Methode an das dazugehörige activation object richtet, wird durch *self* die Nachricht an das Objekt gesendet, das die Methode ausführt. Dazu ein Beispiel:

```
lobby _AddSlots: (|
    y = (|
        a = 1.
        getA1 = (| a=2| ^a).
        getA2 = (|a=2| ^self a)
    |)
|)
```

Es wird in der Lobby ein Slot *y* hinzugefügt, der aus einem Datenslot *a* mit dem initialen Wert 1 und den unären Methoden *getA1* und *getA2* besteht. Beide Methoden definieren lokale Slots *a* mit dem initialen Wert 2. Während *getA1* die implizite Nachricht *a* versendet, schickt *getA2* die gleiche Nachricht explizit an *self*. Wenn man nun beide Methoden auf folgender Weise aufruft erhält man unterschiedliche Ergebnisse:

```
lobby y getA1
2
lobby y getA2
1
```

2.4.2.2 resend

Desweiteren gibt es in SELF die Möglichkeit überschriebene Methoden anzusprechen, indem man das reservierte Wort *resend* gefolgt von einem Punkt und der zu versendenden Nachricht verwendet (*indirected resend*). Dadurch wird veranlaßt, daß SELF die Nachricht an seine Parentslots weiterleitet, vergleichbar mit *super* in Java oder Smalltalk. Wenn man hingegen nur einen Parent ansprechen möchte, kann man statt *resend* auch einfach seinen Namen angeben (*directed resend*), um so den verwendeten Namensraum einzuschränken. Ein Beispiel hierzu:

```
(|
    a = 'local a'.
    parent1* = (| a = 'a of parent1' |).
    parent2* = (| a = 'a of parent2' |).
    getLocal_a = (a).
    getDirectedResend_a = (^parent1.a).
    getIndirectedResend_a = (^resend.a).
|)
```

Folgende Eingaben führen zu folgenden Resultaten:

```
getLocal_a
-> 'local a'
getDirectedResend_a
-> 'a of parent1'
getIndirectedResend_a
-> ambigiuous message error
```

Die letzte Fehlermeldung entsteht durch die Mehrdeutigkeit bei der Mehrfachvererbung, da nicht klar ist, ob auf den Slot *a* von *parent1* oder *parent2* zugegriffen werden soll.

2.4.2.3 Method activation object

Bisher wurde nur betrachtet, wie Methoden in SELF aufgebaut sind. Es stellt sich nun die Frage, wie Methoden zur Laufzeit aufgerufen werden. Wenn in SELF eine Nachricht an ein Objekt gesendet wird, testet dieses Objekt zunächst, ob es im Besitzt eines gleichnamigen Slots ist. Ist dies nicht der Fall, so wird die Nachricht über die Parentslots weitergeleitet. Wenn ein Objekt gefunden wird, das den gewünschten Slot besitzt und es sich dabei um ein Methodenslot handelt, wird zuerst das darin enthaltene Methodenobjekt geklont. Bei diesem Klonen wird ein method activation object generiert, welches aus lokalen Slots, Argumentslots, einem Parent slot namens *self* sowie dem dazugehörigen Code besteht. Als nächstes wird der Parent slot *self* mit dem ursprünglichen Empfänger der Nachricht initialisiert. Dadurch wird sichergestellt, daß alle Nachrichten, die von der Methode selber versendet werden, zunächst beim aufrufenden Objekt selber gesucht werden (Polymorphie). Im nächsten Schritt werden die Argumentslots mit den Argumentobjekten initialisiert, die mit der Nachricht gesendet wurden. Dabei werden die Argumente in der Nachricht von links nach rechts in gleicher Reihenfolge in die Argumentslotvorkommen eingesetzt. Anschließend wird der Code im Kontext des neuen activation object ausgeführt. Der Rückgabewert wird durch den letzten Ausdruck bestimmt, der optional mit einem Circumflex ("^") beginnen kann. Eine Ausnahme bilden eingebettete Blöcke die mit einem abschließenden Circumflex vor dem letzten Ausdruck auftreten, da dann die Ausführung der Methode beendet wird (*non-local return*).

2.4.2.4 Primitive Methoden

Neben den bisher besprochenen Methoden, die vom Benutzer implementiert werden können, gibt es noch primitive Methoden in SELF. Primitive Methoden beginnen immer mit einem Unterstrich und sind *keyword*-Methoden. Sie verhalten sich bezüglich der Vererbung und Typisierung anders als herkömmliche Methoden. Während normale Methoden die Empfänger- und Argumentobjekte nicht auf ihren Typ überprüfen findet eine solche Prüfung bei primitiven Methoden statt, wie zum Beispiel bei *_IntAdd:*, *_FloatCeil*, usw. Der Vererbungsmechanismus für Methoden wird für primitive Methoden nicht angewandt. Wenn das Empfängerobjekt einer primitiven Nachricht diese nicht verarbeiten kann, wird sie also nicht an die Parentslots weitergereicht, sondern direkt eine Fehlermeldung zurückgegeben. Diese Fehlermeldungen können durch ein angehängtes *IfFail:* abgefangen werden.

Nach *IfFail:* wird als Argument ein Block angegeben, dem zwei Argumente übergeben werden. Das erste ist ein String, der die Fehlerursache beschreibt und das zweite gibt den Namen der fehlgeschlagenen, primitiven Nachricht an. Ein Beispiel für *IfFail:* ist:

```
5 _IntDiv: 0 IfFail:
      [| :error. :name |
          error _StringPrint.
          name _StringPrint.
          ^ 'caught an error'
      ]
```

Hier wird eine "Division durch Null" abgefangen, die Fehlermeldungen ausgegeben und anschließend ein String als Resultat zurückgegeben. Bei unären Nachrichten wird das *IfFail:* einfach an den Namen angehängt, also aus *_Perform* wird zum Beispiel *_PerfomIfFail:*.

2.5 Primitive Objekte in SELF

Bisher wurden nur Objekte betrachtet, die vom Benutzer in SELF selber generiert werden konnten. Es gibt aber auch primitive Objekte in SELF, die ein fester Bestandteil der Sprache sind. Eine Gemeinsamkeit von allen primitiven Objekten ist, daß man sie direkt nicht modifizieren kann, da sie nicht wie normale Datenobjekte behandelt werden.

Eine Manipulation wie zum Beispiel:

```
0 _AddSlots: (| x |)
```

ist nicht möglich, da Integerobjekte, wie alle primitiven Objekte, keine Slots aufnehmen können. Alle primitiven Objekte besitzen nur einen Parentslot namens *parent*, der auf ein normales SELF-Objekt verweist, welches wiederum einen Parentslot namens *parent* besitzt, der auf die Lobby verweist. Dieses anfangs (bis auf den Parentslot) leere Objekt dient dazu, benutzerdefinierte Slots aufzunehmen. So läßt sich zum Beispiel die Methode "+" für Integer auf folgender Weise implementieren:

```
0 parent _AddSlots: (| + arg = (_IntAdd: arg) |)
```

Wenn man einer Zahl die Nachricht "+" mit einem Argument sendet, wird die dazugehörige Methode gefunden und durch *_IntAdd:* die beiden Zahlen addiert.

2.5.1 Integer

Objekte vom Typ Integer stellen die ganzen Zahlen dar und beginnen mit einer Basis zwischen 2 und 36, gefolgt von einem "r" oder "R" und den Ziffern der Zahl. Wurde keine Basis angegeben, wird als Basis 10 angenommen. Optional kann noch ein Minuszeichen vor die Zahl gesetzt werden um negative Zahlen erzeugen zu können. Einige Beispiele für ganze Zahlen in SELF sind:

2r1011, 16r3fe4, 36rzz3r5, -8r777, 12000

2.5.2 Float

Objekte vom Typ float stellen die Fließkommazahlen in SELF dar. Fließkommazahlen werden durch einen Punkt seperiert, der die Vor- und Nachkommastellen trennt. Desweitern kann durch ein angefügtes "e" oder "E" ein positiver oder negativer Exponent zur Basis 10 angegeben werden. Einige Beispiele für Fließkommazahlen in SELF sind:

123.12, 66e5, 23.12E-3, 0.1e+11

2.5.3 String

Strings sind beliebige Zeichenketten die von "'"-Zeichen eingegrenzt werden. Wie in Sprachen wie C oder Java gibt es auch in SELF Escapesequenzen, die durch ein "\" eingeleitet werden, wie zum Beispiel "\t" (Tabulator), "\n" (Newline), "\'" (Single quote), usw. . Einzelne Zeichen können auch durch ihren ASCII-Code angegeben werden, der in hexadezimaler ("\x..."), oktaler ("\o...") oder dezimaler Form ("\d...") dargestellt wird. Ein Beispiel für einen String ist:

'Strings \tare \tnested in \n \'-characters'

Strings bauen in SELF auf Bytevektoren auf.

2.5.4 Byte vector

Bytevektoren sind die Grundlage für Strings oder auch Bytecode in SELF. Sie können durch primitive Methoden modifiziert werden, wie zum Beispiel *_ByteAt:Put:*.

2.5.5 Object vector

Objektvektoren können mehrere Objekte aufnehmen, die unter einem Index vom Typ Integer abgelegt werden können. Sie werden erzeugt, indem man dem leeren Prototypen des Objektvektors die primitive Nachricht *_Clone:Filler:* sendet. Dadurch wird der Vektor geklont und mit dem dem Wert, der hinter *Filler:* erscheint, initialisiert. Für den Zugriff auf die Daten existieren primitive Nachrichten wie z.B: *_At:* oder *_At:Put:*.

2.6 Vererbung in SELF

Bisher wurde beschrieben, wie Daten- und Methodenobjekte aufgebaut sind und wie sie ausgeführt werden, aber nicht wie sie gefunden werden, wenn sie von einem Objekt angefordert wurden. Dafür gibt es den *lookup algorithm*, der die Vererbung realisiert. Er wird in der Programmierreferenz [1] im Abschnitt 2.3.6 (*message lookup semantics*) auf Seite 15 formal definiert, weshalb sich im folgenden auf eine informelle Beschreibung beschränkt wird. Zusammengefaßt funktioniert der *lookup algorithm* in nachfolgend beschriebener Weise.

> Wenn ein Objekt eine Nachricht erhält, schaut es zunächst nach, ob diese Nachricht mit einem Unterstrich beginnt oder nicht. Beginnt sie mit einem solchen, dann handelt es sich dabei um eine primitive Nachricht. Primitive Nachrichten werden immer auf das Objekt angewandt, das diese empfangen hat und **nicht** weiterdelegiert an seine Parent-Objekte, falls sie nicht ausführbar ist. In diesem Fall wird sofort eine Fehlermeldung ausgegeben und der *lookup algorithm* beendet. Wenn die Nachricht nicht mit einem Unterstrich beginnt, dann handelt es sich dabei um eine gewöhnliche Nachricht und der Empfänger einer solchen prüft, ob er einen gleichnamigen Slot besitzt. Ist dies der Fall, wird sein Inhalt evaluiert und ein Rückgabewert zurückgegeben. Wenn hingegen kein passender Slot gefunden wird, wird die Nachricht an alle Objekte delegiert, die sich in den vorhandenen Parentslots befinden. Da SELF Mehrfachvererbung unterstütz, können dies beliebig viele sein. Dort wird die Nachricht so lange an die Parentslots weiterdelegiert, bis ein passender Slot gefunden wurde. Dieser wird dann gecloned und zu dem Objekt geschickt, das die Nachricht ursprünglich erhalten hat und dort evaluiert.
>
> Ein Sonderfall stellen Nachrichten dar, die mittels eines *resend* verschickt wurden. In so einem Fall wird der erste gefundene Slot nicht evaluiert, sondern an alle Parentobjekte des Objektes, das diesen Slot sendete (*sending method holder*), diese Nachricht weiterdelegiert. Bei einem *indirected resend* wird die Nachricht statt an alle Parentobjekte nur an das spezifizierte Parentobjekt (*delegatee*) weiterdelegiert.

Beim delegieren können drei verschiedenen Fehlerarten auftreten:

1. Zyklen beim Delegieren (message cycling)
2. Mehrdeutige Vererbung (ambigious inheritance)
3. Kein Slot gefunden

Zyklen können dann auftreten, wenn die Vererbungshierachie so angeordnet ist, daß eine Nachricht aif ihrem Suchweg ein und dasselbe Objekt mehr als einmal erreicht.

Mehrdeutige Vererbung tritt dann auf, wenn mehr als ein Slot gefunden wurde, dessen Inhalte unterschiedliche Objekte sind. Wenn mehrere Slots gefunden wurden, die alle auf dasselbe Objekt referenzieren, so handelt es sich dabei um keine mehrdeutige Vererbung, unabhängig davon, wo diese Slots sich befanden. Wenn kein Slot gefunden wurde, ist die Suche fehlgeschlagen und es wird eine Fehlermeldung ausgegeben.

Im folgenden werden einige Situationen an einem Beispiel dargestellt. Gegeben seien folgende Objekte:

```
A = (|
    method1 = (...)
|)
B = (|
    parent* = A.
    method2 = (...)
|)
C = (|
    parent* = A.
    method2 = (...).
    method3 = (...)
|)
D = (|
    parent1* = B.
    parent2* = C.
    method4 = (...)
|)
```

Hier werden 4 Objekte A, B, C und D deklariert. A stellt die Wurzel dar, während B und C von diesem erben. D erbt von B und C.

Wenn man nun " D method4" eingibt, wird der Slot direkt gefunden und ausgführt.

"D method3" wird über "parent2*" im Objekt C gefunden und anschließend im Kontext von Objekt D ausgeführt.

"D method1" wird zweimal gefunden, da der Slot über beide Parentslots erreichbar ist. Dies führt allerdings zu keiner Mehrdeutigkeit, da der Inhalt jeweils dasselbe Objekt ist.

"D method2" führt hingegen zu einer Fehlermeldung, da hier zwei Slots mit unterschiedlichem Inhalt gefunden werden. Somit ist die Nachricht also mehrdeutig.

"D method5" führt zu einem Fehler, da ein Slot mit solch einem Namen nicht existiert.

2.7 Die initiale SELF-Welt

Wenn man die Virtual Machine von SELF startet, befindet man sich in der initialen Welt von SELF. Mit initial ist hier gemeint, daß sich in ihr nur die grundlegendsten Objekte von SELF befinden und nicht die SELF-Welt mit zum Beispiel den *traits*, *globals* oder *mixins*, wie sie in vielen Dokumenten zu SELF beschrieben werden. Diese Objekte werden durch ein Skript "init.self" hinzugeladen, welches wiederum andere Skripte hinzulädt, die diese Objekte definieren. Gleiches gilt wenn man einen Snapshot startet, da in ihm "init.self" bereits enthalten ist. Auf diese SELF-Welt wird im folgenden nicht näher eingegangen, da eine solche Betrachtung zum einen sehr umfangreich wäre und zum anderen dazu umfangreiche Dokumente von Sun verfügbar sind, wie zum Beispiel [2].

Die initiale Welt besteht aus einem Objekt *lobby*, das die drei Slots *snapshotAction*, *shell* und den parent slot *systemObjects* beinhaltet. In *systemObjects* befinden sich Slots, die alle grundlegenden Objekte von SELF beinhalten, wie zum Beispiel *nil*, *true*, *false*, *vector*, usw. Desweiteren befinden sich dort noch Slots für Mirror- und Proxyobjekte sowie einem Profiler. Im folgenden werden einige dieser Objekte genauer beschrieben.

2.7.1 Mirrorobjekte

Bei Mirrorobjekten handelt es sich um "Spiegelbilder" von Objekten in Form eines Objektvektors (ähnlich zu Javas Reflections). Dieser Objektvektor besteht aus einer Ansammlung von Schlüsseln vom Typ string, die auf die Objekte verweisen, auf die die Slots des gespiegelten Objekts (*reflectee*) verweisen. Mirrors können benutzt werden, um Objekte zu untersuchen oder um sie zu modifizieren, indem man vorhandene Slots löscht oder neue hinzufügt.

2.7.2 Proxyobjekte

Bei Proxyobjekten handelt es sich um Schnittstellen zur Programmiersprache C oder C++. Man kann mit ihnen in C/C++ geschriebenen Code importieren und ihn als SELF-Objekt ansprechen.

2.7.3 Profiler

Der Profiler ist ein Hilfsmittel, mit dem man die Effizienz von Programmen, genauer gesagt die benötigte Zeit, messen kann.

2.7.4 Die Lobby

Die Lobby (deutsch: Vorhalle) stellt die Wurzel der ganzen SELF-Welt dar. Sie ist im Gegensatz zu den vorhergenannten Objekten kein primitives, sondern ein normales Datenobjekt, aber sie nimmt in der SELF-Welt eine Sonderrolle ein.

Sie spielt insbesondere beim konstruieren neuer Datenobjekte eine Rolle, da alle Ausdrücke, die den Datenslots zugewiesen werden, im Kontext der Lobby ausgeführt werden. Dazu ein Beispiel:

```
(|
  slot1 = 'hallo'.
  slot2 = self slot1.
|)
```

In diesem Beispiel wird ein Datenobjekt mit zwei Datenslots erzeugt. Zu beachten ist aber, daß der zweite Slot im Kontext der Lobby ausgeführt wird und sich *self slot1* daher nicht auf den ersten Slot im gleichen Objekt bezieht, sondern auf einen Slot *slot1* in der Lobby.

2.8 SELF versus klassenbasierte Sprachen

Wie in den Kapiteln zuvor schon ausführlich dargestellt wurde, gibt es in SELF nur Objekte und keine Klassen. Dies hat weitreichende Konsequenzen für die Programmierung, die im folgenden dargestellt werden.

2.8.1 Klassen versus Prototyp-Objekte

In klassenbasierten Programmiersprachen wie zum Beipiel Java oder Smalltalk unterscheidet man zwischen Klassen und deren Instanzen, den Objekten. Klassen stellen hier das Verhalten und die Objekte den Zustand einer Entität dar. In SELF hingegen gibt es nur Objekte, die Zustand und Verhalten beinhalten. Eine spezielle Aufgabe haben hier die Prototypen, die als Vorlage für neue Objekte benutzt werden, indem man sie klont, inklusive aller initialisierten Datenslots. Klonen bedeutet eine exakte Kopie (*shallow copy*) des Prototyps zu erzeugen, was einer Instanz in klassenbasierten Sprachen entspricht. Dies ist einfacher als das Erzeugen einer Klasseninstanz, da dort die Objekte erst nach einem vorgegebenen Plan (im Konstruktor) erzeugt und initialisiert werden müssen, was für einen Betrachter weniger anschaulich ist als das einfache kopieren.

Wenn es keine Klassen gibt, dann kann es natürlich auch keine Klassenvariablen geben. Man kann sie aber leicht simulieren, indem man zum Beispiel eine Klasse wie:

```
Klasse A:
    Klassenvariable x
    Instanzvariable y
```

in SELF durch folgende Objekte realisiert:

```
Objekt A = (| x |)
Object A' = (| y. p* = A |)
```

Wenn man nun Objekt *A'* klont, dann erhält man eine Kopie, die einen eigenen Slot *y* besitzt, den sie alleine nutzt (entspricht einer Instanzvariable) und erbt von Objekt *A* den Slot x, der von allen Klonen von *A'* gemeinsam benutzt wird (entspricht einer Klassenvariable).

Ein weiterer Vorteil von Prototyp-Objekten stellen die sogenannten *one-of-a-kind-objects* dar. Darunter versteht man Objekte, von denen es nur ein Objekt im ganzen System gibt, wie zum Beispiel *true* und *false*. In einer Sprache wie Smalltalk gibt dafür die zwei Klassen *True* und *False*, die die Methoden wie zum Beispiel *ifTrue:* oder *ifFalse:* zur Verfügung stellen und deren Instanzen, die die Werte *true* und *false* darstellen. In SELF ist dafür jeweils nur ein Objekt notwendig, da dieses Zustand und Verhalten in sich vereint.

2.8.2 Slots versus Variablen und Methoden

Ein weiterer Unterschied zwischen SELF und anderen herkömmlichen objektorientierten Sprachen sind die Slots. Während zum Beispiel in Java eine Variable in einer abgeleiteten Klasse nur von einer anderen Variablen mit dem gleichen Namen verdeckt werden kann und nur eine Methode von einer anderen Methode, kann hingegen in SELF ein Datenslot von einem Methodenslot überdeckt werden und umgekehrt. Dies ermöglicht zum Beispiel einen Datenslot mit einer Zahl durch eine Methode zu ersetzen, die per Zufallsgenerator eine Zahl erzeugt.

Ein weiterer Unterschied liegt in der Dynamik von SELF, die es ermöglicht Objekte zur Laufzeit zu manipulieren, indem man zum Beispiel neue Slots hinzufügt, vorhandene löscht oder durch andere ersetzt, wobei es keine Rolle spielt, ob man einen Methodenslot durch einen Datenslot ersetzt oder einen Zuweisungsslot. In klassenbasierten Sprachen ist durch die Trennung von Zustand und Verhalten so eine Modifikation nur schwer zu realisieren, denn wenn man zum Beispiel eine Klasse zur Laufzeit erweitern würde, stellt sich die Frage was dies für die bereits existierenden Instanzen dieser Klasse bedeutet. Da Objekte in SELF Zustand und Verhalten in sich vereinen gibt es solche Probleme nicht.

Da in SELF die Parentslots, die für die Vererbung verwendet werden, wie andere Datenslots auch zuweisbar sein können, gibt es in SELF die sogenannte *dynamische Vererbung*. Darunter versteht man, daß ein Objekt zur Laufzeit sein Parent-Objekt ändern kann. Speziell in typisierten klassenorientierten Sprachen wäre dies nur schwerlich zu realisieren.

2.8.3 Traits, Mixins und Globals

In SELF wird, je nach Verwendungszweck, noch zwischen *trait*, *mixin* und *global objects* unterschieden.

Unter *traits* versteht man Objekte, die ein Verhalten, das von anderen Objekten geteilt werden soll, zur Verfügung stellen. So stellt *traits clonable* zum Beispiel die Methode zum klonen von Objekten zur Verfügung, die von den meisten klonbaren Objekten geteilt wird.

Mixins sind kleine Objektsammlungen, die keine Parentslots besitzen. Sie stellen Methoden zur Verfügung, die dazu gedacht sind in andere Objekte nach einer Art Baukastenprinzip "hinzugemischt" zu werden.

Globals sind Objekte, die global im ganzen System zur Verfügung stehen sollen. Dies sind zum Beispiel die Prototypen oder auch die *one-of-a-kind-objects*, die man auch *oddballs* nennt.

2.9 Ein Beispiel in SELF

Nachdem in den vorhergehenden Abschnitten die einzelnen Bestandteile der Sprache beschrieben wurde, soll nun an einem Beispiel die Programmierung verdeutlicht werden. Es soll ein Stack implementiert werden.

```
_AddSlots: (| stack = () |).
stack _Define: (|
    stack = vector cloneSize: 100.
    topPointer <- 0.
```

```
        push: anObject = (
            topPointer: (topPointer + 1).
            stack at: topPointer Put: anObject.
        ).
        pop = (topPointer: (topPointer - 1)).
        top = (^stack at: topPointer)
    |)
```

Hier wird ein Stackobjekt erzeugt, das auf einem Objektvektor mit der Länge 100 basiert, welches im Datenslot *stack* gespeichert wird. Der Integerwert im zuweisbaren Datenslot *topPointer* stellt den aktuellen Stackzeiger dar und wird mit 0 initialisiert. Der Methodenslot *push:* übernimmt ein Objekt als Argument und legt ihn im Objektvektor mit der Methode *at:Put:* an der Stelle *topPointer + 1* ab. *pop* entfernt das oberste Element des Stacks und *top* gibt das oberste Element des Stacks zurück.

Dieses Objekt stellt nun einen Prototypen für Stacks dar, das initial leer ist. Um jetzt eine neue Instanz zu erstellen, muß man dieses Objekt klonen. Dies kann man mit der primitiven Nachricht *_Clone* erreichen, indem man

```
    stack _Clone
```

aufruft und somit eine exakte Kopie erzeugt. Es ist allerdings generell keine elegante Vorgehensweise Objekte auf diese Art zu klonen, da man so möglicherweise auch Werte des Objekts mitkopiert, die bei der Kopie nicht benötigt werden. Was kopiert werden soll und was nicht sollte das Objekt selber entscheiden, indem es dem Benutzer eine eigene Klonemethode anbietet. Für unseren Stack reicht es aus ihn einfach zu kopieren, deshalb erben wir einfach die Methode *clone* von *traits clonable*, was zu folgendem Objekt führt.

```
    _AddSlots: (| stack = () |).
    stack _Define: (|
        parent* = traits clonable.
        stack = vector cloneSize: 100.
        topPointer <- 0.
        push: anObject = (
            topPointer: (topPointer + 1).
            stack at: topPointer Put: anObject.
        ).
        pop = (topPointer: (topPointer - 1)).
        top = (^stack at: topPointer)
    |)
```

*parent** enthält eine Referenz zum Objekt *traits clonable* und erbt somit die gewünschte Methode. Somit kann mit

```
    instance = stack clone
```

eine neue Instanz erstellen, die alle Slots des Prototypen besitzt. Nun kann man *instance* beliebig manipulieren, indem man zum Beispiel neue Slots hinzufügt oder *pop* durch einen eigenen Methodenslot ersetzt. Wenn wir aber nur Instanzen von Stack bilden wollen, deren Verhalten aber nicht verändert werden soll, dann bietet sich eine Trennung von Zustand und Verhalten an. Wir teilen dafür den Stack in zwei neue Objekte auf:

```
_AddSlots: (| stackBehaviour = () |).
stackBehaviour _Define: (|
    parent* = traits clonable.
    push: anObject = (
        topPointer: (topPointer + 1).
        stack at: topPointer Put: anObject.
    ).
    pop = (topPointer: (topPointer - 1)).
    top = (^stack at: topPointer)
|).
_AddSlots: (| stackInstance = () |).
stackInstance _Define: (|
    parent* <- stackBehaviour.
    stack = vector cloneSize: 100.
    topPointer <- 0.
|)
```

stackBehaviour beeinhaltet das Verhalten und *stackInstance* den initialen Zustand eines Objekts, daher übernimmt *stackBehaviour* hier gewissermaßen die Aufgabe einer Klasse in klassenbasierten Sprachen.

Unser Stack hat aber noch ein Problem, denn er ist nicht in der Lage einen Stacküber- oder Unterlauf lauf zu erkennen. Um dies zu erreichen stellen wir ein weiteres Objekt zur Verfügung:

```
_AddSlots: (| boundedStackBehaviour = () |).
boundedStackBehaviour _Define: (|
    parent* = stackBehaviour.
    stackLimit = 100.
    push: anObject = (
        top < stackLimit
            ifTrue: [ resend.push: anObject]
            ifFalse: [ 'Stack overflow' printString]
    ).
    pop = (
        top >= 0
        ifTrue: [resend.pop]
        ifFalse: ['Stack underflow' printString]
    ).
|)
_AddSlots: (| boundedStackInstance = () |).
```

```
boundedStackInstance _Define:
    (stackInstance clone) parent: boundedStackBehaviour
```

Zunächst wird ein neues Objekt *boundedStackBehaviour* erstellt, das das Verhalten von *stackBehaviour* erbt. Zusätzlich stellt es den Datenslot *stackLimit* zur Verfügung, der die maximale Stackgröße beinhaltet. Da dieser Datenslot an das Instanzobjekt vererbt wird und somit von allen zukünftigen Instanzen geteilt wird, verhält es sich wie eine Klassenvariable. Die Methodenslots *push:* und *pop* von *stackBehaviour* werden durch neue Versionen überschrieben, die einen Über- oder Unterlauf des Stacks abfangen. Dazu wird die Einhaltung der Grenzen überprüft und im Erfolgsfall die Überschriebene Methode aufgerufen, indem zum Beispiel *pop* an das Objekt im Parentslot *parent** durch einen *resend* weitergereicht wird. In diesem Beispiel reicht ein einfaches, ungerichtetes *resend*, da es nur einen Parentslot gibt. Im Fall von mehreren Parentslots sollte man besser einen *directed resend* wie zum Beipiel *parent.pop* verwenden. Zu beachten ist, daß es falsch wäre, die überschriebene Methode einfach mit *parent pop* anzusprechen, denn wenn *pop* im Kontext von boundedStackInstance ausgeführt wird, würde als *parent* der lokale Slot mit diesem Namen gefunden werden. Dort würde man dann aber wieder die überschriebene Methode finden, die durch dieses Mißverständnis wieder sich selbst aufruft, usw.

boundedStackInstance stellt ein neues Instanzobjekt dar. Es wird erzeugt, indem *stackInstance* geklont und anschließend der Inhalt des Parentslots mit *boundedStackBehaviour* überschrieben wird. Dieses Vorgehen demonstriert die dynamische Vererbung, da dieses Objekt nun nicht mehr von *stackInstance* sondern von *boundedStackBehaviour* erbt. Das daraus resultierende Objekt hat dann folgende Form:

```
boundedStackInstance = (|
    parent* <- boundedStackBehaviour.
    stack = vector cloneSize: 100.
    topPointer <- 0.
|)
```

Kapitel 3

Einführung in dSelf

Diese Einführung soll die Sprache dSelf beschreiben. dSelf ist ein Derivat zu SELF, weshalb zunächst die Unterschiede zwischen SELF und dSelf aufgezeigt werden. Der offensichtlichste Unterschied liegt in der Verteiltheit von dSelf, die durch entfernte Objektreferenzen realisiert wurde und im folgenden Text durch ein Beispiel demonstriert wird. Den Abschluß bildet eine Betrachtung der dSelf Virtual Machine und des dSelf Compilers.

3.1 Unterschiede zu SELF

Beim Entwurf von dSelf wurde Wert darauf gelegt, weitestgehend kompatibel zu SELF zu bleiben. Leider war das nicht an allen Stellen machbar, da die Verteiltheit von dSelf Änderungen notwendig machte. Desweiteren wurden einige Konzepte von SELF nicht übernommen, da sie sich entweder, wie zum Beispiel die priorisierte Mehrfachvererbung, in der Praxis nicht bewährten oder wie das Überlagern von Slots die Sprache teilweise sehr kompliziert gemacht haben. Ein weiterer Unterschied entsteht durch die Einführung von lokalen Methoden, der jedoch keine Inkompatibilität zu SELF darstellt, da es sich nur um eine Erweiterung der Sprache handelt. Ebenfalls eine Erweiterung ist natürlich die Verteiltheit von dSelf.

3.1.1 Keine Kapselung und priorisierte Mehrfachvererbung

Kapselung und priorisierte Mehrfachvererbung, wie in [3] beschrieben, sind nicht übernommen worden, da die Entwickler von SELF diese im nachhinein als ungünstig ansahen. So schreiben Randall B. Smith und David Ungar in [4] auf Seite 6 folgendes:

> “Early in the evolution of Self we made three mistakes: prioritized multiple inheritance, the sender-path tie-breaker rule, and method-holder-based privacy semantics. [...] But each feature also caused us no end of confusion. The prioritization of multiple parents implied

> that Self's "resend" (call-next-method) lookup had to be prepared to backup down parent links in order follow lower-priority paths. The resultant semantics took five pages to write down, but we perserved. After a year's experience with the features, we found that each of the members of the Self group had wasted no small amount of time chasing "compiler bugs" that were merely unforseen consequences of these features. It became clear that the language had strayed from its original path."

Aufgrund dieser beschriebenen Erfahrungen wurde darauf verzichtet diese Konzepte in dSelf zu übernehmen.

3.1.2 Keyword-Methodenslots und Zuweisungsslots

Ein weiterer Unterschied betrifft das Zusammenspiel zwischen Keyword Methodenslots und Zuweisungsslots, wie es unter 3.2.3 beschrieben wird. In SELF können Zuweisungsslots von gleichnamigen Methodenslots überlagert werden und umgekehrt. Wenn ein solcher, überlagernder Slot wieder gelöscht wird, wird der verborgene Slot wieder sichtbar. Je nachdem ob man *_AddSlots:* oder *_AddSlotsIfAbsent:* verwendet erhält man unterschiedliche Ergebnisse, deren Vorhersagen recht kompliziert sind, da sie nicht mit dem Standardverhalten dieser Primitiven Nachrichten übereinstimmen.

In dSelf wurde zugunsten einer leichten Verständlichkeit auf diese schwer nachvollziehbaren Vorgehensweisen verzichtet. Daher werden mit *_AddSlots:* **immer** vorhandene Slots durch gleichnamige ersetzt und mit *_AddSlotsIfAbsent:* **immer** nur nicht vorhandene Slots hinzugefügt.

3.1.3 Binäre Methoden

In SELF ist es nicht möglich unterschiedliche binäre Methoden hintereinander aufzurufen, wie in folgendem Beispiel:

```
x + y + z  ist korrekt,
x + y - z  hingegen nicht
```

Hier würde der Compiler von SELF einen Fehler melden und die Aktion abbrechen. In dSelf hingegen ist so ein Ausdruck erlaubt und wird von links nach rechts ausgewertet. Der Grund dafür liegt darin, daß dSelf mit einem Parsergenerator (CUP) erzeugt wurde, der auf der Basis von LALR(1)-Grammatiken arbeitet. LALR(k)-Grammatiken stellen eine eingschränkte Form der kontextfreien Grammatiken dar, die sich auf die deterministisch parsierbaren Grammatiken beschränkt. Um ein Konstrukt wie das Obige erkennen zu können, müßte die Grammatik aber mindestens kontextsensitiv sein, also läßt sich dies in kontextfreien (und damit auch in LALR(k)) Grammatiken nicht realisieren. Daher muß man insbesondere beim Umgang mit Zahlen auf die richtige Klammerung aufpassen, da es etwas wie Prioritäten für Operatoren (Punkt- vor Strichrechnung, etc.) in SELF und dSelf nicht gibt.

3.1.4 Lokale Methoden

Wenn man in SELF innerhalb einer Methode weitere Methoden deklarieren möchte, führt dies zu folgender Fehlermeldung:

> sorry - local methods not implemented yet.

Da SELF nicht mehr weiter entwickelt wird, ist eine Erweiterung um lokale Methoden auch nicht mehr zu erwarten. In einer verteilten Umgebung sind solche Methoden aber sehr nützlich, da dadurch eine Methode, die von einem entfernten Objekt aufgerufen wird, gleichzeitig alle benötigten Hilfsmethoden auf einen Schlag mitnehmen kann, ohne daß sie diese einzeln über die Grenzen der dSelf Virtual Machine anfordern muß, was einen höheren Kommunikationsaufwand bedeutet.

Ein weiterer Vorteil liegt in einer Art Kapselung von lokalen Methoden. Wie im vorhergehenden Punkt beschrieben wurde, gibt es in dSelf keine Kapselung und damit auch keine als *private* deklarierten Objekte. Lokale Methoden können aber private Methoden simulieren, da sie nur für die Methode sichtbar sind, in der sie deklariert wurden und dadurch auch nicht vererbt werden. Dadurch kann man sicher sein, daß garantiert diese Methode aufgerufen wird und nicht eine gleichnamige überschriebene.

Zu beachten ist, daß lokale Methoden, wie normale Methoden auch, immer im Kontext des Objektes ausgeführt werden, daß die Methode ausführt in dem die lokale definiert wurde.

Zur Verdeutlichung ein Beispiel:

```
test = (|
    data = 'outside'
    method = (|
        data = 'inside'.
        localMethod = (data _StringPrint)
        |
        localMethod
    )
|)
```

Hier wird ein Objekt in einem Slot namens *test* deklariert, das aus einem Datenslot *data* mit dem String "outside" und einer Methode *method* besteht. Die Methode besteht aus einem lokalen Datenslot, der ebenfalls *data* heißt mit dem Inhalt "inside" und einer lokalen Methode *localMethod*, die den Inhalt von *data* ausgibt. Wenn man nun *method* aufruft, führt dies zu folgendem Ergebnis:

```
> test method
outside
```

Wie man sieht, wird nicht *'inside'* ausgegeben, da der Ausführungskontext der lokalen Methode das Objekt ist, daß die Methode aufgerufen hat, in der die lokale Methode eingebettet ist. Daher sind lokale Methoden eher als normale, private Methoden zu betrachten, die nur für die Methode zugreifbar sind, in der sie definiert wurden.

3.1.5 Short, Long, Double

In SELF gibt es initial nur Integer und Floatobjekte, während in dSelf zwischen Short, Integer, Long, Float und Double unterschieden wird. Diese Objekte entsprechen in ihrer Bitbreite den gleichnamigen Javaobjekten, da sie auf diese abgebildet werden. Die Lexikalik für die einzelnen Typen lautet:

Short: <zahl>_s oder <zahl>_S, also z. B. 123_s

Integer: <zahl>, also z. B. 123

Long: <zahl>_l oder <zahl>_L, also z. B. 123_l

Float: <zahl>.<zahl>, also z. B. 123.456

Double: <zahl>.<zahl>_d oder <zahl>.<zahl>_D, also z. B. 123.456_d

Wie man sieht, fangen die Suffixe immer mit einem Unterstrich an und werden nicht wie in Java direkt angehängt. Der Grund dafür die Zahl in Radixnotation. Man kann in dSelf wie in SELF Zahlen zum Beispiel in folgender Form spezifizieren:

```
2r1110, 8r771, 16rff3e, 36rzz0l, 36rzz0_l
```

An den letzten beiden Zahlen wird der Grund für den Unterstrich deutlich. In 36rzz0l ist das "l" am Ende der Zahl vom Typ Integer ein Bestandteil dieser, während 36rzz0_l eine Zahl vom Typ Long spezifiziert. Ohne den Unterstrich wären solche Zahlen mehrdeutig.

3.1.6 Primitive Objekte true, false, nil

In SELF sind *true*, *false* und *nil* normale Objekte und können daher auch wie solche manipuliert werden. In dSelf sind diese Objekte primitiv, das heißt sie haben einen Parentslot, namens *parent*, der auf ein normales SELF-Objekt verweist, welches wiederum einen Parentslot namens *parent* besitzt, der auf die Lobby verweist. Dieses anfangs (bis auf den Parentslot) leere Objekt dient dazu, benutzerdefinierte Slots wie zum Beispiel *ifTrue:* aufzunehmen.

Der Grund, warum diese Objekte als primitive Objekte implementiert wurden, liegt an der Verteiltheit von dSelf, genauer gesagt an der Art und Weise, wie gewöhnliche (nicht primitive) Datenobjekte in dSelf, die sich außerhalb der dSelf Virtual Machine befinden, behandelt werden. Da es beim jetzigen Kenntnisstand schwierig ist, genau zu beschreiben wie dies in dSelf realisiert wird, wird zu einem späteren Zeitpunkt (im Abschnitt "Primitive und gewöhnliche Objekte") noch einmal genauer auf den Grund eingegangen.

3.2 Verteiltheit in dSelf

Der wesentlichste Unterschied zu SELF ist die Verteiltheit von dSelf, was auch durch den Namen (dSelf = distributed Self) zum Ausdruck gebracht werden soll. In den folgenden Abschnitten wird erörtert, wie man eine Verbindung zu einer anderen dSelf-VM aufbaut und wie dies intern realisiert wird.

3.2.1 Lobby-linking

Verteiltheit in dSelf bedeutet, daß Slots existieren können, deren Inhalte sich auf einer entfernten dSelf Virtual Machine (dSelf-VM) befinden können. Realisiert wurde dies in Java unter der Verwendung von RMI (Remote Method Invocation). RMI ermöglicht die Kommunikation von Javaobjekten über die Grenzen einer Java Virtual Machine hinaus. Um auf entfernte Javaobjekte zugreifen zu können, wird über eine RMI-Registry eine Verbindung zu diesem Objekt aufgebaut, daß unter einem selbstdefinierten Namen an diese gebunden wurde.

Um von einer dSelf-VM auf eine entfernte dSelf-VM zugreifen zu können, muß zunächst eine Verbindung zwischen beiden erstellt werden.

Dafür gibt es zwei primitive Methoden:

1. _AddSlot: <Slotname> ConnectedTo: <URL>

2. _AddSlotIfAbsent: <Slotname> ConnectedTo: <URL>

Mit beiden Methoden kann man einen Slot mit dem gegebenen Slotnamen erstellen, der mit dem entfernten Lobby-Objekt einer entfernten dSelf-VM verbunden wird, deren Adresse durch die übergebene URL angegeben wird. Die beiden Versionen unterscheiden sich darin, daß *_AddSlot:ConnectedTo:* einen eventuell bereits vorhandenen Slot überschreibt, während *_AddSlotIfAbsent:ConnectedTo:* einen solchen, bereits vorhandenen Slot nicht verändert, also keine Aktion ausführt.

Der Slotname wird als ein Objekt vom Typ String übergeben und muß einen lexikalisch korrekten Datenslot darstellen. Die URL wird ebenfalls durch einen String dargestellt und muß folgende Form besitzen:

'//hostname:port/vm_name'

Also zum Beispiel:

'//127.0.0.1:1099/myVM'

Der Name des Hosts und die Nummer des Ports spezifizieren den Ort der RMI-Registry und *vm_name* ist der Name, unter dem sich die entfernte dSelf-VM hat registrieren lassen. Der Hostname und die Portnummer sind optional, so daß standardmäßig der lokale Host und die Nummer 1099 angenommen werden. Der Name der dSelf-VM wurde entweder beim Start mit der Option “-b <name>” angegeben und an die lokale Registry gebunden, oder bei fehlender Angabe standardmäßig als “vm” angenommen.

Wenn man zum Beispiel eine Verbindung zum entfernten Lobby-Objekt der dSelf-VM mit dem Namen “vm” auf dem Host “127.0.0.2” aufbauen möchte, die im Slot “remoteVM” in der Lobby abgelegt wird, kann man dies durch folgende Eingabe erreichen:

```
lobby _AddSlot: 'remoteVM' ConnectedTo: '//127.0.0.2/vm'
```

Nun stehen dem Benutzer alle Slots der entfernten dSelf-VM zur Verfügung, auf die er wie auf alle lokalen Slots zugreifen kann.

Dazu ein Beispiel:

```
remoteVM _Print
lobby(r): (| systemObjects* = systemObjects(r). vm* = vm(r). |)
-> nil
```

Hier wird auf die entfernte Lobby zugegriffen und ihr Inhalt auf dem lokalen Host ausgegeben. An den angehängten “(r)” erkennt man, daß diese Slots nicht lokal vorhanden sind. Diese Slots kann man wie lokal vorhandene behandeln, also sie zum Beispiel löschen, neue Slots hinzufügen, ihren Inhalt ändern oder sie sonstwie modifizieren. Dies geschieht für den Benutzer vollkommen analog zur Bearbeitung von lokalen Slots, so daß er sich keine Gedanken darüber machen muß, wo sich die Objekte genau befinden.

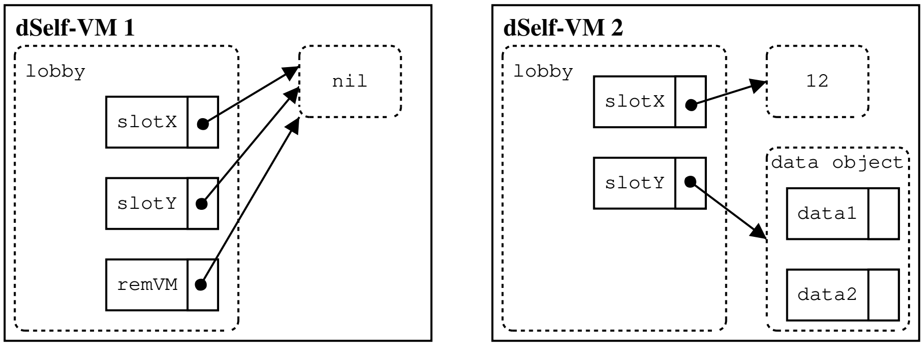
```
remoteVM _AddSlots: (| x = 12 |)
-> lobby(r)
remoteVM _Print
lobby(r): (| systemObjects* = systemObjects(r). vm* = vm(r).
x = integer(12). |)
-> nil
```

Hier wird die entfernte Lobby von der lokalen Lobby aus um den Slot x mit dem Wert 12 erweitert. Auffällig ist hier das Fehlen des “(r)” hinter x = *integer(12)*. Der Grund dafür liegt im Unterschied der Behandlung von primitiven und nicht-primitiven Objekten in dSelf, wie er im nächsten Abschnitt beschrieben wird.

3.2.2 Verteilte Objektreferenzen

Bisher wurde gezeigt, daß man mit *_AddSlot:ConnectedTo:* eine Verbindung zu einer entfernten dSelf-VM aufbauen und auf dortige Objekte zugreifen kann. In diesem Abschnitt soll nun der technische Hintergrund solcher Aktionen augezeigt werden, da man diese Kenntnisse in der Praxis benötigt um mögliche Ineffizienzen vermeiden zu können.

Zur Verdeutlichung der einzelnen Schritte beim Verbindungsaufbau und zur Darstellung, wie sie intern realisiert werden, wird in den folgenden Abbildungen gezeigt, wie ein verteilte Objektreferenz zu einer entfernten Lobby erstellt wird und wie man über diese auf die dort vorhandenen Objekte zugreifen kann.

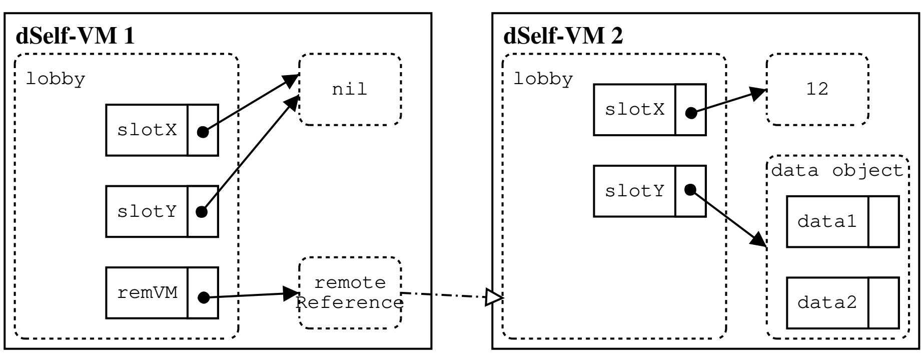


In dieser Abbildung werden zwei unterschiedliche dSelf-VM dargestellt. Die gestrichelten Rechtecke mit den abgerundeten Ecken stellen Objekte dar und die in ihnen befindlichen Rechtecke sind ihre Slots. Die Slots bestehen aus einem Namen und einer Referenz auf die Objekte, die sie beinhalten. In jeder dSelfVM befindet sich ein Lobby-Objekt mit seinen dazugehörigen Slots. Die Lobby der *dSelf-VM1* besitzt drei Slots, die mit *nil* initialisiert wurden, während die Slots der Lobby von *dSelf-VM2* auf das primitive Integerobjekt 12 und ein gewöhnliches Datenobjekt referenzieren.

Nun soll *dSelf-VM1* eine Verbindung zur Lobby von *dSelf-VM2* aufbauen und im Slot *remVM* ablegen. Dies wird durch folgende Eingabe bei *dSelf-VM1* erreicht:

```
lobby _AddSlot: 'remVM' ConnectedTo: '//127.0.0.1/dSelf-VM2'
```

Es wird ein neuer Slot *remVM* in der Lobby von *dSelf-VM1* erzeugt, der auf die Lobby von *dSelf-VM2* verweist. Wie dies intern realisiert wird, geht aus der nächsten Abbildung hervor:

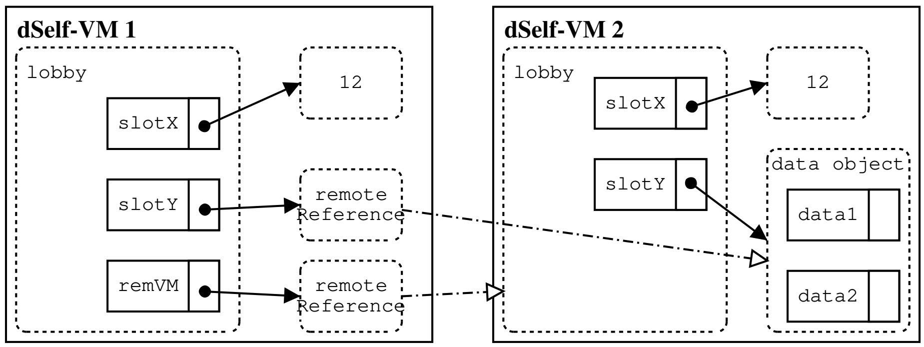


Der neue Slot *remVM* zeigt auf das Objekt *remote Reference*, welches auf die entfernte Lobby verweist. *remote Reference* wurde von *_AddSlot:ConnectedTo:* erzeugt und stellt eine Art “Pseudoobjekt” dar, da es lediglich alle Nachrichten, die es empfängt an die Lobby von *dSelf-VM2* weiterschickt.

Im nächsten Schritt soll nun über diese neue Verbindung auf die Lobby von *dSelf-VM2* zugegriffen werden.

```
lobby slotX _Define: remVM slotX
lobby slotY _Define: remVM slotY
```

Die Slotinhalte von *slotX* und *slotY* werden überschrieben mit den Inhalten der gleichnamigen Slots auf der entfernten Lobby. Das Ergebnis dieser Aktion ist in dieser Abbildung sichtbar.



Dem Slots *slotX* ist nun ein lokales Objekt 12 zugeordnet, während *slotY* über eine Objektreferenz auf das entfernte Datenobjekt zeigt. Hier wird der Unterschied zwischen primitiven und nicht primitiven Objekten deutlich. Da 12 ein Integer und damit ein primitives Objekt ist, wird es über die Grenzen der dSelf Virtual Machine hinweg kopiert und ein lokales Objekt mit dem gleichen Wert erzeugt. Für *slotY* hingegen wurde eine *remote Reference* erzeugt, da es auf ein gewöhnliches Datenobjekt verweist.

3.2.3 Primitive und gewöhnliche Objekte

Aus dem vorhergehenden Beispiel wird eine Regel deutlich, die in dSelf gilt:

> **Primitive Objekte werden immer zu der dSelf Virtual Machine kopiert, von der sie angefordert werden. Gewöhnliche Objekte bleiben immer dort, wo sie erzeugt wurden und werden über eine Objektreferenz angesprochen, die eigens für sie erzeugt wird.**

Der Grund für dieses Vorgehen soll an einem Beispiel verdeutlicht werden:

```
[dSelf-VM2]
lobby _AddSlots: (|
    objectOnVM2 = (|
        xIsZero = (
            0 = x
                ifTrue: [result = 'is zero'].
                ifFalse: [result = 'isn't zero'].
        )
    |)
|)
```

Es wird in der Lobby von *dSelf-VM2* ein Datenobjekt mit einem Methodenslot *xIsZero* erzeugt, der ein Methodenobjekt beinhaltet, das prüft, ob ein Datenslot x positiv ist. Die Art und Weise wie die Methode implementiert wurde, entspricht sicher nicht den gängigen Programmierkonventionen, eignet sich so aber besser für die folgende Demonstration.

Auf der anderen dSelf Virtual Machine wird folgender Slot hinzugefügt:

```
[dSelf-VM1]
lobby _AddSlots: (|
    objectOnVM1 = (|
        parent* = remVM objectOnVM2.
        x = 10.
        result <- ''
    |)
|)
```

Dieses Objekt erbt über die Grenzen der lokalen dSelf-VM hinweg von Objekt *objectOnVM2* die Methode *xIsZero*. Bei einem Methodenaufruf in der Form:

```
[dSelf-VM1]
lobby objectOnVM1 xIsZero
```

führt dazu, daß die Methode *xIsZero* von *dSelf-VM2* nach *dSelf-VM1* geklont und im Kontext von *objectOnVM1* ausgeführt wird. Da die Methode nur primitive Datenobjekte wie 0, String und implizit *true* oder *false* beinhaltet, werden ihre primitiven Pendants auf *dSelf-VM1* benutzt. Dadurch wird die notwendige Kommunikation zwischen *dSelf-VM1* und *dSelf-VM2* auf das bloße Anfordern der Methode reduziert.

Wenn man hingegen primitive Objekte nicht lokal verwenden würde, sondern wie bei gewöhnlichen Objekten über Objektreferenzen auf sie zugreifen würde, **ergäbe** sich folgendes Szenario:

> Wenn die Methode *xIsZero* von *objectOnVM1* bei *objectOnVM2* angefordert wird, wird die Methode über die Grenzen der dSelf Virtual Machine gecloned und im Kontext von *objectOnVM1* ausgeführt. In dieser Methode wird zunächst geprüft, ob der Inhalt des Slots x, der auf der lokalen Maschine gefunden wird, äquivalent mit 0 ist. Dazu wird 0 die Nachricht "=" mit x als Argument geschickt. Da 0 wie *xIsZero* von *dSelf-VM2* stammt, wird "=" über die Grenzen hinweg zur 0 von *dSelf-VM2* geschickt und dort ausgewertet. Das Ergebnis (*false*) dieser Auswertung wird dort erzeugt und wieder zurückgeschickt. Jetzt wieder in *dSelf-VM1* wird *false* die Nachricht *ifTrue:* mit einem Block als Argument gesendet. Da *false* von *dSelf-VM2* stammt, wird *ifTrue:* nach dort geschickt und schlägt fehl. Mit dem anschließenden *ifFalse:* passiert dasselbe, doch diesmal wird der Blockinhalt ausgewertet. Dazu wird innerhalb von *ifTrue:* die Nachricht *value* an den Block auf *dSelf-VM2* geschickt,

> da er ebenfalls dort definiert wurde. Nun soll *result* der Wert 'isn't zero' zugewiesen werden, was eine weitere Kommunikation, diesmal von *dSelf-VM2* nach *dSelf-VM1*. Da 'isn't zero' von *dSElf-VM2* stammt und gewöhnliche Objekte immer dort bleiben, wo sie erzeugt wurden, wird dazu eine neue *remote Reference* von *dSelf-VM1* nach *dSelf-VM2* erzeugt und bei jedem zukünftigen Zugriff auf *result* wird über die Grenzen der dSelf Virtual Machine hin kommuniziert werden müssen, um den Inhalt zu ermitteln.

Es ist vielleicht etwas schwer jedem der beschriebenen Schritte zu folgen, aber das ist hier nicht so wichtig. Was verdeutlicht werden soll, ist der enorme Kommunikationaufwand, der entsteht, wenn man primitive Objekte wie gewöhnliche Objekte behandeln würde. Aus diesem Grund wurden auch *true*, *false* und *nil* in dSelf als primitive Objekte realisiert.

Dies hat allerdings eine Konsequenz, deren sich der Benutzer bewußt sein muß. Wie zuvor schon beschrieben wurde, haben primitive Objekte einen Parentslot, der auf ein (initial leeres) Objekt zeigt, das die vom Benutzer definierten Slots beinhaltet, wie zum Beispiel "+", "-" oder "*". Um ein kompatibles Verhalten auf allen dSelf-VM zu erreichen, muß man dafür Sorgen, daß die Inhalte dieser Parentobjekte überall gleich sind. Dies läßt sich zum Beispiel dadurch realisieren, indem man jede dSelf-VM mit dem gleichen Skript startet, daß alle Methoden für die primitiven Objekte definiert und dann darauf verzichtet diese zu modifizieren.

Andererseits kann man auch selber dafür sorgen, daß bei Änderungen die anderen Parentobjekte ebenfalls angepaßt werden. Hier ein Beispiel, das den Ansatz dafür andeutet:

```
[dSelf-VM1]
lobby _AddSlots: (|
    localIntegerParent = 0 parent.
    localLongParent = 0_l parent.
    localFloatParent = 0.0 parent.
    localStringParent = '' parent.
    localNilParent = nil parent.
    localTrueParent = true parent.
    ...
|)
```

Hier werden alle lokalen Parentobjekte in neuen Slots in der Lobby (oder jedem beliebigen anderen erreichbaren Objekt) abgelegt und dadurch für entfernte Objekte zugreifbar. Eine Anpassung läßt sich dann einfach durch folgenden Code realisieren:

```
[dSelf-VM2]
0 parent _AddSlotsIfAbsent: remVM localIntegerParent.
true parent _AddSlotsIfAbsent: remVM localTrueParent.
...
```

Nun werden alle nicht vorhandenen Slots der Parentobjekte von *dSelf-VM2* mit denen von *dSelf-VM1* angeglichen. Zu beachten ist hier, daß bewußt *_AddSlotsIfAbsent:* und nicht *_AddSlots:* benutzt wurde, da es in jedem Parentobjekt einen Slot namens *parent* gibt, der auf die lokale Lobby verweist. Wenn man *_AddSlots:* verwendet, würde dadurch der lokale Parentslot, der auf die lokale Lobby verweist, durch den entfernten Parentslot überschrieben und fortan auf die entfernte Lobby verweisen. Sollte sich der Gebrauch von *_AddSlots:* nicht vermeiden lassen, sollte man darauf achten den Parentslot anschließend manuell auf die lokale Lobby zu setzen.

3.2.4 Methoden deklarieren und evaluieren

Bisher wurde nur gesagt, daß man Methoden über die Grenzen einer dSelf Virtual Machine hinweg aufrufen und lokal ausführen kann. Desweiteren wurde beschrieben, daß primitive Objekte immer im Kontext der lokalen dSelf-VM benutzt werden und für angeforderte, gewöhnliche Datenobjekte entfernte Objektreferenzen erzeugt werden, die die Nachrichten zum gewünschten Objekt weiterreichen. Es stellt sich aber die Frage, wie gewöhnliche Datenobjekte in Methoden behandelt werden. Hierzu ein Beispiel:

```
[dSelf-VM2]
objectOnVM2 _AddSlots: (|
    method = (
        | local1 = (| int = 1 |). local2 |
        local2: (| int = 1 |).
        local1 _Print.
        local2 _Print.
    )
|)
```

objectOnVM2, das sich auf *dSelf-VM2* befindet wird um eine Methode *method* erweitert. Diese Methode deklariert zunächst zwei Datenslots *local1* und *local2*, deren Objekte beide den gleichen Aufbau haben. Sie besitzen beide einen Slot *int* der auf die Zahl 1 verweist. Der Unterschied zwischen beiden Slots ist der, daß *local1* bei der Deklaration schon definiert wird, während *local2* mit *nil* initialisiert wird und anschließend sein Datenobjekt zugewiesen bekommt. Danach sollen beide Objekte in einer low-level Notation ausgedruckt werden. Wenn man nun diese Methode in *dSelf-VM1* aufruft, führt das zu folgendem Ergebnis:

```
[dSelf-VM1]
objectOnVM1 method
an object(r): (| int = integer(1). |)
an object: (| int = integer(1). |)
-> nil
```

Wie zu sehen ist, haben beide Objekte noch dieselbe Struktur, jedoch mit einem entscheidenden Unterschied. Beim Ausdruck von *local1* wurde ein “(r)” an die

Beschreibung angehängt, während es bei *local2* fehlt. Offensichtlich wurde das Objekt in *local1* auf der entfernten dSelf Virtual Machine erzeugt, während das Objekt in *local2* lokal erzeugt worden ist. Der Grund dafür liegt in folgender Regel:

> **In Methoden werden jene gewöhnlichen Datenobjekte, die bei der Slotdeklaration bereits einem Slot zugewiesen werden, immer im Kontext der dSelf Virtual Machine erzeugt, in der die Methode implementiert wurde. Die gewöhnliche Datenobjekte im Code einer Methode hingegen werden immer im Kontext der dSelf Virtual Machine erzeugt, in der die Methode ausgeführt wird.**

Theoretisch betrachtet, macht dies keinen Unterschied, da beide Objekte gleich angesprochen werden können. Jedoch aus praktischer Sicht würde ein häufiger Zugriff auf ein entferntes Objekt durch den hohen Kommunikationsaufwand viel mehr Zeit beanspruchen als der Zugriff auf ein lokales Objekt.

Prinzipiell hätte man es auch so machen können, daß die Objekte immer im Kontext der ausführenden dSelf-VM erzeugt werden. Doch dann wäre dSelf in diesem Punkt zum einen inkompatibel zu SELF, da dort die Slots der Methoden ebenfalls zum Zeitpunkt der Deklarierung initialisiert werden, und zum anderen würde man dadurch an Mächtigkeit verlieren. Denn durch diese Herangehensweise kann eine Methode auch auf Objekte zugreifen, die sich an ihrem Entstehungsort befinden.

3.2.5 Lokale Methoden in einer verteilten Umgebung

Im Kapitel “Unterschiede zu SELF” wurden bereits lokale Methoden vorgestellt. Hier wird noch einmal auf sie eingegangen, da ihr hauptsächlicher Existenzgrund jetzt erst verständlich wird.

Dazu ein Beispiel:

```
[dSelf-VM2]
objectOnVM2 _AddSlots: (|
    m1 = (...).
    m2 = (...).
    ...
    mn = (...).
    m = ([<cond>] whileTrue: [ m1. m2. ... mn])
|)
```

Hier werden Methoden m_1 bis m_n angedeutet, die in der Methode m in einer *whileTrue:*-Schleife aufgerufen werden. Wenn nun die Methode m auf der entfernten *dSelf-VM1* mit

```
[dSelf-VM1]
objectOnVM1 m
```

aufgerufen wird m geklont und im Kontext von *objectOnVM1* ausgeführt. In m wird nun eine beliebige Bedingung abgefragt und die anschließende Schleife ruft die Methoden m_1 bis m_n so lange auf, bis die Bedingung erfüllt wird. Um m_1 bis m_n auszuführen, muß jede Methode seperat von dSelf-VM2 nach *dSelf-VM1* gebracht werden und dies so oft, bis die Bedingung der Schleife erfüllt wird. Das Kommunikationsaufkommen beträgt für dieses Beispiel also:

Anzahl der Aufrufe = Anzahl Schleifendurchläufe * n + 1

Die 1 steht für den Aufruf der Methode m selber. Selbst wenn die Anzahle der Methoden (n) eher gering ist, kann dieser Wert bei einer häufigen Iteration sehr groß werden und somit viel Kommunikation verursachen. Um diesem Problem entgegen zu wirken, gibt es die lokalen Methoden.

```
[dSelf-VM2]
objectOnVM2 _AddSlots: (|
    m = (|
        m1 = (...).
        m2 = (...).
        ...
        mn = (...).
        |
        [<cond>] whileTrue: [ m1. m2. ... mn])
|)
```

In diesem Beispiel werden die Methoden m_1 bis m_n als lokale Methoden definiert. Wenn nun m von *objectOnVM1* aufgerufen wird, wird m mitsamt den lokalen Methoden zum Empfänger gebracht und dort ausgeführt. Jeder Zugriff auf m_1 bis m_n wird nun lokal auf *dSelf-VM2* ausgeführt und es entsteht keine zusätzliche Kommunikation zu *dSelf-VM1*.

Zu beachten ist, daß der Aufruf einer lokalen Methode gegenüber einer extern definierten Methode einen Unterschied bezüglich der Polymorphie und Sichtbarkeit macht. Lokale Methoden sind nur von der Methode aufrufbar, in der sie definiert wurden und werden immer von ihr direkt gefunden, d. h. sie können nicht von gleichnamigen Slot überschrieben werden.

Vom Verhalten her sind lokale Methoden identisch zu extern definierten Methoden, d. h. ihre Slots werden im Kontext der dSelf-VM initialisiert, in der sie implementiert wurden und der Code immer beim Empfänger evaluiert.

3.2.6 Zusammenfassung

Es wurde gezeigt, wie man in dSelf über die primitiven Nachrichten *_AddSlotIfAbsent:ConnectedTo:* oder *_AddSlot:ConnectedTo:* eine Verbindung zu einer anderen dSelf Virtual Machine aufbauen und dann auf deren Inhalte zugreifen kann. Der Zugriff auf die entfernten Slots erfolgt genauso wie auf lokale Slots, ohne daß der Benutzer sich darüber Gedanken machen muß. In der Praxis muß man sich aber im klaren darüber sein, wo Kommunikation entsteht und in welchem Ausmaß. Um Ineffizienzen vorzubeugen erhalten primitive Datenobjekte daher eine Sonderbehandlung, indem sie physisch kopiert werden, während gewöhnliche Datenobjekte immer referenziert werden. Die Slots und der Code der Methoden werden an unterschiedlichen Orten evaluiert und um unnötige Kommunikation beim Ausführen des Codes zu verhindern wurden lokale Methoden eingeführt.

3.3 Ein Beispiel zu dSelf

Im folgenden wird das Beispiel, das in der Einführung zu SELF vorgestellt wurde weitergeführt und in einer verteilten Umgebung demonstriert. Zur Erinnerung werden die aus dem dortigen Beispiel resultierenden Objekte hier noch mal zusammengefaßt:

```
[dSelf-VM2]
stackBehaviour = (|
    parent* = traits clonable.
    push: anObject = (
        topPointer: (topPointer + 1).
        stack at: topPointer Put: anObject.
    ).
    pop = (topPointer: (topPointer - 1)).
    top = (^stack at: topPointer)
|).
stackInstance = (|
    parent* <- stackBehaviour.
    stack = vector cloneSize: 100.
    topPointer <- 0.
|).
boundedStackBehaviour = (|
    parentStack* = stackBehaviour.
    stackLimit = 100.
    push: anObject = (
        top < stackLimit
            ifTrue: [ resend.push: anObject]
            ifFalse: [ 'Stack overflow' printString]
    ).
    pop = (
        top >= 0
            ifTrue: [ resend.pop]
            ifFalse: ['Stack underflow' printString]
    ).
|).
boundedStackInstance = (|
```

```
        parent* <- boundedStackBehaviour.
        stack = vector cloneSize: 100.
        topPointer <- 0.
    |)
```

Angenommen, daß sich diese Objekte auf einer dSelf Virtual Machine mit dem Name *dSelf-VM2* befinden, soll nun versucht werden von einer entfernten dSelf-VM auf diese Objekte zugegriffen zu werden, da dieser Stack dort nicht zur Verfügung steht.

Zunächst muß eine Verbindung zur dSelf-VM2 aufgebaut werden:

```
[dSelf-VM1]
lobby _AddSlot: 'remoteVM' ConnectedTo: '//127.0.0.2/vm'
```

Als nächstes soll nun eine Slot names myStackInstance in der Lobby erzeugt werden, der eine Instanz von boundedStackInstance beinhaltet.

```
[dSelf-VM1]
lobby _AddSlots: (|
    myStackInstance = remoteVM boundedStackInstance clone
|)
```

Hier wurde der Inhalt von boundedStackInstance über die Grenzen hinweg geklont und in myStackInstance abgelegt. Wenn man ein entferntes Objekt klont, werden seine Slots kopiert und lokal neu erzeugt. Die Slotinhalte werden kopiert, wenn sie primitive Objekte sind oder im Fall von gewöhnlichen Objekten wird eine neue Objetreferenz erstellt. Bei Methoden wird der Code 1:1 kopiert, und die lokalen Slots je nach Slotinhalt kopiert oder referenziert.

Der neue Slot kann nun benutzt werden wie jeder andere lokale Slot auch, also zum Beispiel:

```
[dSelf-VM1]
myStackInstance push: 'Hello World'
myStackInstance push: 123
myStackInstance top
-> integer(123)
myStackInstance pop
myStackInstance top
-> string('Hello World')
myStackInstance pop
-> Stack underflow
```

3.4 Benutzerhandbuch für dSelf

Bisher wurde die Sprache dSelf beschrieben. Nun soll ein Blick auf die konkrete Implementierung geworfen werden, also auf die dSelf Vitual Machine und den Compiler.

3.4.1 Die dSelf Virtual Machine

Die dSelf Vitual Machine stellt die Basis des ganzen dSelf-Systems dar. Sie bearbeitet alle Programme die in dSelf erstellt wurden und bietet dem Benutzer die Möglichkeit diese zu manipulieren. Die dSelf-VM bietet zwei verschiedene Benutzerschnittstellen an. Zum einen ein Terminal Front-End, das auf reiner Textbasis funktioniert und zum anderen ein grafisches Front-End, das ein Fenster öffnet.

3.4.1.1 Die Parameter beim Programmaufruf

Wenn man die dSelf Virtual Machine startet, kann man ihr einige Optionen übergeben. Hier ist die Usage:

```
Usage: dSelfVM [options]
Possible options are:
-h        give this help
-v        verbose, give informations when starting
-t        start with terminal front-end
-cr       create registry when not found
-f <name> load script when starting
-b <name> bind the VM to the given name (default: vm)
-r <name> rebind the VM to the given name
-ds       activate debugger of scanner at start
-dc       activate parser-debugger(CUP) at start
-dp       activate parse-tree debugger (with parenthesis)
-di       activate parse-tree debugger (with indentation)
-dl       activate search path debugger (lookup algorithm)
```

Mit der Option "-h" kann man sich diese Auflistung ausgeben lassen und "-v" startet die dSelf-VM im verbose mode. Dieser Modus erzeugt Informationen beim Programmstart über die gesetzten Optionen und die aktuell vom Programm ausgeführten Aktionen, wie zum Beispiel:

```
Name of VM..........................."vm" (bound)
Chosen front-end.....................GUI
Load file............................"smallWorld.dSelf"
debugger of scanner..................disabled
debugger of parser(CUP)..............disabled
Generate flat parse-trees............enabled
Generate indented parse-trees........disabled
Show search path of lookup algorithm..disabled
Try to bind the name of the VM to the RMI-Registry.....OK
Initialize the dSelf-world...........................OK
Generate the GUI.....................................OK
```

Standardmäßig startet die dSelf-VM im Fenstermodus (GUI). Wenn man hingegen im Terminalmodus starten möchte, wird dies durch die Option "-t" erreicht.

Mit der Option "-cr" kann man die dSelf-VM anweisen, selber eine Registry auf dem gleichen Host zu starten, falls keine bereits bestehende gefunden wird. Diese Option sollte man aber nur verwenden, wenn man weiß, daß diese dSelf-VM die einzige ist, die auf diesem Rechner laufen wird, da beim Programmende die Registry wieder gelöscht wird. Wenn man nun eine zweite Virtual Machine startet, dann würde diese die Registry mitbenutzen was zwangsläufig zu Verbindungsfehlern führen würde, wenn die erste dSelf-VM mitsamt der Registry wieder gelöscht wird, während die zweite noch läuft. Wenn mehr als eine dSelf Virtual Machine auf einem Host laufen soll, muß man die RMI-Registry manuell mit "rmiregistry" starten.

Wenn man beim Programmstart direkt ein Skript ausführen möchte, kann man den Namen dieser Datei mit "-f <name>" spezifizieren. Dies führt dazu, daß automatisch ein Befehl " '<name>' _RunScript " erzeugt und ausgeführt wird.

Mit den beiden Optionen "-b <name>" und "-r <name>" kann man den Namen der dSelf-VM spezifizieren. Dieser Name wird benutzt um die Virtual Machine an die lokale RMI-Registry zu binden, damit sie vom Netz aus unter diesem Namen erreichbar ist. Der Unterschied zwischen "-b" und "-r" liegt darin, daß mit "-b" der Name nur an die Registry gebunden wird, wenn dieser dort noch nicht existiert. Andernfalls bricht die dSelf-VM mit einer Fehlermeldung den Startvorgang ab. Es kann aber passieren, daß eine dSelf-VM beim Programmende ihren Namen nicht mehr aus der Registry entfernen konnte, wenn ihr Prozeß zum Beispiel mit kill beendet wurde. In so einem Fall kann man mit "-r <name>" sie dazu zwingen sich unter diesem Namen anzumelden, egal ob er schon existiert oder nicht. Wenn keine der beiden Optionen angegeben wird, wird standardmäßig "dSelfVM" angenommen.

Mit "-ds" wird der Debugger des Scanners aktiviert. Dieser Debugger gibt beim scannen jedes erkannte Token auf dem Bildschirm aus.

"-dc" schaltet den internen Debugger des Parsers ein, der vom Parsergenerator (CUP) zur Verfügung gestellt wurde. Es werden beim Parsieren die aktuellen Aktionen (shift, reduce) sowie die dazugehörigen Zustandsnummern ausgegeben. Es empfiehlt sich zusätzlich mittels _DebugScannerOn"-ds" die aktuell gelesenen Token des Scanners darzustellen, da sonst nur die intern verwendeten Symbolnummern angegeben werden.

"-dp" generiert beim Parsieren einen Strukturbaum in flacher Notation und druckt ihn aus. Mit flach ist hier eine eindimensionale Darstellung gemeint, die die Struktur des Baumes durch Klammern ausdrückt.

"-di" generiert wie "-dp" einen Strukturbaum, gibt ihn aber in einer eingerückten und damit leichter lesbareren Form aus.

"-dl" schaltet den Suchpfaddebugger ein, der beim delergieren von Nachrichten die Suchwege des *lookup algorithm* (Vererbungsalgorithmus von dSelf) angibt.

3.4.1.2 Das Terminal Front-End

Wenn die dSelf Virtual Machine mit diesem Front-End startet, befindet man sich in einer Shell in einem reinen Textmodus und man erhält zum Beispiel folgende Ansicht:

```
Welcome to dSelf !
#dSelf[0]> _Print
lobby: (| systemObjects* = systemObjects. vm* = vm. |)
-> nil
#dSelf[1]> _AddSlots: (|
             x = (| a. b. c |)
           |)
-> lobby
#dSelf[2]> x _Print
an object: (| a <- nil. b <- nil. c <- nil. |) -> nil
#dSelf[3]>
```

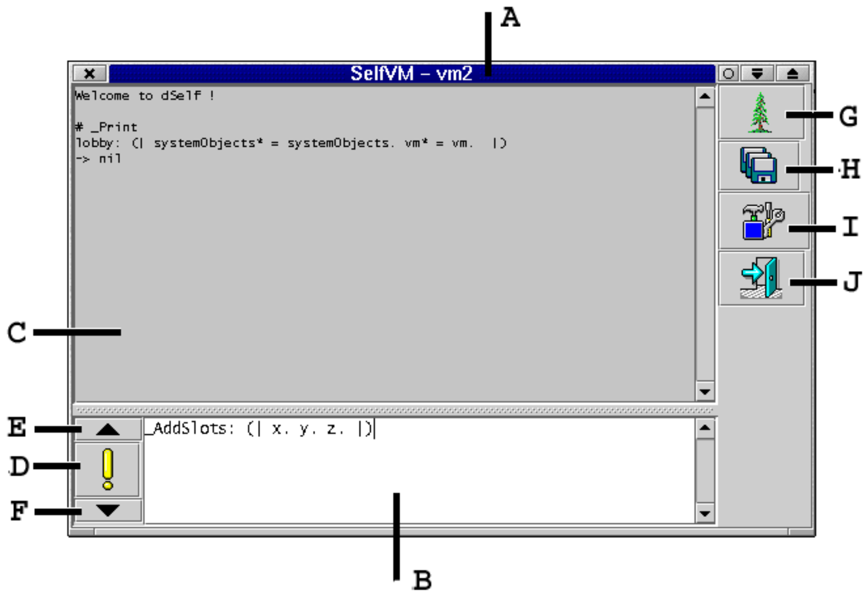
Wie zu sehen ist, erscheint ein Prompt, der auf die Eingaben des Nutzers wartet. Die Eingaben können auch über mehrere Zeilen hinausgehen, da bei der Aktivierung der Eingabetaste immer die Anzahl der offenen, runden Klammern gezählt wird. Sind mehr Klammern geöffnet als geschlossen, dann wird davon ausgegangen, daß die Eingabe noch nicht beendet ist und der Cursor wird in die nächste Zeile direkt hinter dem Prompt positioniert. Es werden nur Klammern gezählt, die eine für die Syntax relevante Bedeutung haben, also keine Klammern in Strings oder Kommentaren.

Die Ergebnisse einer jeden Aktion werden in der nächsten Zeile ausgegeben mit einem davorgesetzten "->".

3.4.1.3 Das GUI Front-End

Die etwas komfortablere Alternative zum Textmodus ist das GUI Front-End. Dieser wird beim Programmstart automatisch aktiviert und stellt dem Benutzer ein Fenster zur Eingabe zur Verfügung. Dieses Fenster wurde in Java unter der Verwendung der Swingbibliothek entwickelt, die auf dem Rechner vorhanden sein müssen (Standard ab Java 1.3).

3.4.1.3.1 Das Hauptfenster Das erste Fenster, das man sieht, ist das Hauptfenster der dSelf-VM, wie es hier abgebildet wird:



Die einzelnen Elemente des Fensters sind hier durch Buchstaben angedeutet. Die genaue Bedeutung dieser Elemente wird im folgenden beschrieben.

[A] Die Titelleiste besteht aus der Beschriftung "dSelfVM <name>", wobei *name* der Name dieser Virtual Machine ist, wie er beim Start spezifiziert wurde.

[B] Das Eingabefeld nimmt die Eingaben des Benutzers entgegen. Im Gegensatz zum Terminal Front-End wird bei der Aktivierung der Eingabetaste die Eingabe nicht ausgewertet und es werden auch keine Klammern gezählt. Die Eingabe wird erst entgegengenommen, wenn der Eingabeknopf (siehe [D]) gedrückt wird.

[C] Das Ausgabefeld zeigt alle Eingaben und Resultate der vorhergehenden Aktionen an. Dabei zur leichtern Lesbarkeit ein "#"-Zeichen vor die Eingabezeilen gesetzt.

[D] Mit dem Eingabeknopf kann man die Eingabe der dSelf Virtual Machine übergeben und die dazugehörigen Aktionen werden ausgelöst. Alternativ kann man auch ALT-Return drücken.

[E] Mit dem nach oben zeigenden Pfeiltknopf kann man zu den vorhergehenden Eingaben zurückblättern. Im GUI Front-End ist eine Historyfunktion eingebaut, die sich die letzten Eingaben merkt.

[F] Dieser Pfeilknopf macht dasselbe wie der Knopf in [E], jedoch in die andere Richtung.

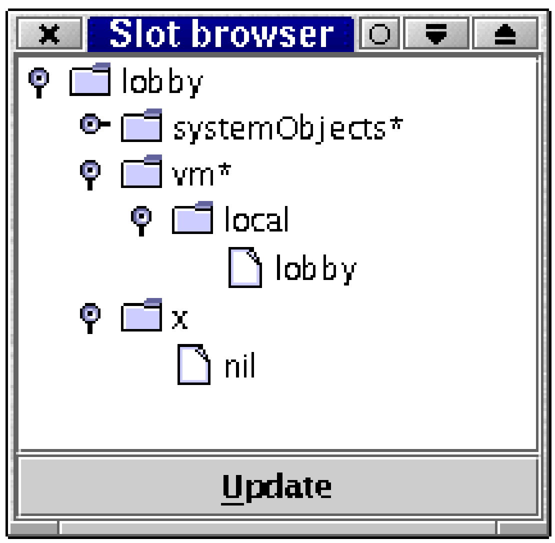
[G] Wenn man diesen Knopf drückt, wird der Slotbrowser aktiviert und angezeigt.

[H] Öffnet den Dialog, mit dem man eine Datei auswählen kann, die in die Virtual Machine hinzugeladen werden soll.

[I] Falls man die vorgegebenen Einstellungen, wie zum Beispiel das Erscheinungsbild der GUI, ändern will, kann man hier das Fenster für solche Einstellungen erreichen.

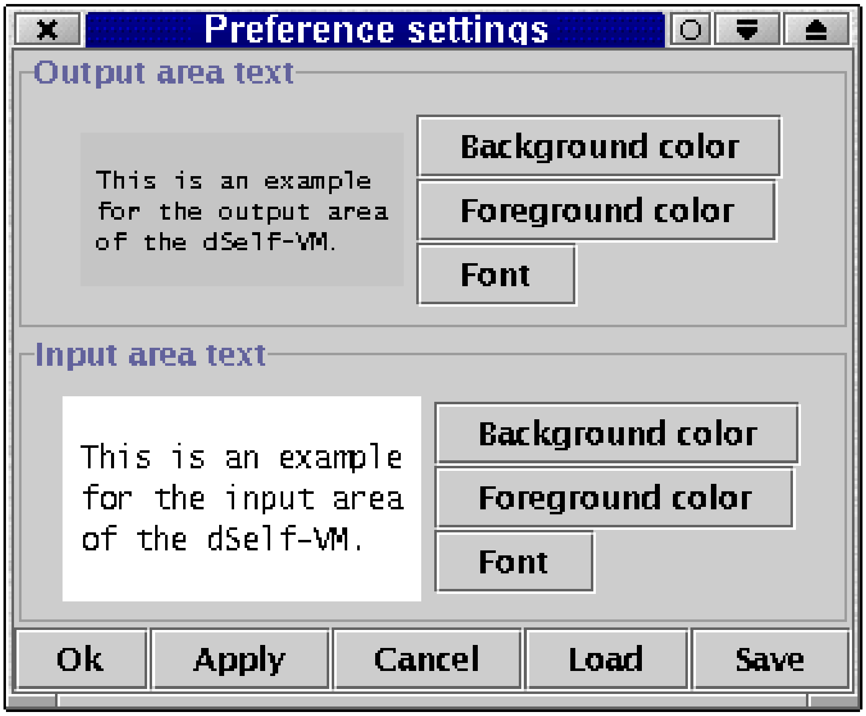
[J] Beendet das Programm

3.4.1.3.2 Der Slotbrowser



Der Slotbrowser stellt die Slots ausgehend von der Lobby als Baum dar. Den Baum durch Klicken auf die Knoten erweitern oder verkleinern. Der Baum wird bei Änderungen an der dSelf-VM nicht automatisch aktualisiert. Dafür gibt es den Updateknopf.

3.4.1.3.3 Das Konfigurationsfenster



Dieses Fenster stellt alle Möglichkeiten zur Konfiguration des Front-Ends zur Verfügung. In der oberen Hälfte kann man das Ausgabefenster konfigurieren und in der unteren Hälfte das Eingabefenster. Die Änderungen werden anhand von Beispieltextfeldern angezeigt, damit man einen optischen Voreindruck bekommt. Ändern lassen sich die Hintergrund- und Vordergrundfarben sowie der Zeichensatz. Wenn man einen der entsprechenden Knöpfe drückt werden neue Dialogfenster geöffnet, die alle möglichen Alternativen zur Verfügung stellen.

In der unteren Reihe stehen die Knöpfe zur Verfügung, die die Änderungen übernehmen oder verwerfen. Sie haben folgende Bedeutung:

[OK] Schließt das Konfigurationsfenster und wendet alle Änderungen auf das Hauptfenster an.

[Apply] Wendet alle Änderungen auf das Hauptfenster an ohne das Konfigurationsfenster zu schließen.

[Cancel] Schließt das Konfigurationsfenster und verwirft alle Änderungen.

[Load] Lädt Konfigurationen aus einer Datei und wendet sie an.

[Save] Speichert die aktuelle Konfiguration in eine Datei.

3.4.2 Der Compiler

Der Compiler von dSelf besitzt zwei verschiedene Schnittstellen. Eine davon wird von der dSelf VirtualMachine aufgerufen und parsiert Eingaben aus einem String. Diese Schnittstelle ist für den Benutzer nicht zugreifbar und wird nur der Vollständigkeit halber hier erwähnt. Die zweite Schnittstelle (main) hingegen macht aus dem Compiler ein eigenständiges Programm, das mit einigen Optionen für den Benutzer verfügbar ist. Der dSelf-Compiler erzeugt aus dSelfdateien Javacode, genauer gesagt eine Javaklasse mit dem gleichen Namen wie die Datei, die den erzeugten Code beinhaltet. Damit eine Javaklasse erzeugt werden kann, muß der Name einem gültigen Javaidentifier entsprechen, sonst bricht der dSelf-Compiler mit einer entsprechenden Fehlermeldung ab.

Die Usage lautet:

```
Usage: dSelfComp [options] <file>
Possible options are:
-h give this help
-r recognizer (don't generate code)
-j invoke java-compiler and generate class file
-jc <name> the name of the java-compiler (default: javac)
-jm <size> set prefered size for methods(default: 10000 Bytes)
-c <path> set the CLASSPATH for java compiler (default: ".")
-ds print debug-informations of scanner
-dc print debug-informations of parser (CUP)
-dp print the parse-tree (with parenthesis)
-di print the parse-tree (with indentation)
```

Mit der Option "-h" kann man sich diese Auflistung ausgeben lassen.

Die Option "-r" veranläßt den Compiler sich wie ein Recognizer zu verhalten. Ein Recognizer erzeugt im Gegensatz zu einem Compiler keinen Code und überprüft ein gegebenes Programm lediglich auf seine syntaktische Korrektheit.

"-j" veranlaßt den dSelf-Compiler nach der Erzeugung der Javaklasse den Compiler von Java aufzurufen um die erzeugte Javaklasse in eine Class-Datei umzuwandeln. Diese Aktion ist standardmäßig nicht aktiviert, da es im Fehlerfall zu Problemen kommen kann. Diese Fehler beziehen sich auf die Ausgaben des Java-Compilers, die vom dSelf-Compiler abgefangen und auf der Konsole ausgegeben werden. Es kann passieren, daß es bei sehr langen Ausgaben zu Deadlocks kommt, so daß im Fehlerfall der dSelf-Compiler ewig auf das Ende des Prozeßes des Java-Compilers wartet. Ob es zu solch einem Fehler kommt ist leider plattformabhängig und somit schwer zu finden, da der Prozeß außerhalb der Java Virtual Machine läuft.

Die Option "-jc" setzt den Namen für den verwendeten Java-Compiler. Standardmäßig ist er auf "javac" gesetzt.

Mit der Option "-jm" kann man die gewünschte Größe für die erzeugten Javamethoden setzen. Standarmäßig werden Methoden mit 10.000 Zeichen erzeugt. Der gewünschte Wert muß zwischen 100 und 65.000 Zeichen liegen. Der Code wird in Methoden zerlegt, die sich gegenseitig der Reihe nach aufrufen, da sonst

Methoden erzeugt werden könnten, die größer als 64 kByte und somit zu groß für die Java Virtual Machine sind.

"-c" spezifiziert den Klassenpfad (CLASSPATH) mit dem der Java-Compiler (Option "-cp") gestartet werden soll.

Mit "-ds" wird der Debugger des Scanners aktiviert. Dieser Debugger gibt beim scannen jedes erkannte Token auf dem Bildschirm aus.

"-dc" schaltet den internen Debugger des Parsers ein, der vom Parsergenerator (CUP) zur Verfügung gestellt wurde. Es werden beim Parsieren die aktuellen Aktionen (shift, reduce) sowie die dazugehörigen Zustandsnummern ausgegeben. Es empfiehlt sich zusätzlich mittels "-ds" die aktuell gelesenen Token des Scanners darzustellen, da sonst nur die intern verwendeten Symbolnummern angegeben werden.

"-dp" generiert beim Parsieren einen Strukturbaum in flacher Notation und druckt ihn aus. Mit flach ist hier eine eindimensionale Darstellung gemeint, die die Struktur des Baumes durch Klammern ausdrückt.

"-di" generiert wie "-dp" einen Strukturbaum, gibt ihn aber in einer eingerückten und damit leichter lesbareren Form aus.

Kapitel 4

Grammatik: Von SELF nach dSelf

In diesem Abschnitt wird dargelegt, wie die Grammatik von SELF, wie sie in [1] beschrieben wird, in die Grammatik von dSelf überführt wurde. Beginnend bei der originalen SELF-Grammatik wird Schritt für Schritt erklärt, warum eine Umstellung von Nöten war und wie sie zustande kam.

4.1 Gründe für Grammatikänderungen

In der Programmierreferenz [1] von SELF wird auf Seite 26 die Grammatik der Sprache formal in EBNF-Notation aufgelistet. Leider ist diese Grammatik so wie sie dort steht nicht zur Verwendung geeignet. So ist sie zum Beispiel leider nicht korrekt, da sie ein nicht erreichbares Nonterminalsymbol definiert (unary-slot), das auf keiner rechten Regelseite auftritt. Desweiteren definiert sie an einer anderen Stelle eine Regel (slot-list), die einer vorherigen, informellen Spezifikation nicht entspricht und in der Implementierung von SELF auch nicht unterstützt wird. Durch die Reperatur dieser Fehler, bei der die fehlerhaften Regeln durch andere, der Spezifikation entsprechende Regeln ersetzt werden, entsteht ein Problem durch Mehrdeutigkeiten. Hiermit sind nicht die grammatikalischen Mehrdeutigkeiten gemeint, die in der Fußnote auf Seite 26 in [1] erwähnt werden, die sich durch Assoziativitäts- oder Vorrangregeln auflösen lassen, sondern sprachliche Mehrdeutigkeiten. Nach ausführlichem Testen der Implementierung stellte sich heraus, daß es in SELF neben Expressions (Ausdrücken) auch noch eingeschränkte Subexpressions gibt, mit deren Hilfe sich diese Mehrdeutigkeiten auflösen lassen. Diese Subexpressions werden in der Dokumentation zu SELF leider nicht erwähnt, sind aber zum Verständnis der Grammatik ein wichtiger Bestandteil und wurden in die dSelf-Grammatik integriert. Im folgenden wird zunächst die Grammatik von SELF, so wie sie in der Programmierreferenz beschrieben wird, aufgelistet und die Umstellungen, die notwendig waren, im Detail beschrieben. Letztendlich enden diese Umstellungen in der Grammatik, wie sie in dSelf verwendet wird.

4.2 Die Grammatik von SELF

expression -> constant | unary-message | binary-message | keyword-message | ‘(’ expression ‘)’

constant -> **self** | number | string | object

unary-message -> receiver unary-send | resend ‘.’ unary-send

unary-send -> identifier

binary-message -> receiver binary-send | resend ‘.’ binary-send

binary-send -> operator expression

keyword-message -> receiver keyword-send | resend ‘.’ keyword-send

keyword-send -> small-keyword expression { cap-keyword expression }

receiver -> [expression]

resend -> **resend** | identifier

object -> regular-object | block

regular-object -> ‘(’ [‘|’ [‘{’ ‘}’ ‘=’ string] slot-list ‘|’] [code] ‘)’

block -> ‘[’ [‘|’ slot-list ‘|’] [code] ‘]’

slot-list -> { unannotated-slot-list | annotated-slot-list }

annotated-slot-list -> ‘{’ string slot-list ‘}’

unannotated-slot-list -> { slot ‘.’} slot [‘.’]

code ->{ expression ‘.’} [‘^’] expression [‘.’]

slot -> arg-slot | data-slot | binary-slot | keyword-slot

arg-slot -> argument-name

data-slot -> slot-name | slot-name ‘<-’ expression | slot-name ‘=’ expression

unary-slot -> slot-name ‘=’ regular-object

binary-slot -> operator ‘=’ regular-object | operator [identifier] ‘=’ regular-object

keyword-slot -> small-keyword {cap-keyword} ‘=’ regular-object | small-keyword identifier {cap-keyword identifier} ‘=’ regular-object

slot-name -> identifier | parent-name

parent-name -> identifier ‘*’

4.3 Die notwendigen Umstellungen

4.3.1 slot-list

Die Regel zur Syntax von Slotlisten in SELF lautet:

slot-list -> { unannotated-slot-list | annotated-slot-list }

Eine Slotliste ist demzufolge eine beliebige Sequenz von *unannotated-slot-list* oder *annotated-slot-list*, also damit auch unter anderem eine Sequenz von *unannotated-slot-list*. Wie eine *unannotated-slot-list* aufgebaut ist, besagt folgende Regel:

unannotated-slot-list -> { slot '.'} slot ['.']

Eine solche Slotliste besteht also aus einer Liste von Slots, die durch Punkte separiert werden, wobei der letzte Punkt optional ist. Die geschweiften Klammern geben an, daß sein Inhalt mehrfach oder gar nicht auftreten kann. Demzufolge kann eine *unannotated-slot-list* auch nur aus dem Nonterminal *slot* bestehen. Aus den Regeln

slot -> arg-slot | data-slot | binary-slot | keyword-slot

data-slot -> slot-name | slot-name '<-' expression | slot-name '=' expression

slot-name -> identifier | parent-name

läßt sich ableiten, daß ein *slot* aus einem einfachen *identifier* (Zur lexikalischen Spezifikation siehe Seite 25 in [1]) bestehen kann, also zum Beispiel x, y oder z. Alle diese Regeln zusammengenommen läßt sich also folgende Aussage herleiten:

slot-list ->* 'x' 'y' 'z'

Eine Slotliste ist demzufolge auch eine Sequenz von Identifiern ohne separierende Punkte, da 'x', 'y' und 'z' jeweils als einzelne Slotlisten interpretiert werden, deren optionaler, abschließender Punkt fehlt. Allerdings findet sich auf Seite 4 in [1] folgende Beschreibung:

> "A slot-list consists of a (possibly empty) sequence (§2.3) separated by periods. A period at the end of the slot list is optional."

Dieser Beschreibung zufolge ist die Regel für *slot-list* offensichtlich falsch. Eine Regel, die die geforderte Syntax definiert ist folgende:

slot-list -> [unannotated-slot-list] {annotated-slot-list [unannotated-slot-list]}

Eine *slot-list* ist jetzt eine Sequenz von *unannotated-slot-list* und *annotated-slot-list*, wobei *unannotated-slot-list* nie zweimal oder öfter hintereinander auftreten kann. Dadurch ist es nicht mehr möglich, daß mehrere einzelne Identifier ohne abschließenden Punkt als einzelne Slotlisten interpretiert werden können, da danach nur eine *annotated-slot-list* oder das Ende der Slotliste auftreten kann. Eine *annotated-slot-list* kann weiterhin mehrfach hintereinander auftreten, was aber aufgrund seiner geklammerten Struktur keine Probleme bereitet.

4.3.2 unary-slot

Das Nonterminalsymbol unary-slot wird in der Grammatik folgenderweise definiert:

unary-slot -> slot-name '=' regular-object

Es beschreibt die Syntax einer unären Methodenslotdeklaration. Das Problem ist, daß es auf keiner rechten Regelseite in der Grammatik erscheint. Damit ist es niemals erreichbar und es wären keine unären Methodenslots deklarierbar. Offensichtlich gehört dieses Nonterminal noch zur folgenden Regel:

slot -> arg-slot | data-slot | binary-slot | keyword-slot

Hier werden alle Arten von Slotdeklarationen aufgelistet und mit der unären Methodenslotdeklaration ergibt sich folgende Regel, die die alte ersetzt:

slot -> arg-slot | data-slot | binary-slot | keyword-slot | unary-slot

Als Konsequenz aus dieser Änderung ergibt sich allerdings ein neues Problem, welches durch folgende zwei Ableitungen aus *slot* ersichtlich ist:

> slot -> unary-slot -> slot-name '=' regular-object -> slot-name '=' '(' code ')' -> slot-name '=' '(' expression ')'

und

> slot -> data-slot -> slot-name '=' expression -> slot-name '=' '(' expression ')'

Es ist also möglich auf zwei verschiedenen Wegen zu

> slot ->* slot-name '=' '(' expression ')'

gelangen. Dies bedeutet, daß die Sprache mehrdeutig ist, da nicht klar ist, ob mit einer Deklaration wie zum Beispiel

```
x = (3+y).
```

eine unäre Methode definiert wird, die beim späteren Aufruf dieser Methode zur Zahl 3 den aktuellen Inhalt eines Slots y addiert, der von der lobby aus erreichbar ist, oder ob ein Datenslot definiert wird, der bei der Erzeugung dieses Slots mit 3+y initialisiert wird. Es muß also einen Weg geben, geklammerte *expressions* und Objektdeklarationen zu unterscheiden. Wie dieser aussieht, wird durch eine Fehlermeldung, die man erhält, wenn man folgende Zeile in der Virtual Machine von SELF eingibt:

```
i _AddSlots: (| x <- (2+3) |)
```

erhält man folgende Fehlermeldung:

sorry - assignable methods not implemented.

während

```
i _AddSlots: (| x <- 2+3 |)
i _AddSlots: (| x <- (2)+3 |)
i _AddSlots: (| x <- 2+(3) |)
```

keine Fehlermeldungen verursachen.

Es wurde bei der ersten Eingabe versucht einem überschreibbaren Datenslot eine geklammerte Expression zuzuweisen, die SELF als ein Methodenobjekt interpretiert hat, während die anderen drei Eingaben als Datenslots, denen ein Ausdruck zugwiesen wird, erkannt wurden. Warum dies so ist, läßt sich aus folgender Eingabe herleiten

```
2+(|x=1| ^x)
```

die diese Fehlermeldung verursacht:

inner methods are no longer supported, slot-list within a sub expression is not legal

Es wurde versucht zur Zahl 2 das Resultat der darauffolgenden Methode zu addieren, also 1. Aus der Fehlermeldung folgt zum einen, daß Methoden innerhalb von Ausdrücken (*inner methods*) einmal erlaubt waren und es jetzt nicht mehr sind und zum anderen, daß es in SELF *sub-expressions* gibt. Diese *sub-expressions* werden in der Dokumentation leider nicht erwähnt und treten in der Grammatik ebenfalls nicht auf, stellen aber ein wichtiges Hilfsmittel dar um zu entscheiden, ob es sich bei einem geklammerten Ausdruck um eine Methodendeklaration oder um eine vom Benutzer angegebene Auswertungsreihenfolge für den Parser handelt. Aus den obigen Beispielen und den dazugehörigen Fehlermeldungen läßt sich herleiten, daß eine Klammerung, die einen ganzen Ausdruck umschließt als ein Methodenobjekt mit leeren Slots und ohne das optionale Circumflex interpretiert. Wenn sich die Klammerung hingegen nur auf einen Teilausdruck bezieht, wird sie nicht als Methode sondern als Vorgabe für die Auswertungsreihenfolge angenommen, da Methoden innerhalb von Ausdrücken nicht erlaubt sind. Um dies in der Grammatik auszudrücken, wird ein neues Nonterminalsymbol namens *subexpression* eingeführt

subexpression -> '(' expression ')' | expression

und die bisherige Regel für Ausdrücke

expression -> constant | unary-message | binary-message | keyword-message |
‘(’ expression ‘)’

wird umgewandelt in

expression -> constant | unary-message | binary-message | keyword-message

Eine *subexpression* interpretiert damit die Klammern immer als eine Vorgabe für die Auswertungsreihenfolge für Teilausdrücke. Als nächstes muß an allen Stellen, wo eine *subexpression* statt einer *expression* erwartet wird, diese ersetzt werden. Dies führt zu folgenden neuen Regeln, die die alten Regeln mit gleichnamigem Nonterminal auf der linken Regelseite ersetzen:

receiver -> [subexpression]

binary-send -> operator subexpression

keyword-send -> small-keyword subexpression { cap-keyword subexpression }

4.3.3 Methoden- und Datenobjekte

Allerdings gibt es immer noch das Problem, daß durch diese Grammatik *inner methods* definiert werden können, die laut der Fehlermeldung in SELF 4.0 nicht möglich sind. Andererseits sind Datenobjekte in Teilausdrücken weiterhin erlaubt. Es muß daher zwischen Daten- und Methodenobjekten unterschieden werden und deshalb wird die Regel

regular-object -> '(' ['|' ['{' '}' '=' string] slot-list '|'] [code] ')'

ersetzt durch die folgenden beiden Regeln

data-object -> '(' ['|' ['{' '}' '=' string] slot-list '|'] ')'

method-object -> '(' ['|' ['{' '}' '=' string] slot-list '|'] code ')'

Methodenobjekte bestehen somit immer aus einer optionalen Slotliste und anschließendem Code, während Datenobjekte nur aus einer Slotliste bestehen oder leer sind. Um diese Regeln verwenden zu können, wird die Regel

object -> regular-object | block

durch

object -> data-object | block

ersetzt.

Diese modifizierten Regeln besagen jetzt, daß alle geklammerten Ausrücke, die an ein Objekt gesendet werden oder selber Empfänger einer Nachricht sind immer Teilausdrücke sind, da sie durch eben diese Nachricht nicht die äußersten Klammern eines Ausdrucks sein können. Wenn sie nicht die äußersten Klammern eines Ausdruckes sind, können sie somit auch keine Methodendeklaration sein, da *inner methods* in SELF nicht (mehr) unterstützt werden. Die einzig möglichen Vorkommen von Methodendeklarationen sind somit die Stellen, an denen Methodenslots definiert werden, also werden folgende Regeln

unary-slot -> slot-name '=' method-object binary-slot -> operator '=' method-object | operator [identifier] '=' method-object

keyword-slot -> small-keyword {cap-keyword} '=' method-object | small-keyword identifier {cap-keyword identifier} '=' method-object

eingeführt, die die alten Regeln mit gleichnamigem Nonterminal auf der linken Regelseite ersetzen.

4.3.4 Directed Resend

In SELF gibt es die Möglichkeit eine Nachricht an die Objekte weiterzudelegieren , von denen geerbt wurde, anstatt sie an das Objekt selber zu schicken. Dies geht ungerichtet (*indirect send*) über das reservierte Wort *resend*, oder gerichtet (*directed send*) indem man den entsprechenden Empfänger angibt. Folgende Regeln definieren die dazugehörige Syntax:

unary-message -> receiver unary-send | resend '.' unary-send

unary-send -> identifier

binary-message -> receiver binary-send | resend '.' binary-send

binary-send -> operator expression

keyword-message -> receiver keyword-send | resend '.' keyword-send

keyword-send -> small-keyword expression { cap-keyword expression }

resend -> resend | identifier

Der Empfänger der Nachricht und die Nachricht selber werden also durch einen Punkt seperiert. Dies führt jedoch zu Mehdeutigkeiten, was folgendes Beispiel zeigt:

```
x.y.z
```

Diese Eingabe kann man zum einen als eine Nachricht y betrachten, die zum Objekt x gesendet wird mit einer folgenden unären Nachricht z an *self*. Andererseits läßt sich dies auch als drei verschiedene Ausdrücke interpretieren, die jeweils eine unäre Nachricht an *self* schicken. Ein Versuch mit der Implementierung von SELF führt zu folgendem Ergebnis:

```
x.y.z
```

führt zur Fehlermeldung:

```
unexpected token, expecting: a name token
```

Somit kann es sich bei der Eingabe nicht um drei verschiedene, unäre Nachrichten handeln, da es sonst keine Fehlermeldung hätte geben dürfen. Wenn man die gleiche Nachricht mit Leerzeichen nach den Punkten eingibt erhält man keine Fehlermeldung und ist damit eine Folge von Ausdrücken. Ebenfalls abgelehnt werden die Eingaben

```
resend . x
resend. x
resend .x
```

mit der Meldung

```
syntax error.
```

während

```
resend.x
```

als ein *directed resend* erkannt wird. Somit wird ein Identifier gefolgt von einem Punkt und einem weiter Identifier, wobei keine *whitespaces* auftreten, immer als ein unärer *resend* erkannt. Gleiches gilt auch für binäre und keyword-Nachrichten, bei denen stattdessen ein Operator oder ein keyword nach dem Punkt folgt. Um dies in der Grammatik auszudrücken, werden folgende Regeln eingeführt, die die obigen Regeln ersetzen:

unary-message -> receiver unary-send | indirected-unary-send | directed-unary-send

unary-send -> identifier

binary-message -> receiver binary-send | indirected-binary-send expression | directed-binary-send expression

binary-send -> operator expression

keyword-message -> receiver keyword-send | indirected-keyword-send expression { cap-keyword expression } | directed-keyword-send expression { cap-keyword expression }

keyword-send -> small-keyword expression { cap-keyword expression }

Die neu eingeführten Terminalsymbole werden vom Scanner übergeben und stellen jeweils das Präfix für eins *(in)directed resend* dar. Da diese nun im Scanner ermittelt werden, haben die Punkte eines *resend* eine höhere Priorität, als die Punkte die Ausdrücke separieren und erst im Parser behandelt werden. Eine genauere Beschreibung der neuen Terminalsymbole folgt im Abschnitt 4.5.1.1 (Makros).

4.3.5 Optionale Slots und Circumflex

Bisher wurde die Grammatik von SELF so umgewandelt, daß zwischen Methodenobjekten und Teilausdrücken unterschieden werden kann. Es wurde gesagt, daß es in SELF keine inner methods gibt und somit alle geklammerten Teilausdrücke als eine *subexpression* behandelt. Folgende Eingaben in der Virtual Machine von SELF hingegen führen zu keiner Fehlermeldung:

```
1 + (| | 2)
1 + (| | ^2)
```

während

```
1 + (| | 2. 1)
```

zur der Fehlermeldung

```
cannot have multiple expressions inside a sub-expression.
```

führt.

Auf Seite 5 in [1] unter Punkt 2.1.4 werden Methoden auf folgender Weise beschrieben:

> "The feature that distinguishes a method object from a data object is that it has code, whereas a data object does not."

Dieser Beschreibung zufolge, müßte es sich bei den drei Eingaben um Methoden handeln, da sie Code enthalten. Offensichtlich werden sie aber nicht als solche erkannt, sondern wie die Fehlermeldung verrät als ein Teilausdruck. Dieser Umstand läßt vermuten, daß der Parser von SELF die optionalen, leeren Slots und das anschließende Circumflex als "syntactic sugar" herausfiltern, da sie optional sind, was dann dazu führt, daß der Parser beim anschließenden Auswerten nur noch einen geklammerten Teilausdruck erkennt, der an dieser Stelle erlaubt ist. Aus Kompatibilitätsgründen wird die Regel

subexpression -> '(' expression ')' | expression

erweitert zu

subexpression -> '(' ['|' '|' ['^']] expression ')' | expression

4.3.6 Parent slot

Ein parent slot besteht aus einem Identifier gefolgt von einem Stern, was durch folgende Regeln zum Ausdruck gebracht wird:

unary-slot -> slot-name '=' method-object slot-name -> identifier | parent-name

parent-name -> identifier '*'

Wie aus diesen Regeln ersichtlich ist, können Parentslots auch auf Methoden verweisen. Ein Test mit SELF führt bei einer Eingabe wie zum Beispiel

```
(| p* = (| x = 1 | ^x) |)
```

zu einem Fehler, da Methoden an dieser Stelle nicht erlaubt sind. Um solch eine Konstruktion zu verhindern, werd die obige Regel für *unary-slot* umgewandelt in:

unary-slot -> identifier '=' method-object

4.3.7 True, False und Nil

Eine weitere Änderung an der Grammatik ist notwendig, um die Objekte *true*, *false* und *nil* in dSelf einzuführen. Diese Objekte sind, im Gegensatz zu SELF, in dSelf primitive Objekte und als Lexeme in der Grammatik fest definiert. Der Grund für diese Änderung liegt in der Verteiltheit von dSelf und wird zu einem späteren Zeitpunkt noch ausführlicher beschrieben. Dadurch wird die Regel

constant -> **self** | number | string | object

erweitert zu

constant -> **self** | number | string | object | **nil** | **true** | **false**

4.4 Die Grammatik von dSelf

In den vorhergehenden Abschnitten wurde die Grammatik SELF Schritt für Schritt umgewandelt. Hier wird die Grammatik, wie sie in dSelf verwendet noch einmal komplett aufgelistet:

expression -> constant | unary-message | binary-message | keyword-message

subexpression -> '(' ['|' '|' ['^']] expression ')' | expression

constant -> **self** | number | string | object | **nil** | **true** | **false**

unary-message -> receiver unary-send | indirected-unary-send | directed-unary-send

unary-send -> identifier

binary-message -> receiver binary-send | indirected-binary-send expression | directed-binary-send expression

binary-send -> operator expression

keyword-message -> receiver keyword-send | indirected-keyword-send expression { cap-keyword expression } | directed-keyword-send expression { cap-keyword expression }

keyword-send -> small-keyword expression { cap-keyword expression }

receiver -> [subexpression]

object -> data-object | block

data-object -> '(' ['|' ['{' '}' '=' string] slot-list '|'] ')'

method-object -> '(' ['|' ['{' '}' '=' string] slot-list '|'] code ')'

block -> '[' ['|' slot-list '|'] [code] ']'

slot-list -> [unannotated-slot-list] {annotated-slot-list [unannotated-slot-list]}

annotated-slot-list -> '{' string slot-list '}'

unannotated-slot-list -> { slot '.'} slot ['.']

code ->{ expression '.'} ['^'] expression ['.']

slot -> arg-slot | data-slot | binary-slot | keyword-slot | unary-slot

arg-slot -> argument-name

data-slot -> slot-name | slot-name '<-' expression | slot-name '=' expression

unary-slot -> identifier'=' method-object

binary-slot -> operator '=' method-object | operator [identifier] '=' method-object

keyword-slot -> small-keyword {cap-keyword} '=' method-object | small-keyword identifier {cap-keyword identifier} '=' method-object

slot-name -> identifier | parent-name

parent-name -> identifier '*'

4.5 Die Grammatik von dSelf in der Implementierung

Die Grammatik im vorhergehenden Abschnitt erzeugt die Sprache, wie sie in dSelf verwendet wird. Zur Implementierung wurden der Scannergenerator JFlex [6] und der Parsergenerator CUP [5] verwendet, die beide Javacode erzeugen. Im folgenden werden diese Grammatiken ohne ihre semantischen Aktionen aufgeführt.

4.5.1 Der Scanner

Der Scanner wurde wie bereits erwähnt unter zuhilfenahme von JFLex erstellt. Die folgende Grammatik besteht aus einem Abschnitt, der alle verwendeten Makros spezifiziert, sowie einem Abschnitt, der alle Token beschreibt, die an den Parser weitergegeben werden. Im Gegensatz zu SELF gibt es in dSelf zusätzlich zu *integer* und *float* noch die Datentypen *short*, *long* und *double*, die durch zusätzliche "_s", "_l" und "_d" Suffixe unterschieden werden.

4.5.1.1 Makros

WhiteSpace = \r | \n | \r\n | \t | " "

SmallLetter = [a-z]

CapLetter = [A-Z]

Digit = [0-9]

Letter = {SmallLetter} | {CapLetter}

GeneralDigit = {Digit} | {Letter}

Base = {Decimal} ("r" | "R")

Decimal = {Digit}+

RadixInteger = "-"? {Base} {GeneralDigit}+

Integer = "-"? {Decimal}

RadixLong = "-"? {Base} {GeneralDigit}+ ("_l" | "_L")

Long = {Integer} ("_l" | "_L")

RadixShort = "-"? {Base} {GeneralDigit}+ ("_s" | "_S")

Short = {Integer} ("_s" | "_S")

Float = "-"? (({Decimal} "." {Decimal}) | ({Decimal} ("." {Decimal})? ("e" | "E") ("+" | "-")? {Decimal}))

Double = {Float} ("_d" | "_D")

Identifier = ({SmallLetter} | "_") ({Letter} | {Digit} | "_")*

SmallKeyword = {Identifier} ":"

CapKeyword = {CapLetter} ({Letter} | {Digit} | "_")* ":"

ArgumentName = ":" {Identifier}

OpChar = "!" | "@" | "#" | "$" | "%" | "&" | "*" | "-" | "+" | "=" | "~" | "/" | "?" | "<" | ">" | "," | ";" | "\\" | "^" | "|"

Operator = {OpChar}+

Comment = \" {CommentChar}* \"

CommentChar = [^\"]

EmptySlots = ("|" {WhiteSpace}* "|") | "||"

EmptyObject = ("(" {WhiteSpace}* ")") | ("(" {EmptySlots} ")")

UnaryIndirectedResend = "resend." {Identifier}

UnaryDirectedResend = {Identifier} "." {Identifier}

BinaryIndirectedResend = "resend." {OpChar}

BinaryDirectedResend = {Identifier} "." {OpChar}

KeywordIndirectedResend = "resend." {CapKeyword}

KeywordDirectedResend = {Identifier} "." {CapKeyword}

4.5.1.2 Token

"'" { -> STRING }

"(" { -> LPAR }

")" { -> RPAR }

"[" { -> LSQBRACK }

"]" { -> RSQBRACK)}

"{" { -> LBRACE }

"}" { -> RBRACE }

"." { -> DOT }

"=" { -> EQUAL }

"^" { -> CIRCFLEX }

"<-" { -> LEFTARROW }

"self" { -> SELF_KEYWORD }

“true” { -> TRUE }

“false” { -> FALSE }

“nil” { -> NIL }

"|" { -> BAR }

```
"*"            { -> ASTERISK }
{Operator} { -> OPERATOR }
{Identifier} { -> IDENTIFIER }
{SmallKeyword} { -> SMALLKEYWORD }
{CapKeyword} { -> CAPKEYWORD }
{ArgumentName} { -> ARGUMENTNAME }
{EmptyObject} { -> EMPTY_OBJECT); }
{RadixInteger} { -> INTEGER }
{Integer}   { -> INTEGER }
{RadixLong} { -> LONG }
{RadixShort} { -> SHORT }
{Float}     { -> FLOAT }
{Double}   { -> DOUBLE }
{UnaryIndirectedResend} { -> UNARYINDIRECTEDRESEND }
{UnaryDirectedResend} { -> UNARYDIRECTEDRESEND }
{BinaryIndirectedResend} { -> BINARYINDIRECTEDRESEND }
{BinaryDirectedResend} { -> BINARYDIRECTEDRESEND }
{KeywordIndirectedResend} { -> KEYWORDINDIRECTEDRESEND }
{KeywordDirectedResend} { -> KEYWORDDIRECTEDRESEND }
{Comment} { /* skip */ }
{WhiteSpace} { /* skip */ }
```

4.5.2 Der Parser

Die folgende Grammatik ist in dieser Form im Parser implementiert, der mit CUP erzeugt wurde. Die zuvor in Abschnitt 4 beschriebene Grammatik war aus Gründen der Lesbarkeit nicht LR(1). CUP braucht eine Grammatik, die LALR(1) ist und deshalb mußten mehrere Umstellungen erfolgen, aber sie ist semantisch äquivalent zu der zuvor beschriebenen Grammatik. Im Gegensatz zu SELF dürfen binäre Operatoren in dSelf unterschiedlich sein (siehe 2.3.2 in [1]), da eine Prüfung auf Äquivalenz von Namen mit kontextfreien Grammatiken nicht ausdrückbar ist.

4.5.2.1 Vorrangregeln

In SELF haben unäre, binäre und keyword-Methoden unterschiedliche Assoziativitäten und Prioritäten, die durch folgende Vorrangregeln ausgedrückt werden:

precedence right SMALLKEYWORD;

precedence right CAPKEYWORD, KEYWORDINDIRECTEDRESEND, KEYWORDDIRECTEDRESEND ;

precedence left OPERATOR, LEFTARROW, EQUAL, ASTERISK, BINARYDIRECTEDRESEND, BINARYINDIRECTEDRESEND;

precedence nonassoc IDENTIFIER, UNARYINDIRECTEDRESEND, UNARYDIRECTEDRESEND ;

Unäre Methoden haben die größte Priorität, gefolgt von den binären Operatoren und schließlich den keyword-Nachrichten. Operatoren sind linksassoziativ und keyword-Nachrichten rechtsassoziativ.

Die Terminalsymbole LEFTARROW, EQUAL und ASTERISK werden vom Scanner als einzelne Vorkommen von '<-', '=' und '*' nicht als Operator übergeben, da sie bei der Slotdeklaration benötigt werden. Sie können aber auch als normale Operatoren an anderen Stellen verwendet werden und müssen daher auch die gleiche Priorität und Assoziativität besitzen.

4.5.2.2 Die Grammatik

start_ ::=

expr

| exprdotlist

| exprdotlist expr

|

;

subexpr ::=

LPAR expr RPAR

| expr

| LPAR CIRCFLEX expr RPAR

| LPAR BAR BAR expr RPAR

| LPAR BAR BAR CIRCFLEX expr RPAR

;

expr ::=

unary_message
| binary_message
| keyword_message
| SELF_KEYWORD
| STRING
| block
| data_object
| INTEGER
| FLOAT
| LONG
| SHORT
| DOUBLE
| TRUE
| FALSE
| NIL
;

unary_message ::=

IDENTIFIER
| subexpr IDENTIFIER
| UNARYDIRECTEDRESEND
| UNARYINDIRECTEDRESEND
;

binary_message ::=

operator subexpr
| subexpr1 operator subexpr2
| BINARYDIRECTEDRESEND subexpr
| BINARYINDIRECTEDRESEND subexpr
;

keyword_message ::=

SMALLKEYWORD subexpr capkeywordexprlist
| subexpr1 SMALLKEYWORD subexpr2 capkeywordexprlist
| KEYWORDDIRECTEDRESEND subexpr capkeywordexprlist
| KEYWORDINDIRECTEDRESEND subexpr capkeywordexprlist
;

capkeywordexprlist ::=

capkeywordexprlist CAPKEYWORDk subexpr
|
;

data_object ::=

LPAR BAR objectslotdecl BAR RPAR
| EMPTY_OBJECT
;

method_object[1] ::=

LPAR BAR objectslotdecl BAR exprdotlist CIRCFLEX expr DOT RPAR
| LPAR BAR objectslotdecl BAR exprdotlist CIRCFLEX expr RPAR
| LPAR BAR objectslotdecl BAR CIRCFLEX expr DOT RPAR
| LPAR BAR objectslotdecl BAR CIRCFLEX expr RPAR
| LPAR BAR objectslotdecl BAR exprdotlist expr RPAR
| LPAR BAR objectslotdecl BAR exprdotlist RPAR
| LPAR BAR objectslotdecl BAR expr RPAR
| LPAR BAR BAR exprdotlist CIRCFLEX expr DOT RPAR
| LPAR BAR BAR exprdotlist CIRCFLEX expr RPAR
| LPAR BAR BAR CIRCFLEX expr DOT RPAR
| LPAR BAR BAR CIRCFLEX expr RPAR
| LPAR BAR BAR exprdotlist expr RPAR
| LPAR BAR BAR exprdotlist RPAR
| LPAR BAR BAR expr RPAR
| LPAR exprdotlist CIRCFLEX expr DOT RPAR
| LPAR exprdotlist CIRCFLEX expr RPAR
| LPAR CIRCFLEX expr DOT RPAR
| LPAR CIRCFLEX expr RPAR

[1] *Die Code-Regeln für method_ object lassen sich leider nicht durch zusätzliche Nonterminals kürzer und leserlicher gestalten, da sonst der Lookahead nicht bis zum Ende der Regel gelangt und dadurch Konflikte entstehen. Zum Beispiel: | x = ((2 * 3) - 5) + 3 |. Bis die binäre Methode "+" im Lookahead auftaucht ist nicht klar, ob x = ((2 * 3) - 5) die unäre Methodendeklaration x = (| | ^(2 * 3) - 5) oder wie oben das Präfix eines Datenslots mit einer beginnenden Sub-Expression sein soll. Dadurch muß bis zum "+" geshiftet werden. Daher sind Zusammenfassungen mit Hilfs-Nonterminalen nicht möglich, da sonst unweigerlich Konflikte (shift/reduce) mit eben diesem Nonterminalen auftreten müssen.*

| LPAR exprdotlist expr RPAR
| LPAR exprdotlist RPAR
| LPAR expr RPAR
;

objectslotdecl ::=

LSQBRACK RSQBRACK EQUAL STRING slotlistl
| slotlist
;

block ::=

LBRACE blockslotdecl exprdotlist CIRCFLEX expr DOT RBRACE
| LBRACE blockslotdecl CIRCFLEX expr DOT RBRACE
| LBRACE blockslotdecl exprdotlist CIRCFLEX expr RBRACE
| LBRACE blockslotdecl CIRCFLEX expr RBRACE
| LBRACE blockslotdecl exprdotlist expr RBRACE
| LBRACE blockslotdecl exprdotlist RBRACE
| LBRACE blockslotdecl expr RBRACE
| LBRACE blockslotdecl RBRACE
;

blockslotdecl ::=

BAR slotlist BAR
| BAR BAR
|
;

slotlist ::=

unannotated_slot_list slotlist2
| annotated_slot_list slotlist2
| annotated_slot_list unannotated_slot_list slotlist2
;

slotlist2 ::=

annotated_slot_list slotlist2
| annotated_slot_list unannotated_slot_list slotlist2
|
;

```
annotated_slot_list ::=
        LSQBRACK STRING slotlistl RSQBRACK
        ;
unannotated_slot_list ::=
        slotdotlist slot
        | slotdotlist
        | slot
        ;
slotdotlist ::=
        slotdotlist slotdot
        | slot dot
        ;
slotdot ::=
        slotdot
        | error DOT // Bei Fehler bis zum nächsten Punkt springen
        ;
exprdotlist ::=
        exprdotlist exprdot
        | exprdot
        ;
        exprdot ::=
        exprdot
        | error DOT // Bei Fehler bis nächsten Punkt überlesen
        ;
slot ::=
        arg_slot
        | data_slot
        | binary_slot
        | keyword_slot
        | unary_slot
        ;
```

arg_slot ::=

ARGUMENTNAME

;

data_slot ::=

IDENTIFIER

| IDENTIFIER EQUAL expr

| IDENTIFIER LEFTARROW expr

| IDENTIFIER ASTERISK

| IDENTIFIER ASTERISK EQUAL expr

| IDENTIFIER ASTERISK LEFTARROW expr

;

unary_slot ::=

identifier EQUAL method_object

;

binary_slot ::=

operator EQUAL method_object

| operator IDENTIFIER EQUAL method_object

;

keyword_slot ::=

SMALLKEYWORD capkeywordlist EQUAL method_object

| SMALLKEYWORD IDENTIFIER capkeywordlist EQUAL method_object

;

capkeywordlist ::=

capkeywordlistl CAPKEYWORD

|

;

operator ::=

OPERATOR

| LEFTARROW

| EQUAL

| ASTERISK

;

4.6 Abschließende Bemerkung

Wie in den vorhergehenden Abschnitten zu sehen war, wurden einige Umstellungen an der ursprünglichen Grammatik vorgenommen. Einige Änderungen resultieren aus den Hinweisen, die man zum Beispiel den Fehlermeldungen der Implementierung von Sun entnehmen konnte. Daher kann nicht sicher gesagt werden, ob die Grammatik, die dSelf zu Grunde liegt auch wirklich zu 100 Prozent kompatibel mit der von SELF ist. Unter anderem ist dSelf deshalb als ein Derivat zur Programmiersprache SELF zu betrachten und nicht als eine exakte Kopie.

Kapitel 5

Die Implementierung von dSelf

Nachdem in den vorhergehenden Kapiteln die Sprache dSelf von der Benutzerseite her beschrieben wurde, soll nun ein Blick auf die konkrete Implementierung geworfen werden. Zuerst werden die grundsätzlichen Entscheidungen beschrieben, die getroffen werden mußten, wie zum Beipiel die Wahl eines Interpreters oder Compilers. Danach erfolgt eine Beschreibung des Compilers und der dSelf Virtual Machine, wie sie implementiert wurden. Anschließend werden die wichtigsten Klassen des Systems in Klassendiagrammen präsentiert und die einzelnen Klassen beschrieben und abschließend in der Darstellung der wichtigsten Verfahren ihr Zusammenspiel dargestellt.

5.1 Prinzipielle Realisierungsansätze

Wenn man eine Sprache wie dSelf implementieren will, stellt sich zunächst die Frage, ob man einen Compiler oder einen Interpreter als Grundlage benutzt. Der Vorteil eines Compilers liegt in der Ausführgeschwindigkeit des Codes, während Interpreter dafür flexibler in ihren Möglichkeiten sind. Im folgenden sollen die einzelnen Vor- und Nachteile beider Ansätze im Bezug auf dSelf untersucht werden.

5.1.1 Klassischer Interpreter

Mit "klassisch" ist hier ein reiner Interpreter gemeint, der seinen Code aus einem Text ausliest, parsiert und anschließend die jeweiligen Aktionen direkt ausführt und nicht modernere Ansätze wie Just In Time Compiler oder Bytecodeinterpreter, die bereits vorcompilierten Code auswerten. Dieser Ansatz bietet zunächst einmal den Vorteil seiner Einfachheit. Er besteht im Vergleich zu einem Compiler nur aus dessen syntaktischer Analysephase und es besteht kein Bedarf mittels einer anschließenden Synthesephase Code zu erzeugen. Ein weiterer Vorteil, speziell im Bezug auf dSelf und seine verteilten Methoden, ist die einfache Serialisierung von Strings. Wenn ein entferntes Objekt von einer anderen dSelf Virtual Machine eine Methode anfordert, so müßte man lediglich ihren Quelltext transportieren, und da Zeichenketten in Java serialisierbar sind, wird dies automatisch erledigt. Ein weiterer Vorteil liegt in der Flexibilität. Ein Interpreter ist nicht auf die Möglichkeiten eines von ihm erzeugten, eher statischen Code beschränkt. Gerade im Kontext der interaktiven dSelf Virtual Machine ist diese Flexibilität notwendig.

Es gibt aber auch entscheidende Nachteile. In erster Linie ist dies die Langsamkeit von Interpretern. Da ein reiner Interpreter den Code immer wieder neu parsieren muß, entsteht hier ein großer Aufwand. In Hinsicht auf Methoden ist dieser Aufwand allerdings überflüssig, da Methoden in dSelf nicht mehr geändert werden können. Somit wird hier eine mögliche Flexibilität durch wiederholtes Auswerten des Quellcodes nie benötigt und ist daher redundant. Desweiteren sind Interpreter sehr fehlerträchtig im bezug auf syntaktische Fehler. Da eine syntaktische Analyse erst dann stattfindet, wenn eine bestimmte Stelle im Quelltext ausgeführt werden soll, wird auch erst zu diesem Zeitpunkt ein Fehler gefunden. Um mit Bestimmtheit sagen zu können, daß ein Programm syntaktisch korrekt ist, müssen alle Methoden aufgerufen werden. Speziell bei dSelf entsteht dadurch noch ein zusätzliches Problem, das zu inkonsistenten Zuständen in der dSelf Virtual Machine führen kann.

Hierzu ein Beispiel:

```
_AddSlots: (| objectParent. objectInstance |)
objectParent _Define: (|
    counter <- 0
    dataVec  = vector cloneSize: 100
|)
objectInstance _Define: (|
    parent* = objectParent.
    parentClone* = traits clonable.
    addYourself = (
        ... “some Code” ...
        counter: counter + 1.
        dataVec At: counter Put: self.
        ... “some Code” ...
    )
|)
```

Hier werden zwei Objekte *objectParent* und *objectInstance* definiert. *objectParent* besitzt einen Zähler namens *counter* und einen Vektor *dataVec*. Diese beiden Slots sollen für alle Instanzen zugreifbar sein, sind also mit Klassenvariblen in klassenbasierten Sprachen vergleichbar. *objectInstance* stellt den Prototypen für instanziebare Objekte dar und erbt über seinen Parentslot *parent* von *objectParent* und über *parentClone* von *traits clonable*. addYourself ist nur ansatzweise dargestellt und addiert sich selbst zum geerbten Vektor *dataVec* hinzu.

Dieses Programm macht auf den ersten, flüchtigen Blick einen korrekten Eindruck, doch bei der Ausführung von *addYourself* wird es beim Aufruf der Nachricht *At:Put:* zu einem Syntaxfehler kommen, da Keyword-Nachrichten immer mit einem kleinen Buchstaben beginnen müssen. Es hätte also *at:Put:* heißen müssen. Die Konsequenz bei einem Interpreter wäre nun ein Abbruch der ausgeführten Aktion mit einer entsprechenden Fehlermeldung nachdem der Zähler um eins erhöht wurde, und genau hier liegt das Problem. Da *counter* erhöht wurde, aber aufgrund des Fehlers kein Element hinzugefügt wurde, ist an dieser Stelle nun ein undefiniertes Objekt vorhanden. Da andere von *object-Parent* abgeleitete Objekte noch auf diesen Daten arbeiten könnten hat diese Inkonsistenz weitreichende Konsequenzen.

Da syntaktische Fehler wie der obige häufig vorkommen und somit inkonsistente Zustände in Self schnell eintreten können ist eine reine Interpreterlösung nicht nur aus Effiziengründen nicht als ideal anzusehen.

5.1.2 Klassischer Compiler

Mit "klassisch" ist hier ein Compiler gemeint, der einen Quelltext einliest und eine ausführbare Datei erzeugt. Im Fall von Java wäre dies eine Klasse, die eine statische main-Methode anbietet. Der Vorteil dieses Ansatzes liegt in der Effizienz des ausführbaren Codes, da im Gegensatz zu einem Interpreter der Quelltext nur ein einiges Mal parsiert werden muß. Dabei wird der komplette Quelltext syntaktisch analysiert und alle Syntaxfehler im vorhinein gefunden. Daher kann es nicht so schnell zu Inkonsistenzen kommen, wie sie im Beispiel aus dem vorhergehenden Abschnitt dargestellt wurden. Der Nachteil von Compilerlösungen liegt aber in ihrer zur Interpreterlösung hohen Statik, da man den erzeugten Code im nachhinein gar nicht oder nur mit hohem Aufwand ändern kann. Dies wiederspricht der Interaktivität der dSelf Virtual Machine und damit hohen Dynamik. In der dSelf-VM kann der Benutzer zum Beispiel jederzeit per *_AddSlots:* neue Methoden zum System hinzufügen oder mit *_RemoveSlot:* löschen. Bei einer reinen Compilerlösung müßte man nun aus einer Eingabe des Benutzers diese zunächst compilieren und Javacode erzeugen. Dieser würde dann durch den Java-Compiler in eine Class-Datei umgewandelt, die man dann einladen und dynamisch zum laufenden System hinzu linken müßte um sie anschließend auszuführen. Diese Prozedur würde lange dauern und es ist sicher nicht im Sinne des Benutzers, wenn er bei jeder Eingabe einige Sekunden auf das Ergebnis warten muß.

5.1.3 Lösung in dSelf

Die Lösung, die in dSelf realisiert wurde stellt gewissermaßen eine Mischung aus Interpreter und Compiler dar. Um dies zu realisieren stellt der Compiler von dSelf zwei Schnittstellen zur Verfügung. Eine ist exklusiv für die dSelf Virtual Machine und parsiert aus einem gegebenen String den Quelltext und gibt als Ergebnis ein Expressionobjekt zurück, daß durch den Aufruf der Methode eval evaluiert wird. Diese Art von Objekten können sich selbst evaluieren, stellen also in gewisser Weise einen Interpreter dar, der den parsierten Ausdruck bei syntaktischer Korrektheit verzögert ausführt. Dieser Ansatz bietet den Vorteil, daß er zum einen weniger fehlerträchtig ist als ein reiner Interpreter, da zuerst die Syntax überprüft wird, und zum anderen flexibler als ein reiner Compiler ist, da die erzeugten Codeobjekte ohne Umwege direkt ausführbar sind. Wie diese Codeobjekte im Detail aussehen, wird zu einem späteren Zeitpunkt dargelegt.

Die zweite Schnittstelle ist für den Benutzer gedacht und macht den Compiler über die statische Methode main zu einem eigenständigen Programm. Dieser Compiler erzeugt aus einem gegebenen dSelf-Skript eine Javaklasse, die vom Java-Compiler übersetzt werden kann. Wenn man nun in der dSelf Virtual Machine dieses Skript einladen will, wird zunächst geprüft, ob es eine gleichnamige Class-Datei mit einem neueren Datum existiert. Wenn ja, dann wird diese dynamisch zum laufenden System hinzu gelinkt und ausgeführt. Falls nicht, wird das dSelf-Skript eingeladen, dann zum parsieren an den Compiler weitergegeben und letztendlich das resultierende Expression-Objekt ausgewertet. Prinzipiell sind vorcompilierte Class-Dateien also nicht notwendig, da die dSelf-Skripte ausreichen. Sie stellen daher eine Optimierung dar, die ein wiederholtes parsieren von dSelf-Skripten unnötig machen. Dies ist insbesondere von Bedeutung, wenn man die dSelf Virtual Machine immer mit einem dSelf-Skript lädt, das eine neue dSelf-Welt erstellt, wie dies zum Beispiel in SELF auch gemacht wird. So eine Welt kann sehr groß werden und bräuchte bei jedem Systemstart viel Zeit, da der Quelltext jedesmal neu übersetzt werden müßte und das für jede dSelf-VM. Das reine hinzulinken und ausführen einer vorcompilierten Datei geht dabei wesentlich schneller.

5.2 Der Compiler und die Virtual Machine

Das dSelf-System besteht aus zwei Komponenten. Die erste Komponente stellt der Compiler dar, der alle Eingaben parsiert und in ausfübaren Code umwandelt. Die Aufgabe der dSelf-VM ist im wesentlichen das Entgegennehmen von Benutzereingaben, die sie an den Compiler weiterreicht, und die Darstellung der Ergebnisse.

5.2.1 Der Compiler

Der Compiler wurde unter Zuhilfenahme von JFlex [6] und CUP [5]erstellt. JFlex ist ein Scannergenerator, der die Lexeme eines dSelf-Programms ermittelt und als Token an den Parser weiterreicht. Im folgenden wird zwischen dem Parser und dem eigentlichen Compiler unterschieden. Der Parser stellt den Teil dar, der für die syntaktische Analyse und die Synthese verantwortlich ist, während der Compiler als das gesamte Programm verstanden wird, dessen Teil der Parser ist.

5.2.1.1 Der Parser

Der Parser wurde mit Hilfe von CUP erstellt. CUP ist ein Parsergenerator, der auf der Basis von LALR(1)-Grammatiken einen Parser erstellt, der die vom Scanner erhaltenen Token auf ihre syntaktische Korrektheit hin untersucht. Die zugrundeliegenden Grammatiken wurden bereits im Kapitel über die Grammatikumwandlung von SELF nach dSelf vogestellt.

Der Parser erzeugt bei jedem Parserschritt als semantische Aktion ein Tripel namens *Result*:

Result

`+code: Object`
`+tree: String`
`+javaCode: JavaCodeTree`

`+Result(newCode:Object,newTree:String,newJavaCode:JavaCodeTree)`

Dieses Objekt besteht aus einem *Object* names *code*, das einem Expression-Objekt entspricht, das bei dem jeweiligen Parserschritt erzeugt wurde. Diese Objekte werden von der dSelf Virtual Machine benötigt, die diese ausführt.

Die Variable tree beinhaltet einen String, der den Strukturbaum des parsierten dSelf-Skripts beinhaltet. Er wird erzeugt, wenn der Benutzer den Strukturbaum-Debugger aktiviert hat und die Auswertungsreihenfolge seines Codes einsehen möchte.

javaCode ist vom Typ JavaCodeTree und beinhaltet den Javacode, der dem parsierten Programm entspricht. JavaCodeTree besteht aus einem Baum an dessen Knoten Strings mit Javacode hängen.

Ob eines der gewünschten Objekte erzeugt wird oder nicht hängt von den Einstellungen des Systems ab, die vor der Generierung abgefragt werden. Jede semantische Aktion im Parser hat folgende Struktur:

```
RESULT =
    Globals.errorOccured ?
        null : new Result(
            Globals.genCode ?
                <expression> : null
            ,Globals.debug ?
                "<strukturBaum>" : ""
            ,Globals.genJava ?
                new JavaCodeTree("<JavaCode>") : null
        );
```

RESULT ist eine reservierte Variable von CUP, die immer den Rückgabewert eines Parserschritts beinhaltet. Zunächst wird geprüft ob während einer der vorhergehenden Schritte ein Syntaxfehler aufgetreten ist. Wenn ja, macht es keinen Sinn mehr weiter neue Objekte zu erzeugen, da diese kein brauchbares

Ergebnis liefern würden. Der Grund, daß der Parser dann überhaupt noch weiterarbeitet, ist weitere Fehler zu finden und anzuzeigen, jedoch nicht mehr brauchbaren Code zu erzeugen. Falls kein Fehler auftrat, wird zunächst geprüft, ob ein ausfühbares Expression-Objekt erzeugt werden soll. Dies ist immer nur dann der Fall, wenn der Compiler von der dSelf Virtual Machine aufgerufen wurde.

Als nächstes wird ein String mit dem Strukturbaum erzeugt, wenn der Benutzer den Debugger aktiviert hat. Bei einem auftretenden Fehler wird auch dieser nicht weitererzeugt, da der Parser in so einem Fall eine sogenannte error recovery einleitet. Die besteht darin, daß er versucht so lange die nächsten Token zu shiften, bis er einen Punkt findet, an dem er wieder aufsetzen kann. Dies ist bei dSelf zum Beispiel ein Punkttoken ".", das in der Regel einen neuen Ausdruck einleitet. Durch diese Aktion geht jedoch die gültige Struktur des Baumes verloren und daher ist der Strukturbaumdebugger eher zum Verständnis der syntaktischen Auswertungsreihenfolge korrekter Ausdrücke gedacht.

Als letztes wird noch überprüft, ob Javacode erzeugt werden soll. Dies ist zum Beispiel nicht der Fall, wenn der Aufruf von der dSelf-VM kam oder der Compiler angewiesen wurde, sich wie ein Recognizer zu verhalten. Als nächstes soll die Klasse JavaCodeTree etwas genauer betrachtet werden.

JavaCodeTree
#nodes: Object[] #pos: int = 0 #methods: String[] = new String[100]
+JavaCodeTree(o:Object) +JavaCodeTree(o1:Object,o2:Object) +JavaCodeTree(o1:Object,o2:Object,o3:Object) +JavaCodeTree(o1:Object,o2:Object,o3:Object,o4:Object) +JavaCodeTree(o1:Object,o2:Object,o3:Object,o4:Object,o5:Object) +JavaCodeTree(o1:Object,o2:Object,o3:Object,o4:Object,o5:Object,o6:Object) +JavaCodeTree(o1:Object,o2:Object,o3:Object,o4:Object,05:Object,06:Object,07:Object) +JavaCodeTree(o1:Object,o2:Object,o3:Object,o4:Object,o5:Object,o6:Object,o7:Object,o8:Object,o9:Object) +size(): int +getJavaClass(className:String,javaMethodSize:int): String #toCodeString(length:int): String

Wie man dem Diagramm entnehmen kann, besteht es aus einem Array vom Typ *Object*, das die Knoten des Baumes repräsentiert und einer Anzahl von Konstruktoren, die mit den übergebenen Argumenten einen Unterbaum erzeugen. Die Knoten sind wiederum vom Typ *JavaCodeTree* während die Blätter vom Typ *String* sind und Teilausdrücke des *Javacodes* beinhalten. *size* gibt die Größe dieses Baumes an, die aus der Anzahl aller Zeichen der Strings an den Blättern besteht. Die wichtigste Methode ist *getJavaClass*. Wenn man sie aufruft, gibt sie einen *String* zurück, der den Code einer Javaklasse mit dem angegebenen Namen beinhaltet. Als zusätzliches Argument wird ein Integer übergeben, der die gewünschte Methodengröße beinhaltet. Mit Größe ist hier die Anzahl der Zeichen gemeint, aus denen sie bestehen soll. Diese Zahl ist nicht als eine exakte Vorgabe zu verstehen , sondern als ein Richtwert mit Abweichungen von ca. +/- 50 Zeichen.

Der eigentliche Existenzgrund für *JavaCodeTree* liegt zum einen an der Effizienz und zum anderen an den Einschränkungen der Java-VM. Eine übliche

Java-VM kann nämlich nur Methoden bis zu einer Größe von 64 kByte bearabeiten, aber der erzeugte Code kann "am Stück" deutlich länger werden. Daher muß der erzeugte Ausdruck in Teilausdrücke zerlegt werden. Um dies zu erreichen fügt *JavaCodeTree* einfach so viele Teilausdrücke aus den einzelnen Parserschritten aneinander, bis die Länge ungefähr mit dem gewünschten Wert übereinstimmt. Anschließend wird als letzter Ausdruck ein Methodenaufruf eingefügt, der die nächsten Teilausdrücke beinhaltet, usw.. Der erzeugte Code hat dann in etwa folgende Struktur:

```
import dSelf.*;
import parser.*;
public class smallWorld extends dSelfProgram {
  public CodeExpr getExpression(){
    return (CodeExpr)(getObjn());
  }
  Object getObj0(){
    return (Expr(Expr(...(getObj1())...)));
  }
  Object getObj1(){
    return (Expr(Expr(...(getObj2())...)));
  }

  ...
  Object getObjn(){
    return (Expr(Expr(...)));
  }
}
```

Die erzeugte Klasse erbt von *dSelfProgram* die Schnittstelle *getExpression*, die die erzeugten Expression-Objekte zurückgibt.

5.2.1.2 Der eigentliche Compiler

Der Compiler, wie ihn der Benutzer oder die dSelf-VM zu sehen bekommen hat folgende Schnittstellen:

dSelfComp

```
#javac: String = "javac"
#classpath: String = "."
#javaCompile: boolean = false
#javaMethodSize: int = 10000
+calledByVM: boolean
---
+main(args:String[])
+parseString(parseStr:String): CodeExpr
+parseFile(fileName:String): CodeExpr
#parse(parser:dSelfParser): Result
-isCorrectName(name:String)
#genJavaClass(name:String,codeTree:JavaCodeTree)
+reportError(msg:String)
+reportError(msg:String,line:int,col:int)
#checkArgs(args:String[]): String
#printUsage()
#printParseTree(flat_parse_tree:String)
```

Im String *javac* ist der Name des gewünschten Java-Compilers enthalten, falls dieser automatisch aufgerufen werden soll. Er wird mit dem Klassenpfad *classpath* aufgerufen, wenn *javaCompile* gleich *true* ist. javaMethodSize spezifiziert die gewünschte Javacodegröße und *calledByVM* gibt an, ob der Aufruf von der dSelf-VM kam. Die *main*-Methode bietet die Schnittstelle zum Benutzer als ein eigenständiges Programm und übernimmt die Argumente, die beim Programmstart angegeben wurden. Diese Argumente werden von *checkArgs* ausgewertet und können mittels *printUsage* dem Benutzer angezeigt werden. *parseString* stellt die Schnittstelle zur dSelf-VM dar und übernimmt die Eingabe in einem *String*. Kam der Aufruf jedoch vom Benutzer, so wird stattdessen *parseFile* aufgerufen, mit dem Namen der Datei, die compiliert werden soll. Der resultierende Javacode wird mit *genJavaClass* erzeugt, der den Namen der Klasse und das Resultat des Parsers dafür benötigt. Der Name muß den lexikalischen Anforderungen eines Java-Identifiers entsprechen, die mit *isCorrectName* überprüft werden. Wenn dabei der Strukturbaum ausgegeben werden soll, wird *printParseTree* aufgerufen. Diese Methode übernimmt den Baum in einer geklammerten Form und stellt ihn in einer optisch ansprechenderer Form mit Einrückungen dar. Sollte ein Fehler auftreten, kann man ihn mit *reportError* ausgeben lassen, mit der jeweiligen Nachricht und - falls vorhanden - der genauen Position im Quelltext.

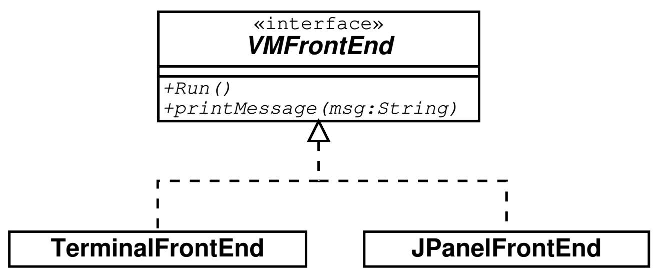
5.2.2 Die Virtual Machine

Die Virtual Machine von dSelf stellt im wesentlichen die Schnittstelle zum Benutzer dar. Ihre Aufgabe ist es, Eingaben aufzunehmen und an den Parser weiterzureichen, sowie dessen Ergebnis an den Benutzer weiterzugeben. Die dSelf-VM hat folgende Schnittstellen:

dSelfVM
#dirPath: String = "" #frontEnd: VMFrontEnd +VMName: String = "vm" +server: Server +remoteVMs: RemoteVMs #rebindName: boolean = false #verboseMode: boolean = false #createRegistry: boolean = false +lobbySO: LocalOrdinarySO
+main(String[]:args) +exit() #initializeSelfWorld() +evalInput(input:String): String +loadScriptDirPath(fileName:String): CodeExpr +loadScript(fileName:String): CodeExpr +loadFromClass(fileName:String): CodeExpr +setDirPath(dir:String): String +getDirPath(): String +printCompiletimeError(errMsg:String) +printCompiletimeError(errMsg:String,line:int,col:int) +printRuntimeError(errMsg:String) +printMessage(msg:String) +printWarning(msg:String) +printUsage() +checkArgs(args:String[])

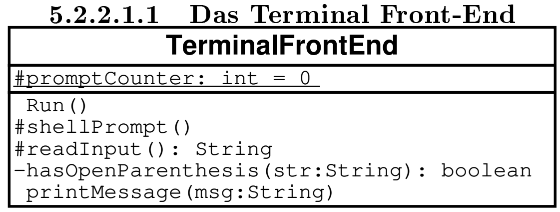
Gestartet wird die dSelf Virtual Machine über die Methode main. Sie übernimmt die Argumente, die beim Programmstart angegeben wurden. Diese Argumente werden von *checkArgs* ausgewertet und können mittels *printUsage* dem Benutzer angezeigt werden. *createRegistry* gibt an, ob eine neue Registry erzeugt werden soll, falls keine vorgefunden wird. Der Name, unter der die dSelf-VM sich dort anmeldet ist in *VMName* abgelegt. Ob dabei ein eventuell vorhanderner Name überschrieben wird, hängt davon ab, ob *rebindName* auf *true* gesetzt ist. Ob dem Benutzer die gesetzten Optionen angezeigt werden, hängt davon ab, ob *verboseMode* gesetzt ist. Das Objekt in *server* stellt die RMI-Schnittstellen zur Außenwelt zur Verfügung, damit andere dSelf-VM auf diese zugreifen können. Damit diese dSelf-VM selber auf andere entfernte Objekte zugreifen kann, gibt es *remoteVMs*. In diesem Objekt befindet sich eine Liste aller entfernten dSelf-VM, die dieser Virtual Machine bekannt sind. Welches Front-End der Benutzer zu sehen bekommt hängt von *frontEnd* ab, das entweder ein Objekt vom Typ *TerminalFrontEnd* oder *JPanelFrontEnd* (die GUI) ist. Wenn die dSelf-VM gestartet wird, wird zunächst *initializeSelfWorld* aufgerufen, um die initiale dSelf-Welt zu erstellen, die nur die grundlegendsten Objekte von dSelf beinhaltet, wie zum Beispiel *lobbySO*. Danach steht die dSelf-VM dem Benutzer zur Verfügung und wartet auf seine Eingaben. Diese werden von *evalInput* entgegengenommen und an den Compiler weitergereicht. Dabei kann es zu Fehlern kommen, die durch *printCompiletimeError* angezeigt werden. Sollte der Fehler erst bei der Ausführung des erhaltenen Expression-Objektes auftreten, so wird dies mit *printRuntimeError* angezeigt. Wenn kein Problem auftritt, wird das Resultat der Ausführung mit *printMessage* ausgerduckt. Wenn zum laufenden System weitere dSelf-Skripte hinzugeladen werden sollen, wird dies mit *loadScript* erreicht, oder mit *loadScriptDirPath*, wenn der Verzeichnispfad in *dirPath* zum übergebenen Dateinamen hinzugefügt werden soll. Diesen Pfad kann man mit *getDirPath* auslesen oder *setDirPath* neu setzen. Beim laden wird getestet, ob es zum gewünschten Skript eine passende Class-Datei gibt, die den vorcompilierten Code beinhaltet, der mit *loadFromClass* dynamisch hinzu gelinkt werden kann. Falls es dabei Probleme gibt, wenn zum Beispiel die gewünschte Klasse nicht im aktuellen Klassenpfad enthalten ist und somit nicht gelinkt werden kann, wird mit *printWarning* der Benutzer gewarnt, daß die Datei nicht erreichbar ist und statt dessen das Skript eingeladen wird. Die Methode *exit* beendet die dSelf-VM wieder und entfernt dabei ihren Namen aus der RMI-Registry.

5.2.2.1 Das Front-End



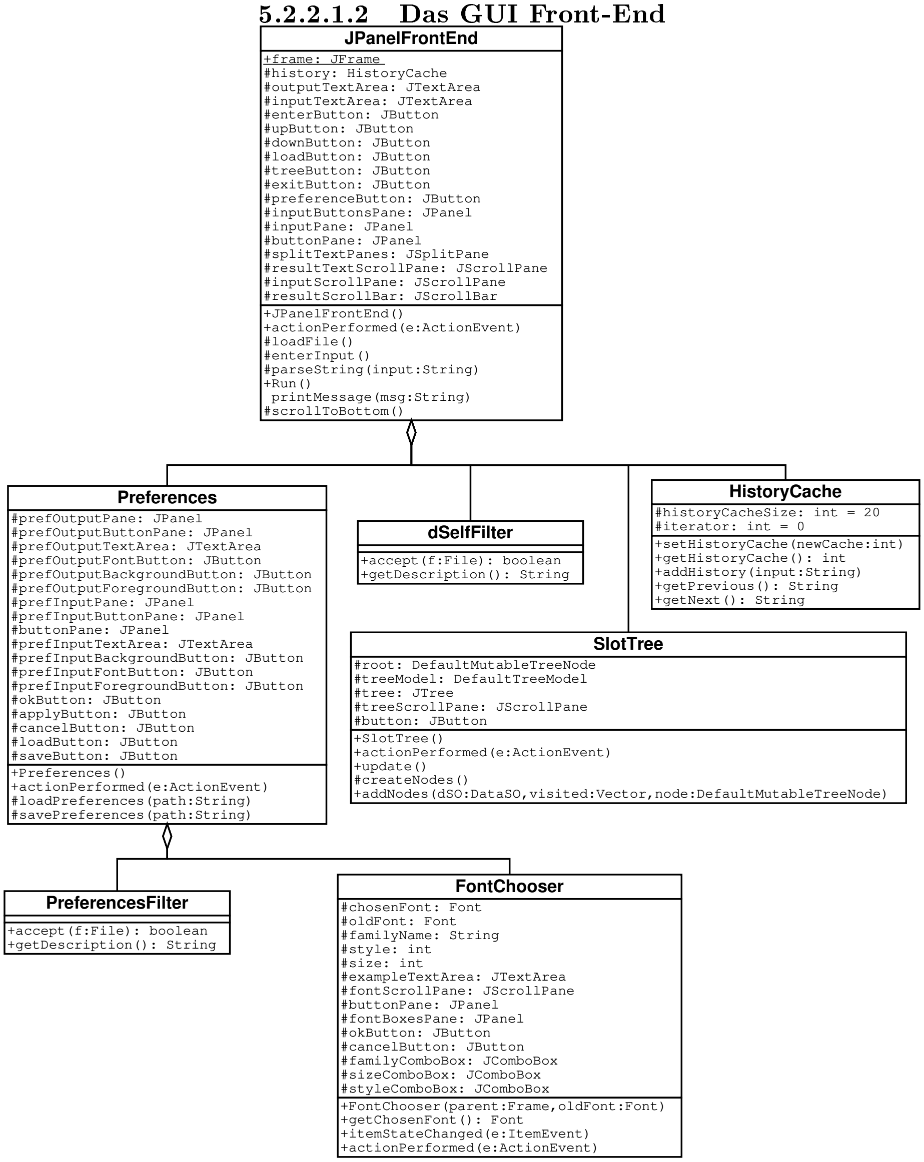
VMFrontEnd ist ein Interface und spezifiziert die gemeinsamen Schnittstellen von *TerminalFrontEnd* und *JPanelFrontEnd*. Mit *Run* läßt sich ein Front-End starten und *printMessage* veranläßt es eine gegebene Nachricht auf seiner Anzeige auszugeben.

5.2.2.1.1 Das Terminal Front-End



Das *TerminalFrontEnd* stellt eine rein Textbasierte Schnittstelle zum Benutzer dar. Es präsentiert sich als eine Shell, die mit *shellPrompt* einen Prompt darstellt, der mit *readInput* die Eingaben von der Tastatur entgegennimmt. Dabei wird mittels *hasOpenParenthesis* herausgefunden, wann das Ende einer Eingabe erreicht worden ist, falls sie sich über mehrere Zeilen hinweg ausdehnt.

5.2.2.1.2 Das GUI Front-End



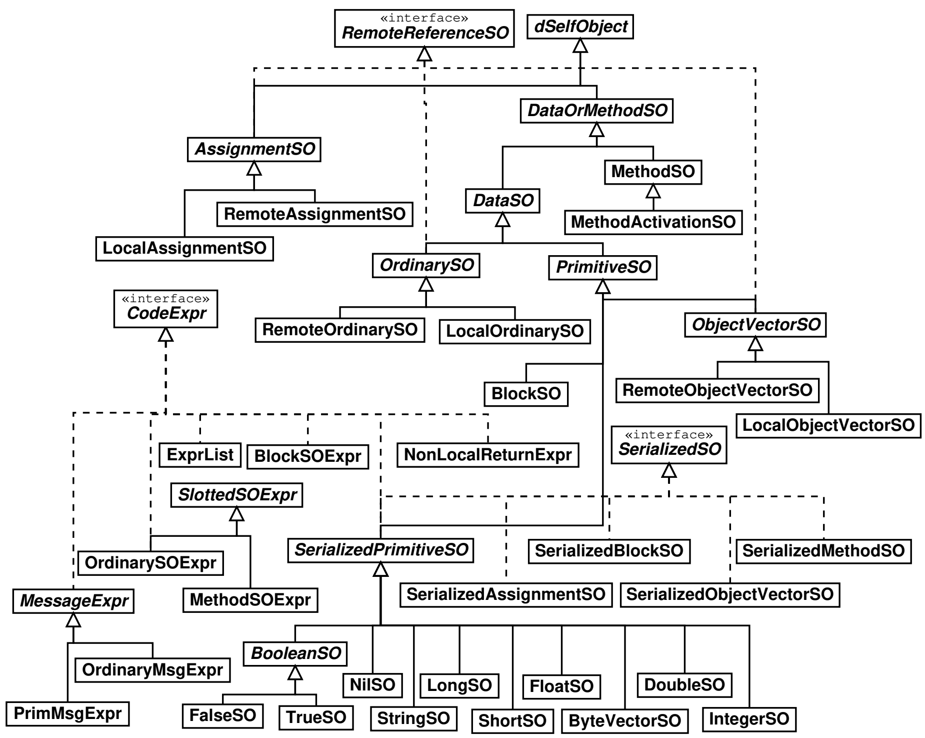
JPanelFrontEnd stellt die GUI-Benutzerschnittstelle zum Benutzer dar. Sie wurde in Java unter zuhilfenahme der Swing-Bibliothek entwickelt. *JPanelFrontEnd* stellt das Hauptfenster dar, während *Preference* das Konfigurationsfenster ist mit einem Dialog *FontChooser* zur Auswahl eines Zeichensatzes. *SlotTree* ist der Slotbrowser und *HistoryCache* stellt die Historyfunktion zur Verfügung,mit der man auf ältere Eingaben zurückgreifen kann. Die einzelnen Filterklassen sind Hilfsklassen für Dateiauswahldialoge, die spezifizieren, welche Dateien geladen werden können. Da diese Klassen hauptsächlich aus grafischen Elementen und den zugehörigen Methoden (für Actionlistener, etc.) bestehen, werden sie an dieser Stelle nicht näher beschrieben. Für detailiertere Beschreibungen für alle Elemente und Methoden dieser Klassen sei hiermit auf die HTML-Klassenbeschreibungen im doc-Verzeichnis der Implementierung verwiesen, die mit JavaDoc erstellt wurde.

5.3 Klassendiagramme

Nachdem im vorhergehenden Abschnitt Objekte wie die Expressions nur oberflächlich beschrieben wurden, soll nun ein deailierterer Blick auf sie geworfen werden. Dazu werden die Klassendiagramme von dSelf dargestellt und das grundlegende Design erörtert. Es werden dabei nur die wichtigsten Klassen dargestellt und auf Hilfsklassen, wie sie zum Beispiel im Compiler verwendet werden, nicht näher eingegangen. Detailierte Informationen hierzu ist den Klassenbschreibungen zu entnehmen, die im HTML-Format mit JavaDoc erstellt wurden und sich im doc-Verzeichnis der Implementierung befinden. Dort werden alle Klassen des Systems mitsamt ihrer Methoden beschrieben.

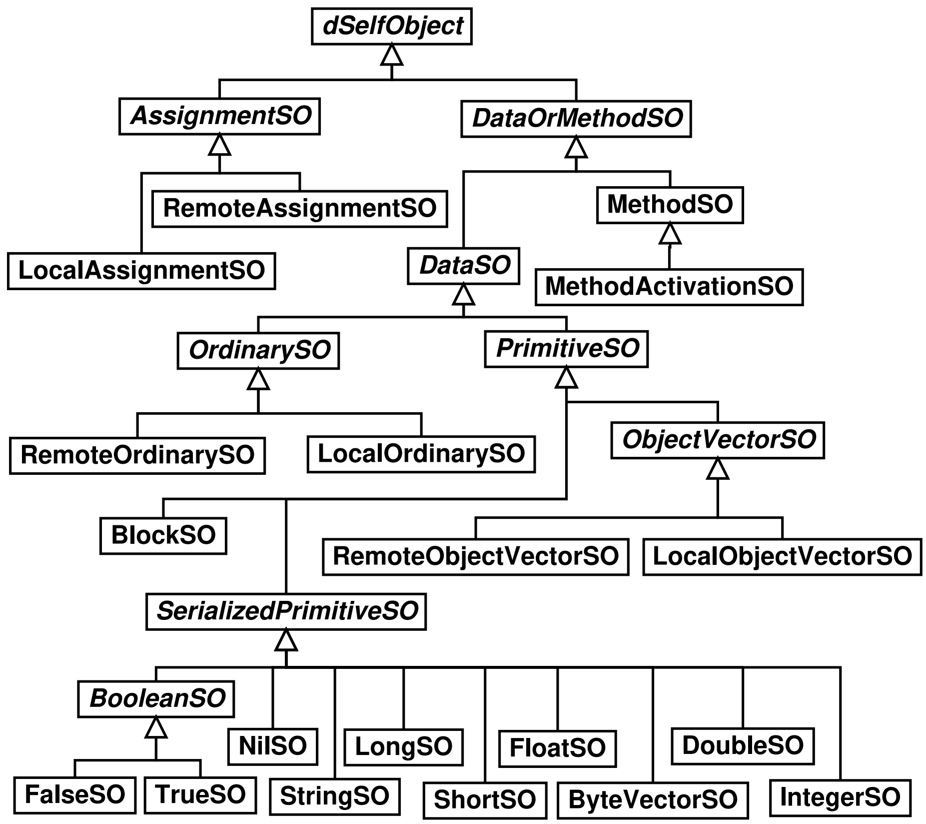
Aus Platzgründen werden in den Diagrammen nur die Klassenhierachien ohne ihre Schnittstellen angegeben. Bei der Einzelbesprechung der wichtigsten Klassen wird jeweils das dazugehörige Diagramm mit allen Schnittstellen in UML-Notation dargestellt.

5.3.1 Expressions und evaluierte Objekte



In diesem Diagramm werden die Klassenhierachien aller Expression-Objekte (abgeleitet von CodeExpr), evaluierten Objekten (abgeleitet von dSelfObject), serialisierbaren Objekten (ageleitet von SerializedSO) und referenzierbaren Objekte (abgeleitet von RemoteReference) dargestellt. Um einen besseren Überblick zu gewinnen, werden diese vier Objektarten samt ihrer Hierachien in den folgenden Abschnitten separat betrachtet.

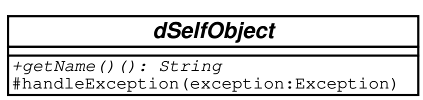
5.3.2 Evaluierte dSelf-Objekte



In diesem Diagramm wird die Klassenhierachie der evaluierten dSelf-Objekte dargestellt. Mit "evaluiert" ist gemeint, daß es sich hierbei im Gegensatz zu den Expression-Objekten um konkrete dSelf-Objekte handelt und nicht um Deklarationen solcher. Die Wurzel dieses Baumes stellt die Klasse *dSelfObject* dar. Alle Klassen, die von ihr abgeleitet werden, repräsentieren dSelf-Objekte, die Inhalt eines Slots sein können. Diese Slotinhalte lassen sich in Zuweisungsobjekte (*AssignmentSO*) und Daten- oder Methodenobjekte (*DataOrMethodSO*) unterteilen. Die Zuweisungobjekte sind, im Unterschied zu Daten- oder Methodenobjekte, keine für den Benutzer direkt zugreifbare Objekte sondern eher Pseudoobjekte, deren Aufgabe es ist andere dSelf-Objekte durch neu zu ersetzen. Man unterscheidet zwischen lokalen (*LocaAssignmentSO*) und entfernten (*RemoteAssignmentSO*) Zuweisungsobjekten, die im Abschnitt "Entfernte Referenzobjekte" näher beschrieben werden. *MethodSO* repräsentiert die Methodenobjekte von dSelf und MethodActivationSO seine Aktivierung. Wenn eine Methode in dSelf aufgerufen wird, wird diese (*MethodSO*) zunächst geklont und zum Empfänger der Nachricht gebracht, die diesen Methodenaufruf veranlaßte. Dort wird sie um einen Self-Slot erweitert (*MethodActivationSO*) und ausgeführt. *DataSO* stellt die Basis für alle Datenobjekte dar. Man unterscheidet zwischen gewöhnlichen Objekten (*OrdinarySO*), die vom Benutzer definiert werden können, und primitiven Objekten (*PrimitiveSO*), die vom System zur Verfügung gestellt werden. Von den gewöhnlichen Objekten gibt es zwei Versionen, nämlich die lokal vorhandenen (*LocalOrdinarySO*) und solche, die sich auf einer entfernten dSelf-VM (*RemoteOrdinarySO*) befinden. Das primitive Objekt *ObjectVectorSO* stellt die Vektoren in dSelf dar und unterteilt sich wie *AssignmentSO* und *OrdinarySO* in eine lokale (*LocalObjectVectorSO*) und eine entfernte (*RemoteObjectVectorSO*) Version. *BlockSO* stellt einen Block dar. Es ist ebenfalls ein Datenobjekt, da es zwar ein Methodenobjekt beinhaltet, aber sich nach außen hin wie ein Datenobjekt verhält. Es kann wie Datenobjekte Nachrichten empfangen und wird erst evaluiert, wenn ihm die Nachricht value geschickt wird, während Methoden bei ihrem Auftreten immer evaluiert werden und ihr Resultat die Nachricht erhält. Eine spezielle Art von primitiven Objekten stellen die *SerializedPrimitiveSO* dar, deren Unterklassen alle serialisierbar sind. Die von *BooleanSO* abgeleiteten boolschen Objekte *TrueSO* und *FalseSO* sowie *NilSO* sind in dSelf im Gegensatz zu SELF primitive Objekte. Weitere primitive Objekte sind die ganzen Zahlen *ShortSO*, *IntegerSO* und *LongSO* sowie die Fließkommazahlen *FloatSO* und *DoubleSO*. *StringSO* repäsentiert die Strings in dSelf und *ByteVectorSO* ist ein Vektor von Bytewerten.

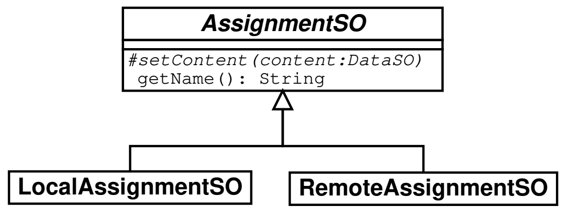
Aus platzgründen wurden hier keine Schnittstellen angegeben, daher werden die einzelnen Klassen im folgenden noch einmal einzeln präsentiert mit allen Variablen und Methoden. Es wird allerdings nur auf die wichtigsten Elemente eingegangen. Für weitere Informationen sei hier auf die Klassendokumentation in der Implementierung von dSelf verwiesen, die sich dort im doc-Verzeichnis befindet.

5.3.2.1 dSelfObject



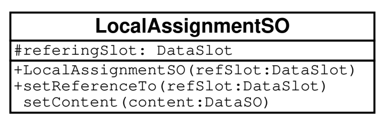
dSelfObject ist die Basis aller evaluierten Objekte in dSelf. Sie bietet eine Methode *getName* an, die den Namen des (z.B.: “a block”, “an object”, etc.). *handleException* ist eine Hilfsmethode zur Behandlung von *Exceptions*. Sie gibt eine zur *Exception* passende Fehlermeldung aus.

5.3.2.2 AssignmentSO



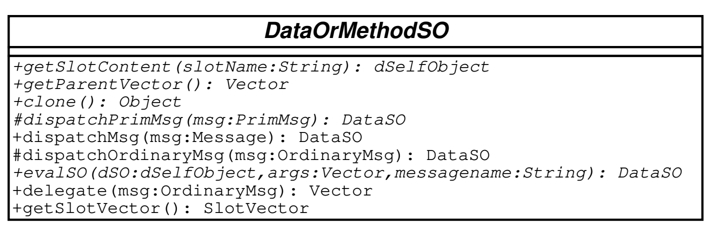
AssignmentSO stellt den Inhalt eines Zuweisungsslots dar. Sie hat zwei Unterklassen *LocalAssignmentSO* und *RemoteAssignmentSO*, die jeweils ein lokales oder verteiltes Zuweisungsobjekt repräsentieren. Beide bieten eine Methode *setContent* an, dessen Argument (ein Datenobjekt) in den Slot gesetzt wird, auf den dieses Objekt verweist. *RemoteAssignmentSO* wird zu einem späteren Zeitpunkt bei der Besprechung von verteilten Referenzobjekten vorgestellt.

5.3.2.3 LocalAssignmentSO



LocalAssignmentSO stellt einen Zuweisungsslot dar, der lokal auf einer dSelf Virtual Machine vorhanden ist. Die Klasse besitzt eine Variable *referingSlot*, die auf den Datenslot verweist, dessen Inhalt überschrieben werden soll. Die Referenz kann dem Objekt entweder bei der Initialisierung über den Konstruktor oder nachträglich per *setReferenceTo* übergeben werden.

5.3.2.4 DataOrMethodSO



DataOrMethodSO stellt die Basis für Daten- und Methodenobjekte dar. Daten- und Methodenobjekte haben Slots, deren Inhalt man mit *getSlotContent* abfragen kann. Mit *getParentVector* erhält man die Inhalte aller Parentslots dieses Objektes und getSlotVector gibt alle Slots dieses Objekts zurück. Die wichtigste Methode stellt *dispatchMsg* dar, da sie alle Nachrichten, die an dieses Objekt gesendet werden, engegennimmt und als Ergebnis ein Datenobjekt zurückgibt. Intern wird in dieser Methode ermittelt, ob es sich bei dieser Nachricht um eine primitive (beginnt mit einem Unterstrich) oder eine gewöhnliche Nachricht handelt. Wenn sie primitiv ist, wird sie an *dispatchPrimMsg* weitergereicht und dort ausgewertet. Falls nicht wird sie zu *dispatchOrdinaryMsg* weitergeleitet, wo durch *delegate* ein lokaler Slot gesucht wird, dessen Name mit dem Selektor der Nachricht übereinstimmt. Wenn dies nicht der Fall ist, wird die Nachricht mit *delegate* an die Parentobjekte weitergereicht und die gefundenen Objekte durch *evalSO* im Kontext dieses Objektes evaluiert. Mit *clone* kann man ein Objekt klonen (shallow copy) und erhält ein Objekt mit neuen, gleichnamigen Slots.

5.3.2.5 MethodSO

MethodSO

```
#slots: SlotVector
#expressions: ExprList
---
+MethodSO(slotVec:SlotVector,expr:ExprList)
 getSlotContent(slotName:String): dSelfObject
 getParentVector(): Vector
 getSlotVector(): SlotVector
+getExpressionList(): ExprList
 clone(): Object
+getActivation(args:Vector,name:String): MethodActivationSO
 getName(): String
 evalSO(dSO:dSelfObject,args:Vector,messagename:String): DataSO
 dispatchPrimMsg(msg:PrimMsg): DataSO
```

MethodSO repräsentiert die Methodenobjekte von dSelf. Es besteht aus lokalen Slots, die in slots abgelegt werden und seinem Code, der in *expressions* enthalten ist. Diese Bestandteile wurden ihm bei der Instanzierung über den Konstruktor übergeben. Mittels *getExpressionList* kann man auf den Code dieses Methodenobjekts zugreifen und *getActivation* erzeugt ein *method activation object* für diese Methode, das die gleichen Slots und denselben Code besitzt.

5.3.2.6 MethodActivationSO

MethodActivationSO

```
#self: SlotContentSO
---
+MethodActivationSO(slots:SlotVector,expr:ExprList,args:Vector,name:String)
+execute(context:DataOrMethodSO): DataSO
 dispatchOrdinaryMsg(msg:OrdinaryMsg): DataSO
 dispatchPrimMsg(msg:PrimMsg): DataSO
 evalSO(dSO:dSelfObject,args:Vector,messagename:String): DataSO
```

MethodActivationSO repräsentiert ein *method activation object*, das einem geklonten Methodenobjekt entspricht, dem ein Self-Slot (*self*) hinzugefügt wird, der auf das aufrufende Objekt verweist. Im Kontext dieses Objekts wird diese Methode dann mit *execute* mit dem aufrufenden Objekt als Argument ausgeführt.

5.3.2.7 DataSO

```
DataSO
------------------------------------------------------------
 getSlotVector(): SlotVector
#getAnnotation(): String
 evalSO(dSO:dSelfObject,args:Vector,messagename:String): DataSO
#checkIfIntegerSO(msg:String,dSO:dSelfObject)
#checkIfShortSO(msg:String,dSO:dSelfObject)
#checkIfLongSO(msg:String,dSO:dSelfObject)
#checkIfFloatSO(msg:String,dSO:dSelfObject)
#checkIfDoubleSO(msg:String,dSO:dSelfObject)
#checkIfStringSO(msg:String,dSO:dSelfObject)
#checkIfBlockSO(msg:String,dSO:dSelfObject)
#checkIfByteVectorSO(msg:String,dSO:dSelfObject)
#checkIfObjectVectorSO(msg:String,dSO:dSelfObject)
#checkIfObjectVectorSO(msg:String,dSO:dSelfObject)
#execFailBlock(msg:PrimMsg,e:dSelfException): DataSO
 dispatchPrimMsg(msg:PrimMsg): DataSO
#_Clone(): DataSO
#_Credits(): DataSO
#_CurrentTimeString(): StringSO
#_Describe(): DataSO
#_DirPath1(): StringSO
#_DirPath2(so:dSelfObject): StringSO
#_Eq(so:dSelfObject): BooleanSO
#_GarbageCollect(): DataSO
#_ObjectID(): IntegerSO
#_Memory(): LongSO
#_OperatingSystem(): StringSO
#_Perform(msg:dSelfObject): DataSO
#_PerformWith(args:Vector): DataSO
#_PerformResend(msg:dSelfObject): DataSO
#_PerformResendWith(args:Vector): DataSO
#_PerformDelegatingTo(msg1:dSelfObject,msg2:dSelfObject): DataSO
#_PerformDelegatingToWith(args:Vector): DataSO
#_Print(): NilSO
#_Quit()
```

DataSO stellt die Basis für die Datenobjekte dar. Solche Objekte besitzen sogenannte Anmerkungen, die man mit *getAnnotation* abfragen kann. Die Methoden, die mit einem Unterstrich beginnen stellen die Implementierungen der gleichnamigen primitiven Nachrichten dar. Innerhalb dieser Methoden werden ihre Argumente auf ihren Typ hin geprüft, indem eine der Methoden mit einem check-Präfix aufgerufen wird. Sollte dabei ein Typfehler entdeckt werden, wird ein eventuell vorhandener Fail-Block der Nachricht mit execFailBlock ausgeführt. Auf die Methoden, die mit einem Unterstrich beginnen, wird im folgenden nicht näher eingegangen, da eine Beschreibung dieser im Anhang zu finden ist.

5.3.2.8 OrdinarySO

```
OrdinarySO
----------------------------------------------------------
#annotation: String
----------------------------------------------------------
 getAnnotation(): String
#removeAllSlots()
#removeSlot(remSlot:String)
#addSlots(slots:SlotVector)
#addSlotsIfAbsent(slots:SlotVector)
 clone(): Object
#isParentSlot(slotName:String,msgName:String): boolean
-isCorrectStartChar(ch:char): boolean
-isCorrectBodyChar(ch:char): boolean
 dispatchPrimMsg(msg:PrimMsg): DataSO
#_AddSlotConnectedTo(newSlotName:DataSO,host:DataSO): DataSO
#_AddSlots(arg:DataSO): DataSO
#_AddSlotsIfAbsent(arg:DataSO): DataSO
#_RemoveAllSlots(): DataSO
#_RemoveSlot(remSlot:dSelfObject): DataSO
```

Gewöhnliche Datenobjekte werden durch *OrdinarySO* repräsentiert. Gewöhnliche Datenobjekte können manipuliert werden, indem mit *addSlots* oder *addSlotsIfAbsent* neue Slots hinzugefügt werden, wobei die erste Version bereits vorhandene, gleichnamige Slots überschreibt. Entfernen kann man diese einzeln mit *removeSlot* oder komplett mit *removeAllSlots*. Mit is *ParentSlot* läßt sich überprüfen, ob ein Slot mit dem spezifizierten Namen ein Parentslot ist.

5.3.2.9 LocalOrdinarySO

```
LocalOrdinarySO
----------------------------------------------------------
#slotVector: SlotVector
----------------------------------------------------------
+LocalOrdinarySO()
+LocalOrdinarySO(slots:SlotVector,anno:String)
+LocalOrdinarySO(slots:Slot)
+LocalOrdinarySO(anno:String)
 getName(): String
 addSlot(:Slot)
 removeAllSlots()
 removeSlot(remSlot:String)
 getSlotVector(): SlotVector
 getParentVector(): Vector
 getSlotContent(slotName:String): dSelfObject
 addSlots(slots:SlotVector)
 addSlotsIfAbsent(slots:SlotVector)
```

LocalOrdinarySO stellt ein gewöhnliches Datenobjekt dar, das auf der lokalen dSelf Virtual Machine lokalisiert ist. Es besitzt eine Reihe von Slots, die in slotVector abgelegt sind. Die zur Verfügung gestellten Methoden stellen Implementierungen bereits besprochener Methoden dar.

5.3.2.10 PrimitiveSO

```
PrimitiveSO
----------------------------------------------------------
----------------------------------------------------------
#getParent(): LocalOrdinarySO
 getAnnotation(): String
 clone(): Object
 getSlotContent(slotName:String): dSelfObject
```

PrimitiveSO stellt die Basisklasse für alle primitiven Objekte in dSelf dar. Primitive Objekte sind im Gegensatz zu gewöhnlichen Datenobjekten nicht modifizierbar, indem man Slots hinzufügt oder löscht. Sie besitzen nur einen Parentslot, auf den man mit *getParent* zugreifen kann.

5.3.2.11 ObjectVectorSO

```
ObjectVectorSO
-----------------------------------------------------------
#parent: LocalOrdinarySO
#slotVector: SlotVector
#parVector: Vector
-----------------------------------------------------------
 clone(): Object
+getObjectAt(pos:int): DataSO
+putObjectAt(pos:int,arg:DataSO)
+getLength(): int
 getSlotVector(): SlotVector
 getParent(): LocalOrdinarySO
 getParentVector(): Vector
 getName(): String
 dispatchPrimMsg(msg:PrimMsg): DataSO
#_At(arg:dSelfObject): DataSO
#_AtPut(at:dSelfObject,arg:dSelfObject): ObjectVectorSO
#_CloneFiller(size:dSelfObject,filler:dSelfObject):  ObjectVectorSO
#_CopyRangeDstPosSrcSrcPosLength(dP:dSelfObj,sV:dSelfObj,sP:dSelfObj,l:dSelfObj): ObjectVectorSO
#_Size(): IntegerSO
```

ObjectVectorSO stellt einen Vektor von dSelf-Objekten dar. Man kann auf seine Elemente mittels *getObjectAt* zugreifen oder durch *putObjectAt* neue Elemente hinzufügen lassen. Die Länge wird duch *getLength* zurückgegeben.

5.3.2.12 LocalObjectVectorSO

```
LocalObjectVectorSO
-----------------------------------------------------------
#objVector: DataSO[]
-----------------------------------------------------------
+LocalObjectVectorSO()
+LocalObjectVectorSO(oVec:DataSO[])
 getObjectAt(pos:int): DataSO
 putObjectAt(pos:int,arg:DataSO)
 getLength(): int
```

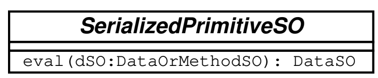
LocalObjectVectorSO implementiert *ObjectVectorSO* als einen lokalen Objektvektor, der seine Elemente in einem Array aus Datenobjekten namens *objVector* ablegt.

5.3.2.13 BlockSO

```
BlockSO
-----------------------------------------------------------
#parent: LocalOrdinarySO
#slotVector: SlotVector
#parVector: Vector
#blockMethodObject: MethodSO
#lexicalLink: DataOrMethodSO
-----------------------------------------------------------
+BlockSO(m:MethodSO)
+BlockSO(context:DataOrMethodSO,methodObject:MethodSO)
 getSlotVector(): SlotVector
 getParent(): LocalOrdinarySO
 getParentVector(): Vector
 getName(): String
+getMethod(): MethodSO
 dispatchOrdinaryMsg(msg:OrdinaryMsg): DataSO
 eval(context:DataOrMethodSO): DataSO
```

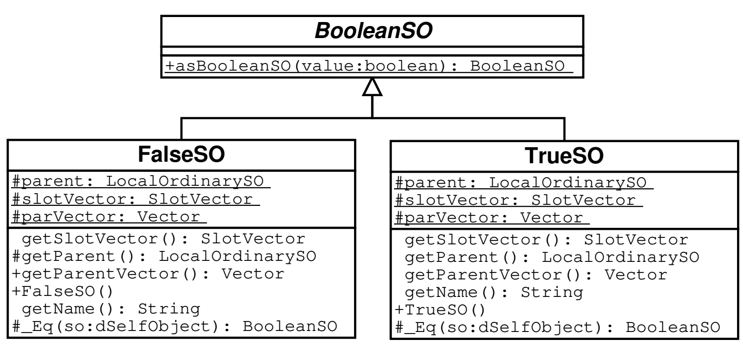
Die Klasse *BlockSO* stellt die Blockobjekte von dSelf dar. Blöcke sind selber Datenobjekte, da sie sich wie solche verhalten. Sie besitzen ein Parentobjekt, in das der Benutzer seine Slots hinzufügen kann und empfängt Nachrichten wie zum Beispiel *value*. Sein Inhalt besteht aus einem Methodenobjekt, das in *blockMethodObject* abgelegt wird und durch *getMethod* zugreifbar ist. Damit ein Block mit seinem umgebenden Kontext (lexical scope) kommunizieren kann, wird dieser in *lexicalLink* abgelegt.

5.3.2.14 SerializedPrimitiveSO



SerializedPrimitiveSO stellt eine spezielle Art von primitiven Datenobjekten dar. Klassen, die von ihr abgeleitet werden, sind alle ohne zusätzlichen Aufwand serialisierbar, da lediglich ihr Inhalt kopiert werden muß. Die Methode *eval* erbt sie von *CodeExpr* und wird bei der Beschreibung von Expression-Objekten näher betrachtet.

5.3.2.15 BooleanSO



BooleanSO stellt die boolschen Werte *true* (*TrueSO*) und *false* (*FalseSO*) dar. Sie stellt eine Klassenmethode namens *asBooleanSO* zur Verfügung, mit der sich Java-Booleans in dSelf-Booleans umwandeln lassen. *TrueSO* und *FalseSO* sind, im Gegensatz zu SELF, primitive Objekte, da sie nur so effizient von einer dSelf Virtual Machine zu einer Anderen transportiert werden können.

5.3.2.16 NilSO

```
NilSO
-----------------------------------------
#parent: LocalOrdinarySO
#slotVector: SlotVector
#parVector: Vector
-----------------------------------------
+NilSO()
 getParent(): LocalOrdinarySO
 getParentVector(): Vector
 getName(): String
#_Eq(so:dSelfObject): BooleanSO
```

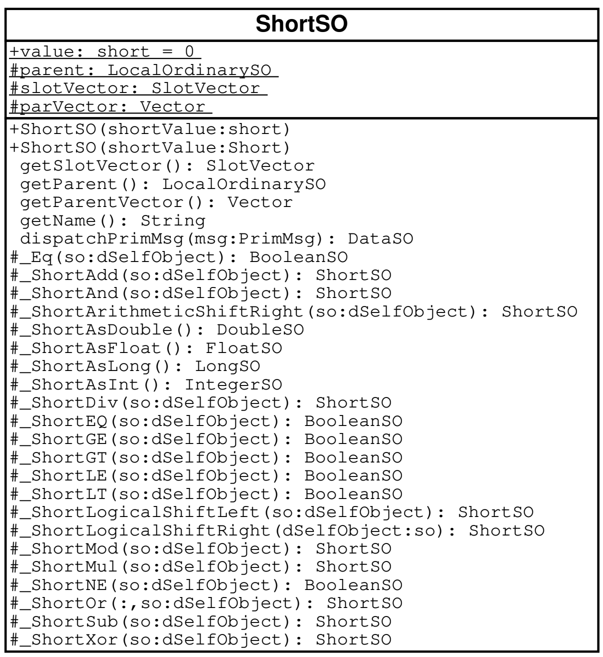
NilSO entspricht dem Objekt *nil* von SELF. Es ist aus dem selben Grund wie *TrueSO* und *FalseSO* als ein primitives Datenobjekt realisiert worden.

5.3.2.17 StringSO

```
StringSO
-----------------------------------------
#string: String
#parent: LocalOrdinarySO
#slotVector: SlotVector
#parVector: Vector
-----------------------------------------
+StringSO(str:String)
 getSlotVector(): SlotVector
 getParent(): LocalOrdinarySO
 getParentVector(): Vector
 clone(): Object
+getString(): String
 getName(): String
 dispatchPrimMsg(msg:PrimMsg): DataSO
#_BitSize(): IntegerSO
#_Eq(arg:dSelfObject): BooleanSO
#_RunScript(): DataSO
#_StringAsByteVector(): ByteVectorSO
#_StringAt(arg:dSelfObject): IntegerSO
#_StringConcatenate(arg:dSelfObject): StringSO
#_StringGetSubFrom(from:dSelfObject): StringSO
#_StringGetSubFromTo(from:dSelfObject,to:dSelfObject): StringSO
#_StringPrint(): StringSO
#_StringSize(): IntegerSO
```

StringSO stellt einen String in dSelf dar. Sein Wert wird als Java-String in *string* gespeichert und ist über *getString* abfragbar.

5.3.2.18 ShortSO



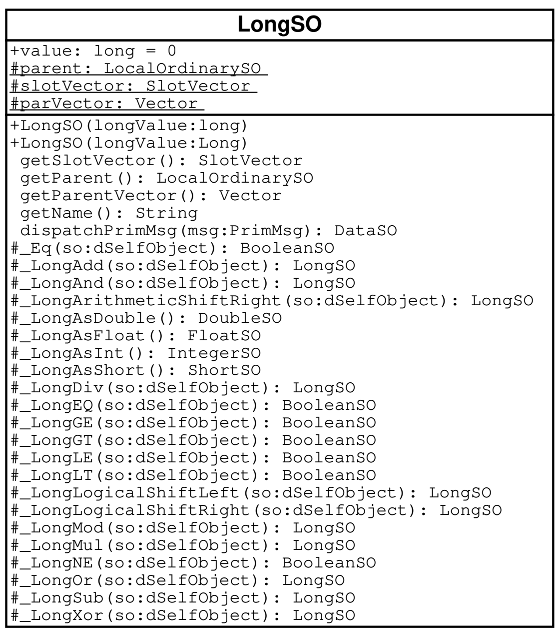
ShortSO stellt eine ganze Zahl dar, die in ihrere Bitbreite einem Java-Short entspricht. Ihr aktueller Wert wird in *value* abgelegt.

5.3.2.19 IntegerSO

```
IntegerSO
+value: int = 0
#parent: LocalOrdinarySO
#slotVector: SlotVector
#parVector: Vector
+IntegerSO(intValue:int)
+IntegerSO(intValue:Integer)
 getSlotVector(): SlotVector
 getParent(): LocalOrdinarySO
 getParentVector(): Vector
 getName(): String
 dispatchPrimMsg(msg:PrimMsg): DataSO
#_Eq(so:dSelfObject): BooleanSO
#_IntAdd(so:dSelfObject): IntegerSO
#_IntAnd(so:dSelfObject): IntegerSO
#_IntArithmeticShiftRight(so:dSelfObject): IntegerSO
#_IntAsDouble(): DoubleSO
#_IntAsFloat(): FloatSO
#_IntAsLong(): LongSO
#_IntAsShort(): ShortSO
#_IntDiv(so:dSelfObject): IntegerSO
#_IntEQ(so:dSelfObject): BooleanSO
#_IntGE(so:dSelfObject): BooleanSO
#_IntGT(so:dSelfObject): BooleanSO
#_IntLE(so:dSelfObject): BooleanSO
#_IntLT(so:dSelfObject): BooleanSO
#_IntLogicalShiftLeft(so:dSelfObject): IntegerSO
#_IntLogicalShiftRight(so:dSelfObject): IntegerSO
#_IntMod(so:dSelfObject): IntegerSO
#_IntMul(so:dSelfObject): IntegerSO
#_IntNE(so:dSelfObject): BooleanSO
#_IntOr(so:dSelfObject): IntegerSO
#_IntSub(so:dSelfObject): IntegerSO
#_IntXor(so:dSelfObject): IntegerSO
```

IntegerSO stellt eine ganze Zahl dar, die in ihrere Bitbreite einem Java-Integer entspricht. Ihr aktueller Wert wird in *value* abgelegt.

5.3.2.20 LongSO



LongSO stellt eine ganze Zahl dar, die in ihrere Bitbreite einem Java-Long entspricht. Ihr aktueller Wert wird in *value* abgelegt.

5.3.2.21 FloatSO

```
FloatSO
-------------------------------------------------
+float: value = 0
#parent: LocalOrdinarySO
#slotVector: SlotVector
#parVector: Vector
-------------------------------------------------
+FloatSO(floatValue:float)
+FloatSO(floatValue:Float)
 getSlotVector(): SlotVector
 getParent(): LocalOrdinarySO
 getParentVector(): Vector
 getName(): String
 dispatchPrimMsg(msg:PrimMsg): DataSO
#_Eq(arg:dSelfObject): BooleanSO
#_FloatAdd(so:dSelfObject): FloatSO
#_FloatAsDouble(): DoubleSO
#_FloatAsInt(): IntegerSO
#_FloatAsLong(): LongSO
#_FloatAsShort(): ShortSO
#_FloatCeil(): FloatSO
#_FloatDiv(so:dSelfObject): FloatSO
#_FloatEQ(so:dSelfObject): BooleanSO
#_FloatFloor(): FloatSO
#_FloatGE(so:dSelfObject): BooleanSO
#_FloatGT(so:dSelfObject): BooleanSO
#_FloatLE(so:dSelfObject): BooleanSO
#_FloatLT(so:dSelfObject): BooleanSO
#_FloatMod(so:dSelfObject): FloatSO
#_FloatMul(so:dSelfObject): FloatSO
#_FloatNE(so:dSelfObject): BooleanSO
#_FloatPrintString(): StringSO
#_FloatPrintStringPrecision(so:dSelfObject): StringSO
#_FloatRound(): IntegerSO
#_FloatSub(so:dSelfObject): FloatSO
#_FloatTruncate(): FloatSO
```

FloatSO stellt eine Fließkommazahl dar, die in ihrere Bitbreite einem Java-Float entspricht. Ihr aktueller Wert wird in *value* abgelegt.

5.3.2.22 DoubleSO

```
DoubleSO
-------------------------------------------------
+value: double = 0
#parent: LocalOrdinarySO
#slotVector: SlotVector
#parVector: Vector
-------------------------------------------------
+DoubleSO(doubleValue:double)
+DoubleSO(doubleValue:Double)
 getSlotVector(): SlotVector
 getParent(): LocalOrdinarySO
 getParentVector(): Vector
 getName(): String
 dispatchPrimMsg(msg:PrimMsg): DataSO
#_Eq(arg:dSelfObject): BooleanSO
#_DoubleAdd(so:dSelfObject): DoubleSO
#_DoubleAsFloat(): FloatSO
#_DoubleAsInt(): IntegerSO
#_DoubleAsLong(): LongSO
#_DoubleAsShort(): ShortSO
#_DoubleCeil(): DoubleSO
#_DoubleDiv(so:dSelfObject): DoubleSO
#_DoubleEQ(so:dSelfObject): BooleanSO
+_DoubleFloor(): DoubleSO
#_DoubleGE(so:dSelfObject): BooleanSO
#_DoubleGT(so:dSelfObject): BooleanSO
#_DoubleLE(so:dSelfObject): BooleanSO
#_DoubleLT(so:dSelfObject): BooleanSO
#_DoubleMod(so:dSelfObject): DoubleSO
#_DoubleMul(so:dSelfObject): DoubleSO
#_DoubleNE(so:dSelfObject): BooleanSO
#_DoublePrintString(): StringSO
#_DoublePrintStringPrecision(so:dSelfObject): StringSO
#_DoubleRound(): LongSO
#_DoubleSub(so:dSelfObject): DoubleSO
#_DoubleTruncate(): DoubleSO
```

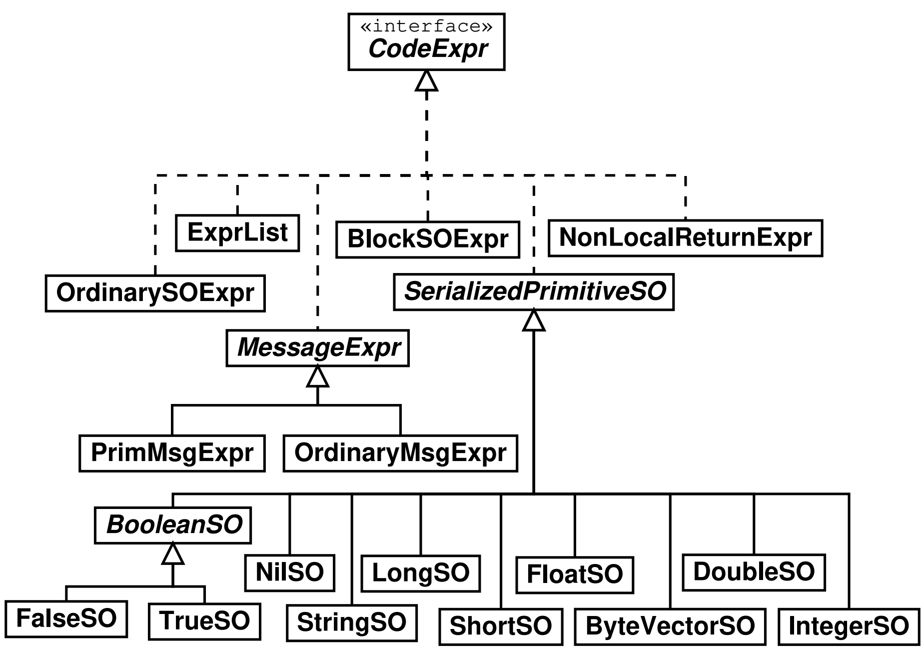
DoubleSO stellt eine Fließkommazahl dar, die in ihrere Bitbreite einem Java-Double entspricht. Ihr aktueller Wert wird in *value* abgelegt.

5.3.2.23 ByteVectorSO

```
ByteVectorSO
--------------------------------------------------------------
+byteVector: byte[]
#parent: LocalOrdinarySO
#slotVector: SlotVector
#parVector: Vector
--------------------------------------------------------------
+ByteVectorSO()
+ByteVectorSO(bytes:byte[])
 getSlotVector(): SlotVector
+getParent(): LocalOrdinarySO
 getParentVector(): Vector
 getName(): String
 clone(): Object
 dispatchPrimMsg(msg:PrimMsg): DataSO
#_ByteAt(arg:dSelfObject): IntegerSO
#_ByteAtPut(at:dSelfObject,arg:dSelfObject): ByteVectorSO
#_ByteSize(): IntegerSO
#_ByteVectorAsString(): StringSO
#_ByteVectorCompare(arg:dSelfObject): IntegerSO
#_ByteVectorConcatenate(arg:dSelfObject): ByteVectorSO
#_CloneBytesFiller(size:dSelfObject,filler:dSelfObject): ByteVectorSO
#_CopyByteRangeDstPosSrcSrcPosLength(dP:dSelfObj,sV:dSelfObj,sP:dSelfObj,l:dSelfObj): ByteVectorSO
#_StringPrint(): ByteVectorSO
```

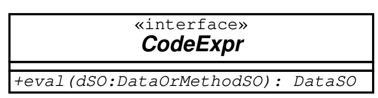
ByteVectorSO stellten einen Vektor von Bytewerten dar, die im Array *byteVector* abgelegt werden.

5.3.3 dSelf-Expressionobjekte



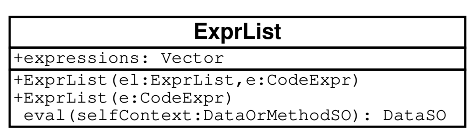
Expression-Objekte stellen die Deklarationen von Objekten dar. Diese Objekte werden benötigt, um den Code von Methodenobjekten realisieren zu können. Da dSelf keinen Interpreter besitzt, der einfach einen String ausliest und dann direkt die dazugehörigen semantischen Aktionen ausführt, braucht man diese Objekte. Sie sind in gewisser Weise noch nicht evaulierte Objekte, die bei der Aufführung des Codes ihr jeweiliges Pendant erstellen. So wird zum Beispiel aus einem *BlockSOExpr* nach dessen Evaluierung ein konkretes *BlockSO*-Objekt. Die Basis der Expression-Objekte bildet CodeExpr. Alle Klassen, die von ihr abgeleitet werden, können Bestandteil des Codes eines Methodenobjekts sein. Dies können gewöhnliche Datenobjekte (OrdinarySOExpr), Blöcke (BlockSOExpr) oder Nachrichten (MessageExpr) sein. *ExprList* stellt eine Liste von Expression-Objekten dar, die selbst wiederum eine Expression darstellt. Eine besondere Aufgabe wird durch *NonLocalReturnExpr* realisiert. Sie stellt die *non-local returns* im Code von Blockobjekten dar, die die Abarbeitung des Codes sofort beenden. Die serialisierbaren, primitiven Datenobjekte (*SerializedPrimitiveSO*) wurden schon in den vorhergehenden Abschnitten beschrieben. Aufgrund ihres einfachen Aufbaus ist es nicht notwendig spezielle Expression-Objekte fürsie zu erzeugen.

5.3.3.1 CodeExpr



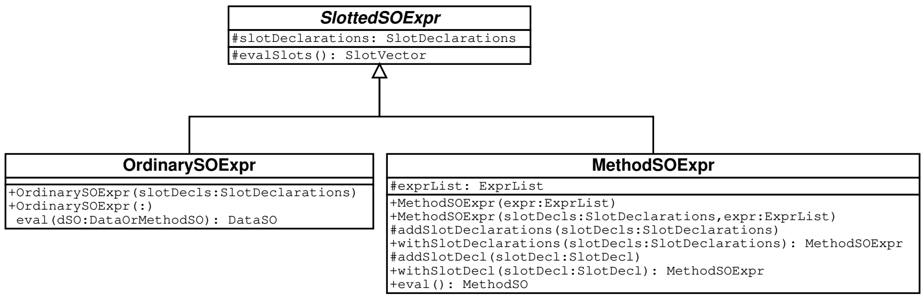
CodeExpr ist die Basis aller Expression-Objekte in dSelf. Sie vererbt die Methodes *eval*, die in einem durch das Argumnet übergebenen Kontext evaluiert wird. Das Resultat einer solchen Evaluierung ist immer ein Datenobjekt.

5.3.3.2 ExprList



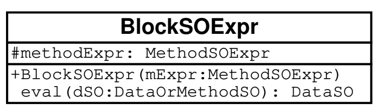
ExprList steltt eine Liste von Expressions dar, die im Vektor *expressions* gesammelt werden. Der Code eines Methodenobjekts besteht aus einer solchen Liste.

5.3.3.3 OrdinarySOExpr



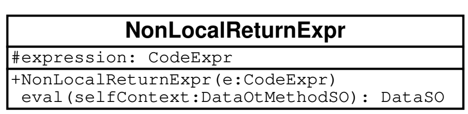
OrdinarySOExpr stellt die Expression eines gewöhnlichen Datenobjekts dar. Es erbt von *SlottedSOExpr* seine Slotdeklarationen vom Typ *SlotDeclarations* (werden später noch vorgestellt), die in *slotDeclarations* abgelegt werden. Mit *evalSlots* kann diese Slots evaluieren. Ebenfalls von *SlottedSOExpr* erbt *MethodSOExpr*. Diese Klasse definiert Methodenobjekte, ohne jedoch wie *OrdinarySOExpr* von *CodeExpr* zu erben. Der Grund liegt darin, daß Methodendeklarationen innerhalb des Codes einer Methode (Inner Method) in dSelf und SELF nicht erlaubt ist. Man kann eine neue Slotdeklaration mit *addSlotDecl* zum Methodenobjekt hinzufügen, oder mehrere mit *addSlotDeclarations*. *withSlotDecl* und *withSlotDeclarations* machen dasselbe wie *addSlotDecl* und *addSlotDeclarations*, geben jedoch zusätzlich einen *this*-Zeiger zurück.

5.3.3.4 BlockSOExpr



BlockSOExpr stellt Block-Expressions dar. Ein solcher Block besteht aus einer Methodenobjektdeklaration *methodExpr*, die zu einem Methodenobjekt evaluiert wird.

5.3.3.5 NonLocalReturnExpr

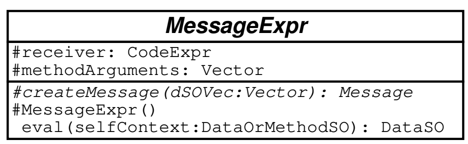


NonLocalReturnExpr stellt einen Ausdruck dar, der einen *non-local return* realisiert. Sein Inhalt besteht aus einer beliebigen *CodeExpr*, die in *expression*. Ein *non-local return* wird in Blöcken benötigt und hat ein anderes Verhalten als eine normale Returnanweisung in einem Block, da durch sie die Abarbeitung der Ausdrücke in dem sie umgebenden Kontext sofort beendet wird. Hierzu ein Beispiel:

```
method = (
    <cond>
        ifTrue:  [...  <expr>] “local return”
        ifFalse: [... ^<expr>| “non-local return”
    ...
)
```

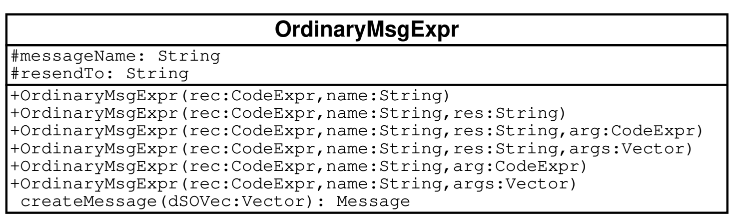
Hier werden zwei Blöcke angegeben, von denen der erste einen *local return* mit dem Wert des letzten Ausdrucks erzeugt. Nach der Auswertung dieses Blocks wird der darauf folgende Code der Methode weiter angearbeitet. Der zweite Block realisiert einen *non-local return*, der durch das “^”-Zeichen markiert wird. Bei dessen Auswertung wird der Wert des evaluierten Ausdrucks als Rückgabewert von der ihn umgebenden Methode betrachtet und dessen Ausführung sofort beendet.

5.3.3.6 MessageExpr



MessageExpr ist die Basis für Ausdrücke von gewöhnlichen oder primitiven Nachrichtenobjekten. Sie besitzt eine *CodeExpr* in receiver, die den Empfänger dieser Nachricht mit seinen Argumenten in *methodArguments* repräsentiert. *createMessage* erzeugt das konkrete Nachrichtenobjekt vom Typ *Message*, das an den Empfänger dieser Nachricht gesendet wird.

5.3.3.7 OrdinaryMsgExpr



OrdinaryMsgExpr stellt eine gewöhnliche Nachricht in dSelf dar. Der Name dieser Nachricht wird in *messageName* abgelegt und wenn diese Nachricht per *resend* verschickt wurde, wird dessen Name in *resendTo* spezifiziert.

5.3.3.8 PrimMsgExpr

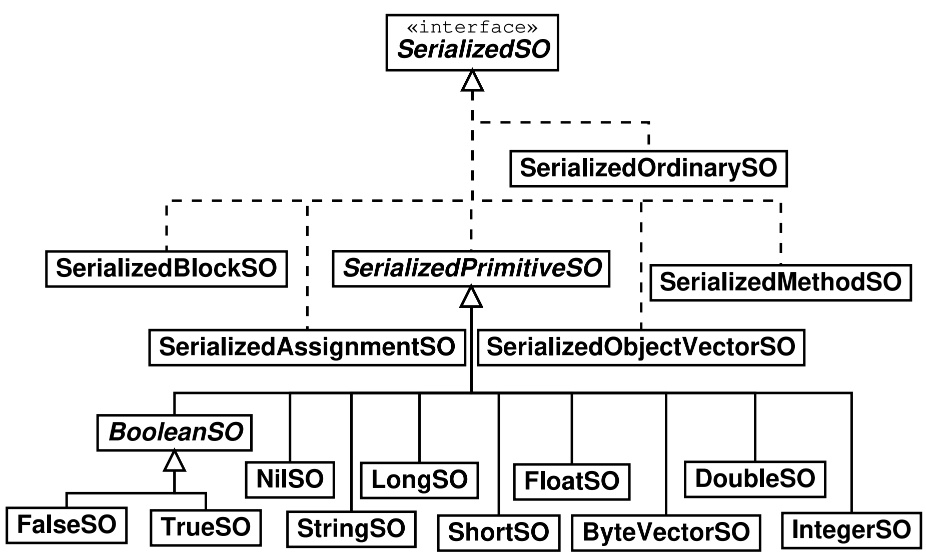
PrimMsgExpr
#primID: int #hasFailBlock: boolean = false
+PrimMsgExpr(rec:CodeExpr,ID:int) +PrimMsgExpr(CodeExpr:rec,ID:int,fail:boolean) +PrimMsgExpr(rec:CodeExpr,ID:int,arg:CodeExpr) +PrimMsgExpr(rec:CodeExpr,ID:int,arg:CodeExpr,fail:boolean) +PrimMsgExpr(rec:CodeExpr,ID:int,args:Vector) +PrimMsgExpr(rec:CodeExpr,ID:int,args:Vector,fail:boolean) createMessage(dSOVec:Vector): Message

PrimMsgExpr stellt eine primitive Nachricht dar. Im Gegensatz zu gewöhnlichen Nachrichten haben sie eine ID statt einem Namen in Stringform. Die IDs zu allen primitiven sind in *PrimMsg* als statische Konstanten angegeben. Falls diese Nachricht über einen angehängten *IfFail:*-Block verfügt ist hasFailBlock gleich *true*.

5.3.3.9 SerializedPrimitiveSO

Die *SerializedPrimitiveSO*-Klasse wurde mit den von ihr abgeleiteten Klassen bereits vorgestellt. Diese Klassen stellen dSelf-Objekte dar, die alle eine einfache Struktur besitzten. In der Regel handelt es sich dabei nur um eine Instanzvariable. Diese Objekte stellen bereits bei ihrer Deklaration ein evaluiertes Objekt dar, so daß beim Aufruf von *eval* einfach das Objekt selber (*this*) zurückgegeben werden muß.

5.3.4 Serialisierbare Objekte



Die serialisierbaren Objekte von dSelf erben alle von *SerializedSO*. Sie spielen in einer verteilten Umgebung eine wichtige Rolle, da sie angeben, wie ein Objekt über die Grenzen der lokalen Java Virtual Machine hinweg transportiert werden sollen. Die hier abgebildeten Klassen *SerializedBlockSO* (Blöcke), *SerializedAssignmentSO* (Zuweisungsobjekte), *SerializedObjectVectorSO, SerializedOrdinarySO* und *SerializedMethodSO* (Methodenobjekte) stellen die serialisierten Versionen der gleichnamigen (ohne Serialized-Präfix) dSelf-Objekte dar. Wenn man sich mit der Javaprogrammierung etwas auskennt, sieht es wahrscheinlich ein wenig seltsam aus, daß es zu bereits bestehenden Klassen noch einmal neue, serialisierte Versionen davon gibt, da dies eine Klasse normalerweise selber erledigt. In der Regel geht dies auch auf diese Weise, doch die aufgezählten Objekte stellen einen Ausnahmefall dar, da sie alle eine Sonderbehandlung benötigen, die in den folgenden Abschnitten näher betrachtet wird. Eine Ausnahme stellen Klassen vom Typ *SerializedPrimitiveSO* dar, da sie nur kopiert werden und somit durch das Erben von Javas *Serializable* automatisch serialisiert werden.

5.3.4.1 SerializedSO

Klassen vom Typ *SerializedSO* erben von Javas *Serializable* und sind somit automatisch serialisierbar, da lediglich ihre Instanzvariablen kopiert werden müssen.

5.3.4.2 SerializedOrdinarySO

SerializedOrdinarySO
`#ID: int`
`+SerializedOrdinarySO(ID:int)` `+getID(): int`

SerializedOrdinarySO wird an eine entfernte dSelf Virtual Machine verschickt, wenn diese ein gewöhnliches Datenobjekt anfordert. In so einem Fall wird nicht das eigentliche Objekt verschickt, sondern lokal eine Referenz darauf erzeugt und in einer Liste abgelegt (siehe *Server* an späterer Stelle). Der Index dieser Referenz in der Liste entspricht der *ID*, unter der man dieses Objekt ansprechen kann und wird zum Aufrufer verschickt.

5.3.4.3 SerializedMethodSO

SerializedMethodSO
`#slots: SerializedSlotVector` `#expressions: ExprList`
`+SerializedMethodSO(method:MethodSO,server:Server)` `+getMethod(remoteVM:ServerInterface): MethodSO`

SerializedMethodSO stellt die serialisierte Version von *MethodSO* dar. Der Code eines Methodenobjekts in *expressions* ist zwar serialisierbar, aber seine Slots brauchen eine Sonderbehandlung, da sich in ihm zum Beispiel gewöhnliche Datenobjekte befinden können, die nicht über das Netz kopierbar sind.

Daher werden seine Slots serialisiert und in *slots* abgelegt. Der Empfänger dieses Objekts kann mit *getMethod* die Methode wieder herstellen lassen und sie in seinem Kontext ausführen.

5.3.4.4 SerializedBlockSO

SerializedBlockSO
`#serializedMethod: SerializedMethodSO`
`+SerializedBlockSO(block:BlockSO,server:Server)` `+getBlock(remoteVM:ServerInterface): BlockSO`

SerializedBlockSO stellt die serialisierte Version von *BlockSO* dar. Da sein Inhalt eine Methode ist, muß diese, wie im vorhergehenden Punkt beschrieben, erst serialisiert und in *serializedMethod* abgelegt werden. Mit *getBlock* wird der Block wieder hergestellt.

5.3.4.5 SerializedAssignmentSO

SerializedAssignmentSO
`#ID: int`
`+SerializedAssignmentSO(ID:int)` `+getID(): int`

Zuweisungsobjekte können nicht über die Grenzen einer dSelf Virtual Machine hinaus kopiert werden, da die Slots, auf die sie referenziern, Bestandteile von gewöhnlichen Datenobjekten sind. Daher wird wie bei gewöhnlichen Datenobjekten eine Referenz erstellt und dessen *ID* durch *SerializedAssignmentSO* dem Empfänger zugesandt.

5.3.4.6 SerializedObjectVectorSO

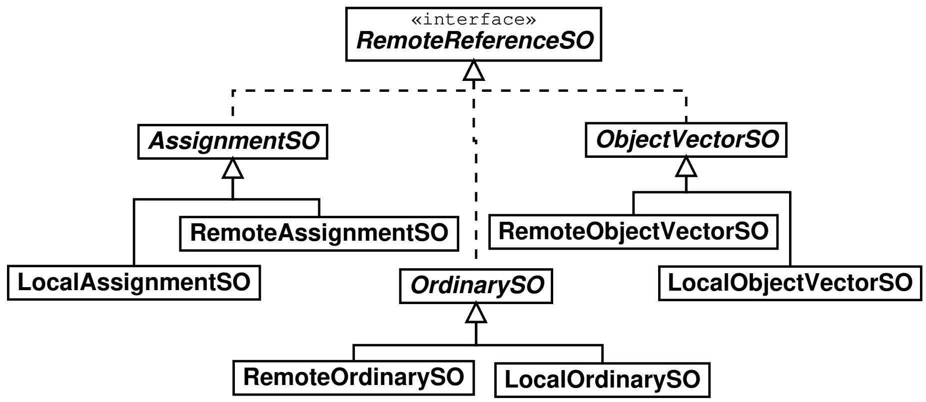
SerializedObjectVectorSO
`#ID: int`
`+SerializedObjectVectorSO(ID:int)` `+ID(): int`

Da die Elemente eines Objektvektors gewöhnliche Datenobjekte sein können, kann ein *ObjectVectorSO* nicht zu einer entfernten dSelf-VM kopiert werden. Daher wird auch hier wie bei gewöhnlichen Datenobjekten eine Referenz erstellt und dessen *ID* durch *SerializedObjectVectorSO* dem Empfänger zugesandt.

5.3.4.7 SerializedPrimitiveSO

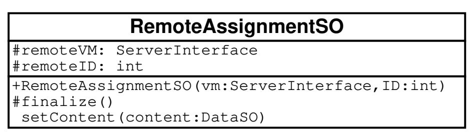
Da primitive Datenobjekte kopierbar sind, brauchen sie keine Sonderbehandlung wie die zuvor beschriebenen Objekte.

5.3.5 Verteilte Referenzobjekte



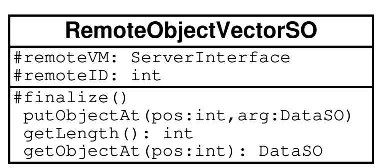
Im vorhergehenden Abschnitt wurde bereits erwähnt, wie die IDs von Refernzen verschickt wurden. Diese IDs verweisen auf lokale Objekte, auf die von einer entfernten dSelf-VM zugegriffen werden soll. Damit von dieser entfernten dSelf-VM auf diese Objekte zugegriffen werden kann, werden dort Referenzobjekte vom Typ *RemoteReferenceSO* erstellt. Diese Objekte stellen eine Art Pseudoobjekt dar, das sich darauf beschränkt die Schnittstellen des refernzierten Objekts anzubieten, um diese Aufrufe an das "echte" Objekt auf der entfernten dSelf-VM weiterzuleiten.

5.3.5.1 RemoteAssignmentSO



RemoteAssignmentSO stellt die Schnittstellen von *AssignmentSO* zur Verfügung und leitet die jeweiligen Aufrufe an *LocalAssignmentSO* auf der entfernten dSelf-VM weiter.

5.3.5.2 RemoteObjectVectorSO



RemoteObjectVectorSO stellt die Schnittstellen von *ObjectVectorSO* zur Verfügung und leitet die jeweiligen Aufrufe an *LocalObjectVectorSO* auf der entfernten dSelf-VM weiter.

5.3.5.3 RemoteOrdinarySO

```
RemoteOrdinarySO
--------------------------------------------------
#remoteVM: ServerInterface
#remoteID: int
#name: String
--------------------------------------------------
+RemoteOrdinarySO(vm:ServerInterface,ID:int)
#finalize()
 getParentVector(): Vector
 getName(): String
 getSlotContent(slotName:String): dSelfObject
 getSlotVector(): SlotVector
 addSlots(slots:SlotVector)
 addSlotsIfAbsent(slots:SlotVector)
 removeAllSlots()
 removeSlot(slotName:String)
```

RemoteOrdinarySO stellt die Schnittstellen von *OrdinarySO* zur Verfügung und leitet die jeweiligen Aufrufe an *LocalAssignmentSO* auf der entfernten dSelf-VM weiter.

5.3.5.4 Server

```
Server
--------------------------------------------------
#references: Vector
#vectorLeaksStack: Stack
--------------------------------------------------
+Server()
+connectToLobby(): int
#addRemoteReference(refObj:RemoteReferenceSO): int
+removeRemoteReference(ID:int)
#getObjAt(ID:int): dSelfObject
+getNameOf(ID:int): String
+getSlotContentOf(ID:int,slotName:String): SerializedSO
+getParentsOf(ID:int): SerializedSO[]
+getSlotsOf(ID:int): SerializedSlotVector
+removeAllSlotsOf(ID:int)
+removeSlotOf(ID:int,slotName:String)
+addSlotsTo(ID:int,VMName:String,remSlots:SerializedSlotVector)
+addSlotsIfAbsentTo(ID:int,VMName:String)
+setContentOf(ID:int,VMName:String,obj:SerializedSO)
+getLengthOfObjectVector(ID:int): int
+getObjectOfObjectVector(ID:int,pos:int): SerializedSO
+putObjectOfObjectVector(ID:int,VMName:String,pos:int,so:SerializedSO)
```

Server stellt eine Klasse dar, die im Kontext der verteilten Referenzen benötigt wird. Sie stellt die Schnittstelle einer dSelf Virtual Machine zur Außenwelt dar. Sie besteht aus einem Vektor namens *references*, der alle lokalen Objekte referenziert, auf die eine externe Referenz verweist. Um Lücken in diesem Vektor zu vermeiden, die entstehen wenn verteilte Referenzen wieder entfernt werden, wird zur Hilfe ein Stack namens *vectorLeaksStack* benutzt. Dieser Stack beinhaltet alle zur Verfügung stehenden Indices des Vektors. Wenn nun eine Referenz hinzugefügt wird, wird vom Stack ein freier Index geholt und an der entsprechenden Stelle im Vektor die Referenz abgelegt und wenn eine Referenz gelöscht wird, wird der freigewordene Index wieder auf den Stack geschoben und kann beim nächsten Aufruf wieder benutzt werden.

Um mit der dSelf-VM, die diesen Server anbietet, kommunizieren zu können, muß man zunächst eine Verbindung zu ihr aufbauen, indem man *connectToLobby* aufruft. Dadurch wird auf die Lobby eine Referenz gesetzt, indem *addRemoteReference* mit der Lobby als Argument aufgerufen wird. Der Rückgabewert entspricht dem Index, an dem sie in *references* abgelegt wurde. Um eine Referenz wieder zu löschen ruft man einfach *removeRemoteReference* mit dem dazugehörigen Index auf. Über diesen Index (ID) kann man dieses Objekt nun manipulieren. Man kann seinen Namen abfragen (*getNameOf*), den Inhalt eines seiner Slots abfragen (*getSlotContentOf*), auf seine Parentobjekte zugreifen (*getParentsOf*), seine Slots anforden (*getSlots*), alle Slot entfernen (*removeAllSlotsOf*), einzelne Slots entfernen (*removeSlotOf*), neue Slots hinzufügen (*addSlotsTo*), neue Slots hinzufügen ohne alte zu entfernen (*addSlotsIfAbsentTo*) und neue Slotinhalte setzen (*setContentOf*). Speziell für Objektvektoren gibt es Methoden, um ihre Länge abzufragen (*getLengthOfObjectVector*), neue Elemente hinzuzufügen (*getObjectOfObjectVector*) und auf Elemente zugreifen (*putObjectOfObjectVector*).

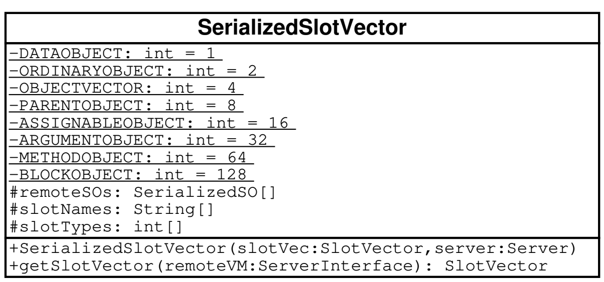
5.3.6 Wichtige Hilfsklassen

5.3.6.1 SlotVector

SlotVector
+NO_SLOT_FOUND: int = -1
+SlotVector() +SlotVector(slots:SlotVector) +SlotVector(slot:Slot) +clone(): Object +getSlotIndex(slotName:String): int +getSlot(slotName:String): Slot +getSlotAt(index:int): Slot +getSlotContent(slotName:String): dSelfObject +getSlotContent(slotNr:int): dSelfObject +getParentVector(): Vector +addSlot(newSlot:Slot) +addSlotIfAbsent(newSlot:Slot) #addAssignableDataSlot(slot:DataSlot) +addAssignableDataSlotIfAbsent(slot:DataSlot) +addSlotVector(newSlots:SlotVector) +addSlotVectorIfAbsent(newSlots:SlotVector) #removeSlot(slotIndex:int): boolean +removeSlot(slotName:String): boolean +printSlots(): String +describeSlots(): String +isAssignable(pos:int): boolean +setAnnotations(a:String): SlotVector +initArgs(args:Vector,methodName:String)

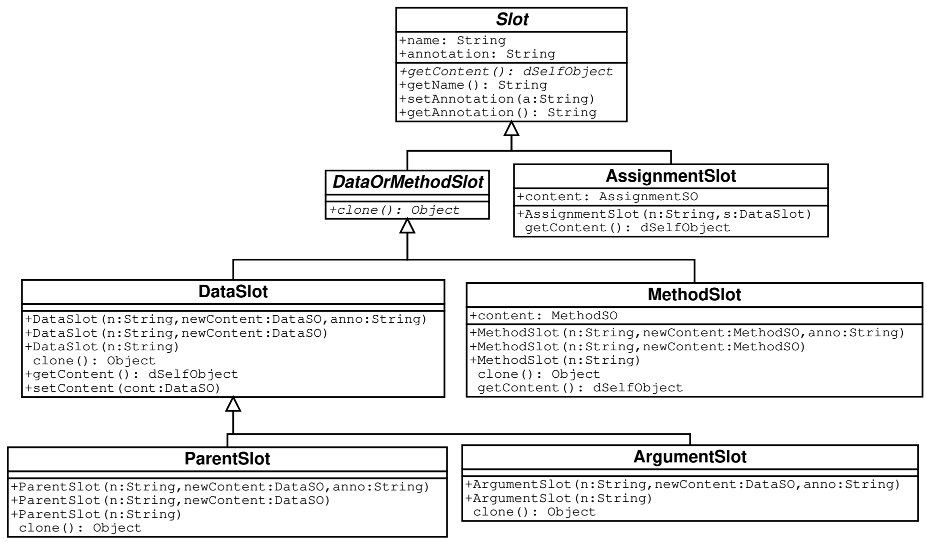
In den vorhergehenden Abschnitten wurde bereits erwähnt, das Methoden- und gewöhnliche Datenobjekte Slots beinhalten, die in einer Klasse namens *SlotVector* gespeichert werden. Diese Klasse erbt von Javas *Vector* und erweitert diese um neue Schnittstellen zur Verwaltung der Slots. Wenn man auf einen Slot mit gegebenen Namen zugreifen möchte, wird dies von *getSlot* erledigt. Alternativ kann man auch seinen Index mit *getSlotIndex* ermitteln und dann mit *getSlotAt* auf ihn zugreifen. *getSlotContent* gibt statt dem Slot seinen Inhalt zurück. Wenn man an den Inhalten der aller Parentslots interessiert ist, kann man diese mit *getParentObjets* erhalten. Neue Slots kann man einzeln mit *addSlot* oder *addSlotIfAbsent* hinzufügen, wobei die erste Version gleichnamige Slots überschreibt. Wenn hingegen mehrere Slots auf einmal hinzugefügt werden sollen, wird dies von *addSlotVector* und *addSlotVectorIfAbsent* erledigt. Wenn man einen Datenslot hinzufügen möchte, der zuweisbar sein soll, muß man *addAssignableDataSlot* oder *addAssignableDataSlotIfAbsent* aufrufen, da dadurch der dazugehörige Zuweisungsslot automatisch mit generiert wird. Mit *removeSlot* kann man einen Slots mit dem spezifizierten Namen oder dem angegebenen Index wieder löschen. Wenn die Struktur der Slots beschrieben werden soll, erhält man eine solche Beschreibung in Stringform durch den Aufruf von *printSlots*. Mit *isAssignable* läßt sich ermitteln, ob der Slot an der spezifizierten Position zuweisbar ist oder nicht. Slots können in dSelf mit sogenannten Annotations (Anmerkungen) beschriftet werden, indem man *setAnnotations* mit einer Beschreibung als Argument aufruft. Diese Beschreibungen lassen sich mit *describeSlots* ausgeben. Wenn dieser *Slotvector* zu einer Methode gehört, dann beinhaltet er eventuell Argumentslots, die mit den Argumenten der Methode gefüllt werden. In so einem Fall werden diese Argumentslots durch *initArgs* mit den übergebenen Argumenten belegt. Wenn eine Kopie von diesen Slots erstellt werden soll, kann man mit *clone* (shallow copy) eine erstellen lassen.

5.3.6.2 SerializedSlotVector



SerializedSlotVector stellt die serialisierte Version von *SlotVector* dar. Bei der erzeugung eines solchen Objekts, muß der *SlotVector* übergeben werden, der serialisiert werden soll. Beim Empfänger angekommen, kann der ihn durch den Aufruf von *getSlotVector* wieder erstellen lassen.

5.3.6.3 Slots



Es gibt verschiedene Arten von Slots, je nach ihrem Verwendungszweck. Die gemeinsame Oberklasse ist Slot und besitzt einen Namen, der in name abgelegt wird und mit *getName* abgefragt werden kann. Desweitern besitzt sie eine Anmerkung, die in *annotation* abgelegt und über *getAnnotation* ausgegeben oder mit *setAnnotation* neu gesetzt wird. Alle Slots besitzen einen Inhalt, den man durch *getContent* erhält.

AssignmentSlot stellt einen Zuweisungsslot dar, dessen Inhalt ein *AssignmentSO* ist. Er wird erstellt, indem man dem Konstruktor seinen Namen und den Datenslot übergibt, dessen Inhalt durch ihn ersetzt werden soll.

Die Oberklasse von Daten- und Methodenslots stellt *DataOrMethodSlot* dar, die eine Methode *clone* zum kopieren dieses Slots anbietet.

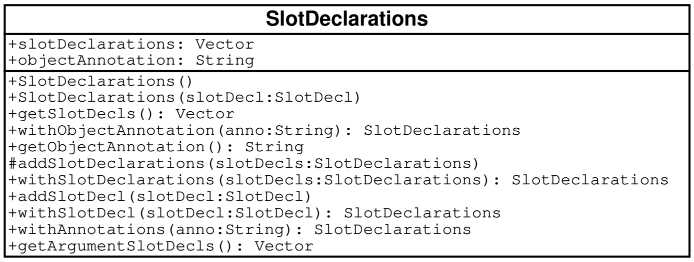
Ein Methodenslot (*MethodSlot*) nimmt ein Methodenobjekt auf, das ihm gemeinsam mit seinem Namen und optional einer Anmerkung bei der Erzeugung übergeben wird.

Datenslots (*DataSlot*) werden auf die gleiche Weise erzeugt, erhalten aber statt einer Methode ein Datenobjekt, das durch *setContent* zu einem späteren Zeitpunkt durch ein Anderes ausgetauscht werden kann.

Eine Spezialisierung von Datenslots stellen Parentslots (*ParentSlot*) dar. Sie besitzen zwar dieselbe Struktur wie normale Datenslots, werden aber durch ihren Typ vom *lookup algorithm* erkannt und zur Vererbung verwendet.

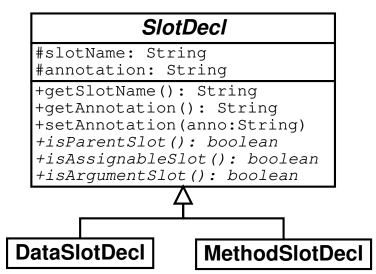
ArgumentSlot stellt einen Slot einer Methode dar, die mit den Argumenten einer Methode beim Aufruf gefüllt wird.

5.3.6.4 SlotDeclarations



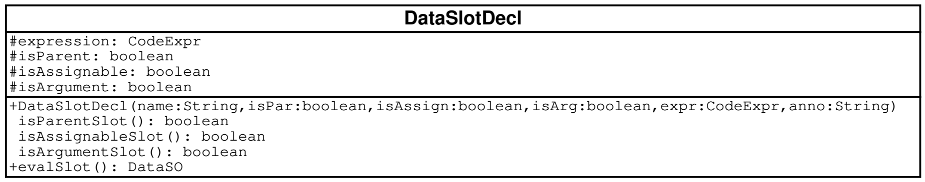
So wie es evaluierte Objekte und die dazugehörigen Expression-Objekte gibt, gibt es zu *SlotVector* die Klasse *SlotDeclarations*. Sie nimmt die Deklarationen von Slots auf, die erst zu einem späteren Zeitpunkt evaluiert werden sollen. *SlotDeclarations* besteht aus einem Vector mit dem Namen *slotDeclarations*, der die einzelnen Deklarationen beinhaltet und einem String *objectAnnotation*, der die Anmerkung für das Objekt beinhaltet, das diese Slots erhalten wird. Mit *getSlotDecls* kann man auf den Inhalt von *slotDeclarations* zugreifen und *getObjectAnnotation* gibt die Anmerkung zurück. *withObjectAnnotaion* fügt eine neue Anmerkung hinzu und gibt eine Selbstreferenz zurück (*this*). Wenn man neue Deklarationen hinzufügen möchte, kann man dies einzeln mit *addSlotDecl* oder im Packet durch *addSlotDeclarations* erreichen, oder mit *withSlotDecl* oder *withSlotDeclarartions*, wenn als Resultat eine Selbstreferenz erhalten werden soll. Wenn man hingegen nur an den Argumentslots interessiert ist, kann man sie durch Aufruf von *getArgumentSlots* bekommen.

5.3.6.5 SlotDecl



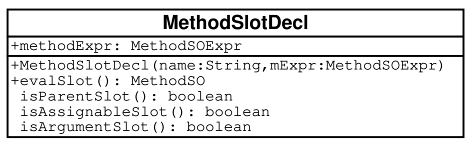
SlotDecl stellt das nicht evaluierte Gegenstück zu *Slot* dar. Es besteht aus einem Namen in *slotName* und einer Anmerkung in *annotation*. Den Inhalt dieser Variablen kann man mit *getSlotName* und *getAnnotation* auslesen oder mit *setAnnotation* neu setzten. *isParentSlot*, *isAssignableSlot* und *isArgumentSlot* geben Auskunft darüber, ob der Slot ein Parentslot, zuweisbar oder ein Argumentslot ist.

5.3.6.6 DataSlotDecl



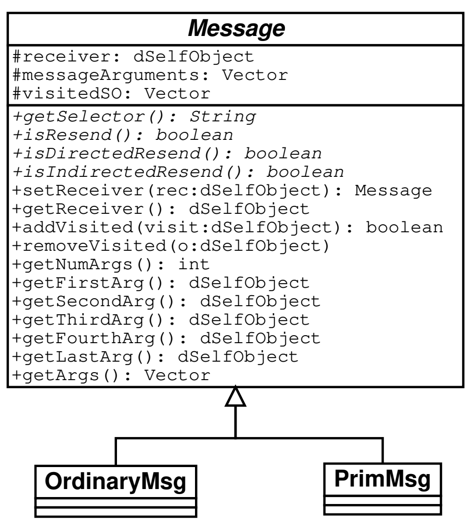
DataSlotDecl stellt eine Deklaration eines Datenslots dar. Sie besitzt ein Expression-Objekt als Inhalt in *expression* und boolsche Variablen *isParent*, *isAssignable* und *isArgument*, die darüber Auskunft geben, ob der Slot ein Parentslot, zuweisbar oder ein Argumentslot ist. Mit *evalSlot* wird der Slot im Kontext der Lobby evaluiert.

5.3.6.7 MethodSlotDecl



MethodSlotDecl stellt die Deklaration eines Methodenslots dar, der mit *evalSlot* evaluiert werden kann. *isParentSlot*, *isAssignableSlot* und *isArgumentSlot* geben immer *false* zurück, da sie diese Eigenschaften nicht besitzen können.

5.3.6.8 Message



In dSelf kommunizieren die Objekte untereinander durch das Versenden von Nachrichten. Diese Nachrichten werden durch das Objekt *Message* repräsentiert. Eine Nachricht besteht aus einem Empfängerobjekt namens receiver, an das sie gesendet wird und einem *Vector* mit Argumenten, der in *messageArguments* abgelegt wird. *visitedSO* ist ein Vector von besuchten dSelf-Objekten, der vom Vererbungsalgorithmus benötigt wird, um Zyklen im Suchpfad zu entdecken. Mit *getSelector* erhält man den Namen dieser Nachricht und *isResend* gibt Auskunft darüber, ob sie durch *resend* verschickt wurde. Ob dieser *resend* gerichtet oder ungerichtet ist, kann man durch *isDirectedResend* und *isIndirectedResend* erfahren. Der Empfänger dieser Nachricht kann mit *getReceiver* ausgelesen und *setReceiver* neu gesetzt werden. *addVisited* fügt ein besuchtes Objekt in *visitedSO* hinzu und *removeVisited* entfernt ein solches. Auf die Argumente kann man entweder mit *getFirstArg*, *getSecondArg*, *getThirdArg*, *getFourthArg* und *getLastArg* das erste, zweite, dritte, vierte und letzte Argument zugreifen oder alle gemeinsam durch *getArgs* erhalten.

5.3.6.9 OrdinaryMsg

```
OrdinaryMsg
----------------------------------------------------
#selector: String
#resendTo: String
----------------------------------------------------
+OrdinaryMsg(sel:String)
+OrdinaryMsg(sel:String,args:Vector)
+OrdinaryMsg(sel:String,res:String)
+OrdinaryMsg(sel:String,args:Vector,res:String)
 isResend(): boolean
 isIndirectedResend(): boolean
 isDirectedResend(): boolean
 getResendReceiver(): String
+getSelector(): String
```

OrdinaryMsg stellt eine gewöhnliche Nachricht dar. Sie besteht aus einem Namen, der in *selector* abgelegt ist und kann, wenn sie per *resend* versendet wurde, einen weiteren Empfänger in *resendTo* haben.

5.3.6.10 PrimMsg

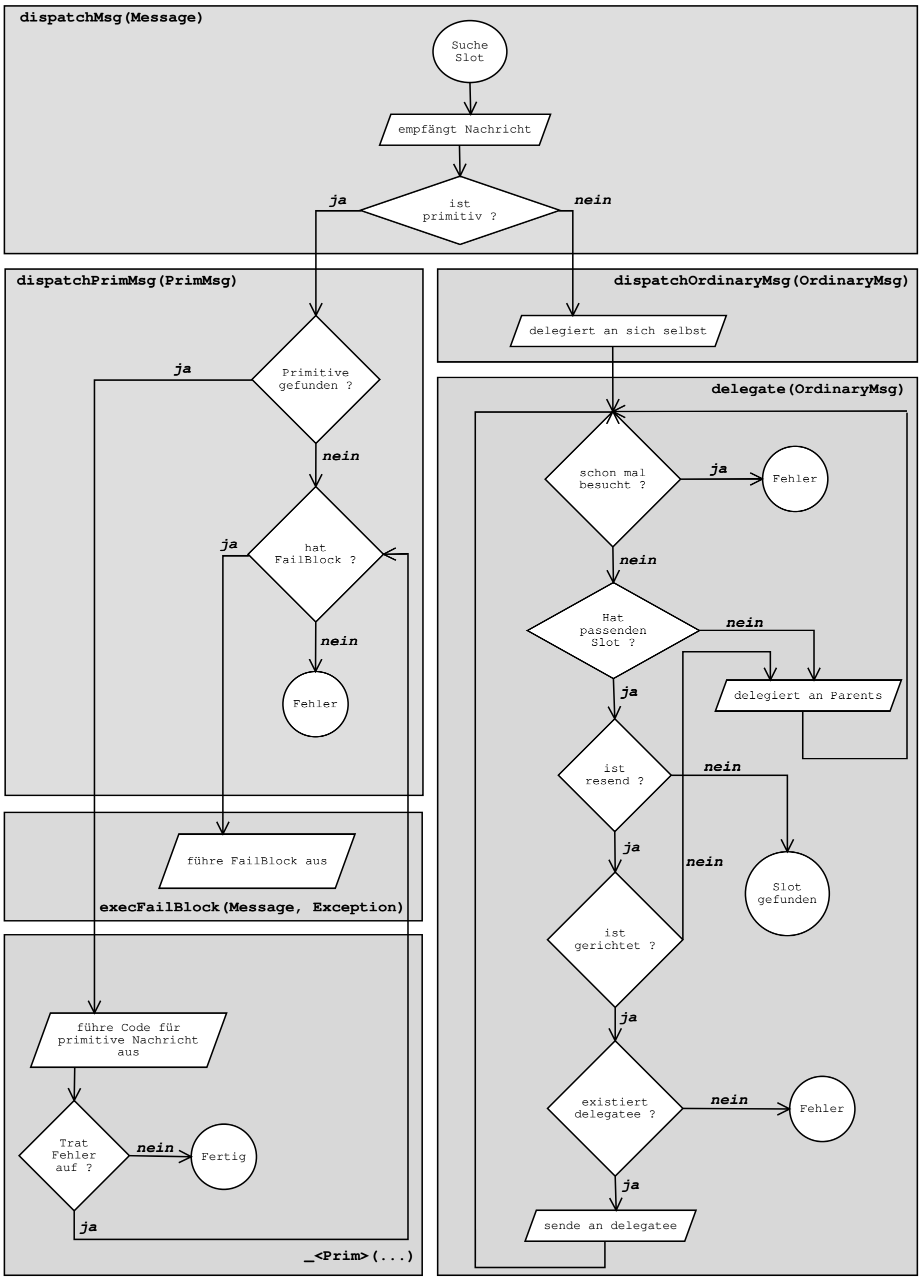
```
PrimMsg
----------------------------------------------------
#numPrim: int = 222
+SELF: int = 0
+ADDSLOTS: int = 1
+ADDSLOTSIFABSENT: int = 2
+AT: int = 3
 ...:
+TIMEREAL: int = 212
#primTable: String[]
#primID: int
#hasFailBlock: boolean = false
----------------------------------------------------
+stringToPrimID(primString:String): int
+hasFailBlock(primString:String): boolean
+primIDToString(ID:int): String
+PrimMsg(ID:int)
+PrimMsg(ID:int,hasFail:boolean)
+PrimMsg(ID:int,args:Vector)
+PrimMsg(ID:int,hasFail:boolean,args:Vector)
+hasFailBlock(): boolean
+getMessageID(): int
+getSelector(): String
+isResend(): boolean
+isDirectedResend(): boolean
+isIndirectedResend(): boolean
```

PrimMsg stellt eine primitive Nachricht dar. Sie hat statt einem String als Namen eine Nummer als ID. Welche Bedeutung eine ID hat, wird durch die statischen Konstanten wie zum Beispiel *ADDSLOTS* zum Ausdruck gebracht. Da es über 200 solcher Konstanten gibt, werden sie aus Platzgünden hier nicht alle aufgelistet. Eine solche Beschreibung findet sich im doc-Verzeichnie der Implementierung. *hasFailBlock* gibt an, ob diese Nachricht Block besitzt, der im Fall eines Fehlers ausgeführt werden soll und kann über die gleichnamige Methode abgefragt werden. *primID* ist die Nummer dieser Nachricht die mit *getMesageID* ausgelesen werden kann. Sie kann mit *primIDToString* in den dazugehörigen Namen konvertiert werden. Dabei greift diese Methode auf das Array *primTable* zu, das alle Namen beinhaltet. *isResend*, *isDirectedResend* und *isIndirectedResend* geben immer *false* zurück, da eine primitive Nachricht nicht per *resend* verschickt werden kann.

5.4 Grundlegende Verfahren in dSelf

Nachdem die wichtigsten Klassen einzeln vorgestellt wurden, soll nun ihr Zusammenspiel anhand einiger Verfahren in dSelf demonstriert werden.

5.4.1 Der Vererbungsalgorithmus

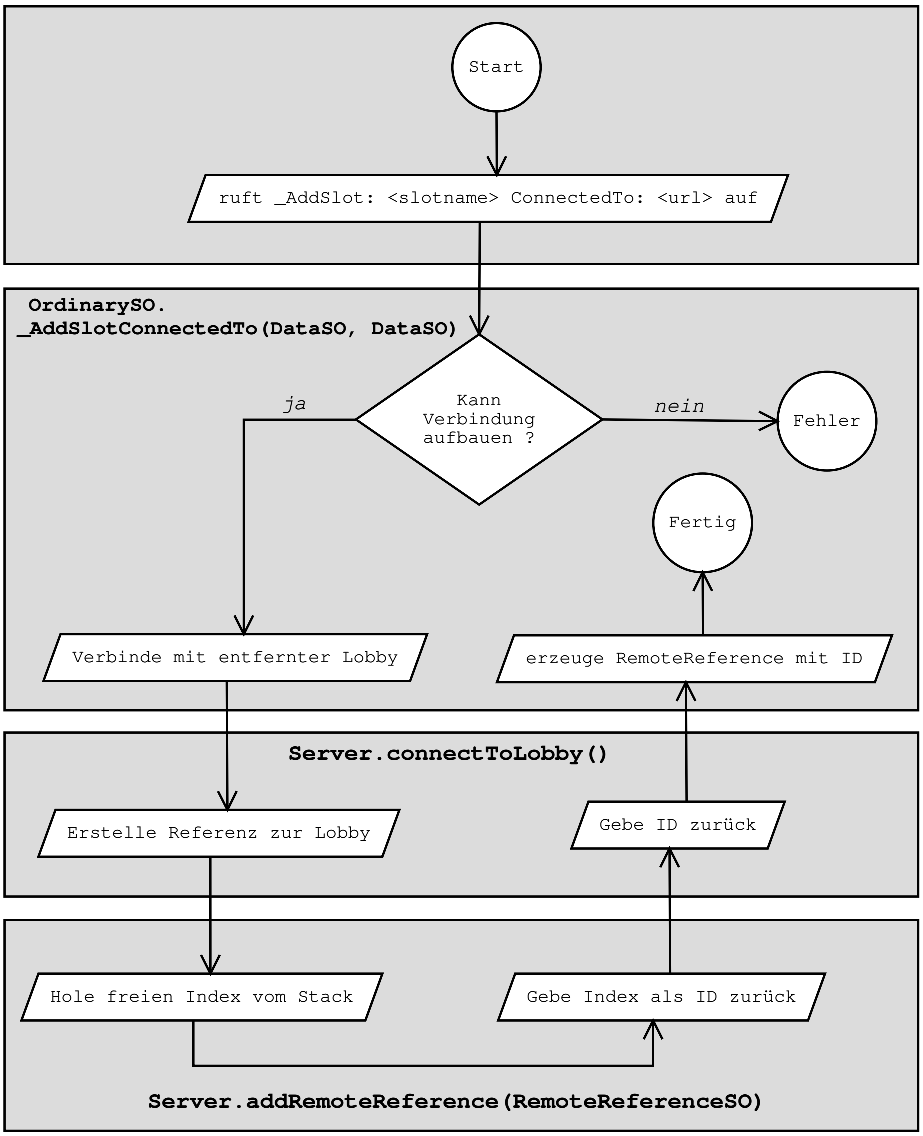


Einer der wichtigsten Algorithmen in dSelf ist der *lookup algorithm*, der die Vererbung realisiert. Das abgebildete Ablaufdiagramm zeigt seine Funktionsweise. Die grauen Kästen geben an, in welcher Methode der von *DataOrMethodSO* abgeleiteten Klassen welche Aktion ausgeführt wird.

In Worte gefaßt, funktioniert der Algorithmus auf folgende Weise:

> Wenn ein Nachrichtenobjekt an ein *DataOrMethodSO* geschickt wird, nimmt die Methode *dispatchMsg(Message)* diese entgegen. Zunächst wird geprüft, ob die Nachricht primitiv (vom Typ *PrimMsg*) oder gewöhnlich (vom Typ *OrdinaryMsg*) ist. Wenn sie primitiv ist, wird sie an *dispatchPrimMsg(PrimMsg)* weitergereicht. Dort angekommen, wird nachgeschaut, ob dieses Objekt für diese Nachricht eine passende Implementierung besitzt. Wenn nicht, dann führt sie den Fail-Block mit *execFailBlock(Message, Exception)* aus, falls sie einen besitzt, ansonsten wird eine Fehlermeldung ausgegeben. Wenn jedoch eine passende Implementierung gefunden wurde, dann wird die dazugehörige Methode aufgerufen. Solche Methoden haben immer einen Unterstrich als Präfix. Wenn während der Abarbeitung ein Fehler auftritt, wird dieser an den Fail-Block der primitiven Nachricht weitergegeben, insofern einer existiert.
>
> War die ursprünglich empfangene Nachricht eine gewöhnliche Nachricht, wird sie an die Methode *dispatchOrdinaryMsg(OrdinaryMsg)* weitergereicht. Die Aufgabe dieser Methode ist es, die Nachricht zum Delegieren weiterzugeben und die die gefundenen Objekte, die einem Vektor gesammelt wurden, auszuwerten. Wenn der Vektor leer ist, dann wurde kein passender Slot gefunden. Besitzt er hingegen mehr als ein Objekt, so ist die Nachricht mehrdeutig, insofern es sich bei den Objekten im Vektor nicht um ein und dasselbe Objekt handelt. Wenn genau ein Objekt gefunden wurde, so wird dieses evaluiert und sein Ergebnis zurückgegeben. Um an den beschriebenen Vektor zubekommen, müssen die Slots erst einmal gesucht werden. Dafür wird die Nachricht mit *delegate(OrdinaryMsg)* zuerst an sich selber delegiert.
>
> In *delegate(OrdinaryMsg)* wird zunächst geprüft, ob die Nachricht schon einmal an dieses Objekt per *delegate(OrdinaryMsg)* gesendet wurde. Ist dies der Fall, dann gibt es einen Zyklus im Vererbungsgraphen und eine Fehlermeldung wird ausgegeben. Wenn nicht, dann wird geprüft, ob es einen lokalen Slot gibt, dessen Namen dem Selektor der Nachricht entspricht. Wenn nicht, dann wird die Nachricht an alle Parentslots weiterdeligiert. Wenn doch, dann wird zunächst geprüft ob die Nachricht per *resend* verschickt wurde. Ist dies nicht der Fall, dann wurde der Slot gefunden und die Suche ist beendet. Ansonsten wird geprüft, ob die Nachricht gerichtet (*directed resend*) versandt wurde. Wurde sie ungerichtet versandt, wird die Nachricht an alle Parentslots des Objekts weitergereicht, und im gerichteten Fall muß geprüft werden, ob der angegebene Empfänger (*delegatee*) existiert. Wenn er existiert wird die Nachricht an ihn delegiert und wenn nicht wird ein Fehler ausgegeben.

5.4.2 Aufbau von verteilten Referenzen



Wie man eine Verbindung zu einer entfernten Lobby aufbaut. wurde bereits in Abschnitt 3.2.2 (Verteilte Objektreferenzen) vorgestellt. Hier soll dies nun an der konkreten Implementierung von dSelf demonstriert werden. In Worten ausgedückt passieren folgende Schritte:

> Um eine Verbindung aufzubauen, wird zunächst die Methode *_AddSlotConnectedTo* (oder auch *_AddSlotIfAbsentConnectedTo*) von *OrdinarySO* mit zwei Argumenten aufgerufen, von denen das

erste den Namen des Slots spezifiziert, der erzeugt werden soll, und das zweite eine URL zur entfernten dSelf Virtual Machine. Dort wird zunächst geprüft, ob eine Verbindung zu dieser VM aufgebaut werden kann. Dies kann scheitern, wenn zum Beipiel die URL falsch ist, oder der angegebene Slotname lexikalisch nicht korrekt geformt ist. Hat die Verbindungsaufnahme hingegen funktioniert, wird auf der Gegenseite die Methode *connectedToLobby* von *Server* aufgerufen. Dort wird eine Referenz auf die lokale Lobby erstellt und diese als Argument an *addRemoteReference(RemoteReferenceSO)* weitergegeben. Nun wird ein freier Index vom Stack *vectorLeaksStack* genommen und an dieser Position die Lobby eingetragen. Als Resultat wird eine ID, die diesem Index entspricht, über *connectToLobby* an *_AddSlotConnectedTo* zurückgegeben. Dort wird nun ein Objekt vom Typ *RemoteOrdinarySO* mit der gegebenen ID und der Referenz zum *Server* erzeugt. Über diese verteilte Referenz kann nun auf alle von der Lobby aus erreichbaren Slots zugegriffen werden, da alle Methodenaufrufe an das entfernte Objekt von *RemoteOrdinarySO* weitergereicht werden. Daher können entfernt Objekte wie lokale Objekte angesprochen werden.

Kapitel 6

Die primitiven Nachrichten von dSelf

Die folgenden primitiven Nachrichten sind in dSelf verfügbar. Primitive Nachrichten sind bereits in der Virtual Machine eingebaute Befehle, die alle mit einem Unterstrich beginnen, und können nicht vom Benutzer neu definiert werden. Einige primitive Nachrichten können scheitern, was zu einem Laufzeitfehler führt, wenn dieser nicht durch ein angefügtes *IfFail:* abgefangen wird. Folgende Fehlerursachen können in dSelf auftreten:

primitiveFailedError

Allgemeine Fehlermeldung, wenn z.B. ein Argument einen ungültigen Wert besitzt.

badTypeError

Wenn der Empfänger oder ein Argument einen falschen Typ besitzen

divisionByZeroError

Wenn eine Division durch Null durchgeführt wurde

badIndexError

Wenn in einem Vektor auf einen Index außerhalb des gültigen Bereichs zugegriffen wurde

securityError

Wenn versucht wird auf Resourcen zuzugreifen, die in der jeweiligen Umgebung (z.B. als Applet in einem Browser) nicht zugreifbar sind.

ioError

Wenn ein Fehler beim Zugreifen auf Dateien entsteht (z.B. bei *_RunScript*)

6.1 Primitive Nachrichten für Objekte vom Typ Integer

_IntAdd:

Addiert das Empfängerobjekt vom Typ Integer auf das Argumentobjekt vom gleichen Typ. Es wird nicht auf Über- oder Unterlauf getestet.

_IntAnd:

Das Empfänger- und das Argumentsobjekt (beide vom Typ Integer) werden bitweise mit AND verknüpft und das resultierende Objekt zurückgegeben.

_IntArithmeticShiftRight:

Das Empfängerobjekt vom Typ Integer wird um die Anzahl von Bits nach rechts geshiftet, die durch das Argumentobjekt vom Typ Integer angegeben werden. Das Vorzeichenbit bleibt erhalten.

_IntAsDouble

Konvertiert das Empfängerobjekt vom Typ Integer in ein Objekt vom Typ Double.

_IntAsFloat

Konvertiert das Empfängerobjekt vom Typ Integer in ein Objekt vom Typ Float.

_IntAsLong

Konvertiert das Empfängerobjekt vom Typ Integer in ein Objekt vom Typ Long.

_IntAsShort

Konvertiert das Empfängerobjekt vom Typ Integer in ein Objekt vom Typ Short. Es wird nicht auf Über- oder Unterlauf getestet.

_IntDiv:

Dividiert das Empfängerobjekt durch das Argumentobjekt (beide vom Typ Integer). Kann bei einer Division durch Null scheitern.

_IntEQ:

Prüft Empfänger- und Argumentobjekt auf Äquivalenz (beide vom Typ Integer). Wenn zwei Objecte *_IntEQ:* sind, dann sind sie auch *_Eq:* .

_IntGE:

Gibt *true* zurück, wenn das Empfängerobjekt größer oder gleich dem Argumentobjekt (beide vom Typ Integer) ist, ansonsten *false*.

_IntGT:

Gibt *true* zurück, wenn das Empfängerobjekt größer als das Argumentobjekt (beide vom Typ Integer) ist, ansonsten *false*.

_IntLE:

Gibt *true* zurück, wenn das Empfängerobjekt kleiner oder gleich dem Argumentobjekt (beide vom Typ Integer) ist, ansonsten *false*.

_IntLT:

Gibt *true* zurück, wenn das Empfängerobjekt kleiner als das Argumentobjekt (beide vom Typ Integer) ist, ansonsten *false*.

_IntLogicalShiftLeft:

Das Empfängerobjekt vom Typ Integer wird um die Anzahl von Bits nach links geshiftet, die durch das Argumentobjekt vom Typ Integer angegeben werden. Es wird nicht auf Überlauf getestet.

_IntLogicalShiftRight:

Das Empfängerobjekt vom Typ Integer wird um die Anzahl von Bits nach rechts geshiftet, die durch das Argumentobjekt vom Typ Integer angegeben werden. Das Vorzeichenbit bleibt erhalten.

_IntMod:

Berechnet das Empfängerobjekt (beide vom Typ Integer) modulo dem Argumentobjekt. Kann bei einer Division durch Null scheitern.

_IntMul:

Multipliziert das Empfängerobjekt mit dem Argumentobjekt (beide vom Typ Integer) und gibt das Resultat zurück. Es wird nicht auf Über- oder Unterlauf getestet.

_IntNE:

Test auf Ungleichheit. Gibt *true* zurück, wenn Empfänger und Argument (beide vom Typ Integer) unterschiedlich sind, ansonsten *false*.

_IntOr:

Das Empfänger- und das Argumentsobjekt (beide vom Typ Integer) werden bitweise mit OR verknüpft und das resultierende Objekt zurückgegeben.

_IntSub:

Subtrahiert das Argumentobjekt vom Empfängerobjekt (beide vom Typ Integer) und gibt das Resultat zurück. Es wird auf nicht Über- oder Unterlauf getestet.

_IntXor:

Das Empfänger- und das Argumentsobjekt (beide vom Typ Integer) werden bitweise mit XOR verknüpft und das resultierende Objekt zurückgegeben.

6.2 Primitive Nachrichten für Objekte vom Typ Short

_ShortAdd:

Addiert das Empfängerobjekt vom Typ Short auf das Argumentobjekt vom gleichen Typ. Es wird nicht auf Über- oder Unterlauf getestet.

_ShortAnd:

Das Empfänger- und das Argumentsobjekt (beide vom Typ Short) werden bitweise mit AND verknüpft und das resultierende Objekt zurückgegeben.

_ShortArithmeticShiftRight:

Das Empfängerobjekt vom Typ Short wird um die Anzahl von Bits nach rechts geshiftet, die durch das Argumentobjekt vom Typ Integer angegeben werden. Das Vorzeichenbit bleibt erhalten.

_ShortAsDouble

Konvertiert das Empfängerobjekt vom Typ Short in ein Objekt vom Typ Double.

_ShortAsFloat

Konvertiert das Empfängerobjekt vom Typ Short in ein Objekt vom Typ Float.

_ShortAsInt

Konvertiert das Empfängerobjekt vom Typ Short in ein Objekt vom Typ Integer.

_ShortAsLong

Konvertiert das Empfängerobjekt vom Typ Short in ein Objekt vom Typ Long.

_ShortDiv:

Dividiert das Empfängerobjekt vom Typ Short durch das Argumentobjekt, das ebenfalls vom Typ Short ist. Kann bei einer Division durch Null scheitern.

_ShortEQ:

Prüft Empfänger- und Argumentobjekt (beide vom Typ Short) auf Äquivalenz. Wenn zwei Objecte *_IntEQ:* sind, dann sind sie auch *_Eq:* .

_ShortGE:

Gibt *true* zurück, wenn das Empfängerobjekt größer oder gleich dem Argumentobjekt ist (beide vom Typ Short), ansonsten *false*.

_ShortGT:

Gibt *true* zurück, wenn das Empfängerobjekt größer als das Argumentobjekt ist (beide vom Typ Short), ansonsten *false*.

_ShortLE:

Gibt *true* zurück, wenn das Empfängerobjekt kleiner als oder gleich dem Argumentobjekt ist (beide vom Typ Short), ansonsten *false*.

_ShortLT:

Gibt *true* zurück, wenn das Empfängerobjekt kleiner als das Argumentobjekt ist (beide vom Typ Short), ansonsten *false*.

_ShortLogicalShiftLeft:

Das Empfängerobjekt vom Typ Short wird um die Anzahl von Bits nach links geshiftet, die durch das Argumentobjekt vom Typ Integer angegeben werden. Es wird nicht auf Überlauf getestet.

_ShortLogicalShiftRight:

Das Empfängerobjekt vom Typ Short wird um die Anzahl von Bits nach rechts geshiftet, die durch das Argumentobjekt vom Typ Integer angegeben werden. Das Vorzeichenbit bleibt erhalten.

_ShortMod:

Berechnet das Empfängerobjekt modulo dem Argumentobjekt (beide vom Typ Short). Kann bei einer Division durch Null scheitern.

_ShortMul:

Multipliziert das Empfängerobjekt mit dem Argumentobjekt (beide vom Typ Short) und gibt das Resultat zurück. Es wird nicht auf Über- oder Unterlauf getestet.

_ShortNE:

Test auf Ungleichheit. Gibt *true* zurück, wenn Empfänger und Argument (beide vom Typ Short) unterschiedlich sind, ansonsten *false*.

_ShortOr:

Das Empfänger- und das Argumentsobjekt (beide vom Typ Short) werden bitweise mit OR verknüpft und das resultierende Objekt zurückgegeben.

_ShortSub:

Subtrahiert das Argumentobjekt vom Empfängerobjekt (beide vom Typ Short) und gibt das Resultat zurück. Es wird nicht auf Über- oder Unterlauf getestet.

_ShortXor:

Das Empfänger- und das Argumentsobjekt (beide vom Typ Short) werden bitweise mit XOR verknüpft und das resultierende Objekt zurückgegeben.

6.3 Primitive Nachrichten für Objekte vom Typ Long

_LongAdd:

Addiert das Empfängerobjekt vom Typ Long auf das Argumentobjekt vom gleichen Typ. Es wird auf nicht Über- oder Unterlauf getestet.

_LongAnd:

Das Empfänger- und das Argumentsobjekt (beide vom Typ Long) werden bitweise mit AND verknüpft und das resultierende Objekt zurückgegeben.

_LongArithmeticShiftRight:

Das Empfängerobjekt vom Typ Long wird um die Anzahl von Bits nach rechts geshiftet, die durch das Argumentobjekt vom Typ Integer angegeben werden. Das Vorzeichenbit bleibt erhalten.

_LongAsDouble

Konvertiert das Empfängerobjekt vom Typ Long in ein Objekt vom Typ Double.

_LongAsFloat

Konvertiert das Empfängerobjekt vom Typ Long in ein Objekt vom Typ Float.

_LongAsInt

Konvertiert das Empfängerobjekt vom Typ Long in ein Objekt vom Typ Integer. Es wird nicht auf Über- oder Unterlauf getestet.

_LongAsShort

Konvertiert das Empfängerobjekt vom Typ Long in ein Objekt vom Typ Short. Es wird nicht auf Über- oder Unterlauf getestet.

_LongDiv:

Dividiert das Empfängerobjekt durch das Argumentobjekt (beide vom Typ Long). Kann bei einer Division durch Null scheitern.

_LongEQ:

Prüft Empfänger- und Argumentobjekt (beide vom Typ Long) auf Äquivalenz. Wenn zwei Objecte *_LongEQ:* sind, dann sind sie auch *_Eq:* .

_LongGE:

Gibt *true* zurück, wenn das Empfängerobjekt größer oder gleich dem Argumentobjekt (beide vom Typ Long) ist, ansonsten *false*.

_LongGT:

Gibt *true* zurück, wenn das Empfängerobjekt größer als das Argumentobjekt (beide vom Typ Long) ist, ansonsten *false*.

_LongLE:

Gibt *true* zurück, wenn das Empfängerobjekt kleiner als oder gleich dem Argumentobjekt (beide vom Typ Long) ist, ansonsten *false*.

_LongLogicalShiftLeft:

Das Empfängerobjekt vom Typ Long wird um die Anzahl von Bits nach links geshiftet, die durch das Argumentobjekt vom Typ Integer angegeben werden. Es wird nicht auf Überlauf getestet.

_LongLogicalShiftRight:

Das Empfängerobjekt vom Typ Long wird um die Anzahl von Bits nach rechts geshiftet, die durch das Argumentobjekt vom Typ Integer angegeben werden. Das Vorzeichenbit bleibt erhalten.

_LongMod:

Berechnet das Empfängerobjekt modulo dem Argumentobjekt (beide vom Typ Long). Kann bei einer Division durch Null scheitern.

_LongMul:

Multipliziert das Empfängerobjekt mit dem Argumentobjekt (beide vom Typ Long) und gibt das Resultat zurück. Es wird nicht auf Über- oder Unterlauf getestet.

_LongNE:

Test auf Ungleichheit. Gibt *true* zurück, wenn Empfänger und Argument (beide vom Typ Long) unterschiedlich sind, ansonsten *false*.

_LongOr:

Das Empfänger- und das Argumentsobjekt (beide vom Typ Long) werden bitweise mit OR verknüpft und das resultierende Objekt zurückgegeben.

_LongSub:

Subtrahiert das Argumentobjekt vom Empfängerobjekt (beide vom Typ Long) und gibt das Resultat zurück. Es wird nicht auf Über- oder Unterlauf getestet.

_LongXor:

Das Empfänger- und das Argumentsobjekt (beide vom Typ Long) werden bitweise mit XOR verknüpft und das resultierende Objekt zurückgegeben.

6.4 Primitive Nachrichten für Objekte vom Typ Float

_FloatAdd:

Addiert das Empfängerobjekt vom Typ Float auf das Argumentobjekt vom gleichen Typ. Es wird nicht auf Über- oder Unterlauf getestet. Maximale Werte sind positiv oder negativ Unendlich (Infinity).

_FloatAsDouble

Konvertiert das Empfängerobjekt vom Typ Float in ein Objekt vom Typ Double.

_FloatAsInt

Konvertiert das Empfängerobjekt vom Typ Float in ein Objekt vom Typ Integer.

_FloatAsLong

Konvertiert das Empfängerobjekt vom Typ Float in ein Objekt vom Typ Long.

_FloatAsShort

Konvertiert das Empfängerobjekt vom Typ Float in ein Objekt vom Typ Short.

_FloatCeil

Gibt den größten integralen Wert der größer oder gleich dem Empfängerobjekt (vom Typ Float) ist zurück, wobei nach positiv Unendlich gerundet wird. Das zurückgegebene Resultatobjekt ist ebenfalls vom Typ Float.

_FloatDiv:

Dividiert das Empfängerobjekt durch das Argumentobjekt (beide vom Typ Float). Kann bei einer Division durch Null scheitern.

_FloatEQ:

Prüft Empfänger- und Argumentobjekt auf Äquivalenz (beide vom Typ Float). Der Unterschied zu *_Eq:* liegt in der Unterscheidung zwischen *0.0 und -0.0* . Es gilt *0.0 _FloatEq: -0.0*, aber nicht *0.0 _Eq -0.0* .

_FloatFloor

Gibt den grösten integralen Wert der kleiner oder gleich dem Empfängerobjekt (vom Typ Float) ist zurück, wobei nach negativ Unendlich gerundet wird. Das zurückgegebene Resultatobjekt ist ebenfalls vom Typ Float.

_FloatGE:

Gibt *true* zurück, wenn das Empfängerobjekt größer oder gleich dem Argumentobjekt (beide vom Typ Float) ist, ansonsten *false*.

_FloatGT:

Gibt *true* zurück, wenn das Empfängerobjekt größer als das Argumentobjekt (beide vom Typ Float) ist, ansonsten *false*.

_FloatLE:

Gibt *true* zurück, wenn das Empfängerobjekt kleiner als oder gleich dem Argumentobjekt (beide vom Typ Float) ist, ansonsten *false*.

_FloatMod:

Berechnet das Empfängerobjekt modulo dem Argumentobjekt (beide vom Typ Float). Kann bei einer Division durch Null scheitern.

_FloatMul:

Multipliziert das Empfängerobjekt mit dem Argumentobjekt (beide vom Typ Float) und gibt das Resultat zurück. Es wird nicht auf Über- oder Unterlauf getestet. Maximale Werte sind positiv oder negativ Unendlich (Infinity).

_FloatNE:

Test auf Ungleichheit. Gibt *true* zurück, wenn Empfänger und Argument (beide vom Typ Float) unterschiedlich sind, ansonsten *false*.

_FloatPrintString

Gibt das Empfängerobjekt vom Typ Float als einen kanonischen String zurück.

_FloatPrintStringPrecision:

Gibt das Empfängerobjekt vom Typ Float als einen kanonischen String zurück. Das Argumentobjekt ist vom Typ Integer und spezifiziert die Anzahl der Nachkommastellen.

_FloatRound

Rundet das Empfängerobjekt vom Typ Float zum nächsten integralen Wert. 0.5 wird zur nächsten, geraden Zahl gerundet, also z.B. 1.5 und 2.5 jeweils zu 2. Das Resultatobjekt ist vom Typ Float.

_FloatSub:

Subtrahiert das Argumentobjekt vom Empfängerobjekt (beide vom Typ Float) und gibt das Resultat zurück. Es wird nicht auf Über- oder Unterlauf getestet. Maximale Werte sind positiv oder negativ Unendlich (Infinity).

_FloatTruncate

Rundet das Empfängerobjekt vom Typ Float zum nächsten integralen Wert. Gerundet wird gegen Null, also z.B. 1.6 zu 1.0 und -1.6 zu -1. Das Resultatobjekt ist ebenfalls vom Typ Float.

6.5 Primitive Nachrichten für Objekte vom Typ Double

_DoubleAdd:

Addiert das Empfängerobjekt vom Typ Double auf das Argumentobjekt vom gleichen Typ. Es wird nicht auf Über- oder Unterlauf getestet. Maximale Werte sind positiv oder negativ Unendlich (Infinity).

_DoubleAsFloat

Konvertiert das Empfängerobjekt vom Typ Double in ein Objekt vom Typ Float.

_DoubleAsInt

Konvertiert das Empfängerobjekt vom Typ Double in ein Objekt vom Typ Integer.

_DoubleAsLong

Konvertiert das Empfängerobjekt vom Typ Double in ein Objekt vom Typ Long.

_DoubleAsShort

Konvertiert das Empfängerobjekt vom Typ Double in ein Objekt vom Typ Short.

_DoubleCeil

Gibt den größten integralen Wert der größer oder gleich dem Empfängerobjekt (vom Typ Double) ist zurück, wobei nach positiv Unendlich gerundet wird. Das zurückgegebene Resultatobjekt ist ebenfalls vom Typ Double.

_DoubleDiv:

Dividiert das Empfängerobjekt durch das Argumentobjekt (beide vom Typ Double). Kann bei einer Division durch Null scheitern.

_DoubleEQ:

Prüft Empfänger- und Argumentobjekt (beide vom Typ Double) auf Äquivalenz. Der Unterschied zu *_Eq:* liegt in der Unterscheidung zwischen 0.0 und -0.0 . Es gilt *0.0 _DoubleEq: -0.0*, aber nicht *0.0 _Eq: -0.0* .

_DoubleFloor

Gibt den größten integralen Wert der kleiner oder gleich dem Empfängerobjekt (vom Typ Double) ist zurück, wobei nach negativ Unendlich gerundet wird. Das zurückgegebene Resultatobjekt ist ebenfalls vom Typ Double.

_DoubleGE:

Gibt *true* zurück, wenn das Empfängerobjekt größer oder gleich dem Argumentobjekt (beide vom Typ Double) ist, ansonsten *false*.

_DoubleGT:

Gibt *true* zurück, wenn das Empfängerobjekt größer als das Argumentobjekt (beide vom Typ Double) ist, ansonsten *false*.

_DoubleLE:

Gibt *true* zurück, wenn das Empfängerobjekt kleiner als oder gleich dem Argumentobjekt ist, ansonsten *false*.

_DoubleLT:

Gibt *true* zurück, wenn das Empfängerobjekt kleiner als das Argumentobjekt (beide vom Typ Double) ist, ansonsten *false*.

_DoubleMod:

Berechnet das Empfängerobjekt modulo dem Argumentobjekt. Kann bei einer Division durch Null scheitern.

_DoubleMul:

Multipliziert das Empfängerobjekt mit dem Argumentobjekt (beide vom Typ Double) und gibt das Resultat zurück. Es wird nicht auf Über- oder Unterlauf getestet. Maximale Werte sind positiv oder negativ Unendlich (Infinity).

_DoubleNE:

Test auf Ungleichheit. Gibt *true* zurück, wenn Empfänger und Argument (beide vom Typ Double) unterschiedlich sind, ansonsten *false*.

_DoublePrintString

Gibt das Empfängerobjekt vom Typ Double als einen kanonischen String zurück.

_DoublePrintStringPrecision:

Gibt das Empfängerobjekt vom Typ Double als einen kanonischen String zurück. Das Argumentobjekt ist vom Typ Integer und spezifiziert die Anzahl der Nachkommastellen.

_DoubleRound

Rundet das Empfängerobjekt vom Typ Double zum nächsten integralen Wert. 0.5 wird zur nächsten, geraden Zahl gerundet, also z.B. 1.5 und 2.5 jeweils zu 2. Das Resultatobjekt ist vom Typ Double.

_DoubleSub:

Subtrahiert das Argumentobjekt vom Empfängerobjekt (beide vom Typ Double) und gibt das Resultat zurück. Es wird nicht auf Über- oder Unterlauf getestet. Maximale Werte sind positiv oder negativ Unendlich (Infinity).

_DoubleTruncate

Rundet das Empfängerobjekt vom Typ Double zum nächsten integralen Wert. Gerundet wird gegen Null, also z.B. 1.6 zu 1.0 und -1.6 zu -1. Das Resultatobjekt ist ebenfalls vom Typ Double.

6.6 Primitive Nachrichten für Objekte vom Typ String

_BitSize

Das Empfängerobjekt muß vom Typ String sein und entweder *'integer'*, *'float'*, *'long'* oder *'double'* beinhalten. Der Rückgabewert ist vom Typ Integer und beinhaltet die Anzahl der Bits zum jeweiligen Datentyp.

_RunScript

Das Empfängerobjekt vom Typ String stellt die Datei dar, die mit dieser primitiven Nachricht geladen wird. Bevor die Datei eingeladen werden kann, werden der Verzeichnispfadname (siehe *_DirPath*) und der Dateiname zusammengefügt und nach einer gleichnamigen Klassendatei in diesem Verzeichnis gesucht. Wenn eine gefunden wird, wird getestet ob diese Datei ("*<name>.class*") jünger ist als die dazugehörige dSelfdatei ("*<name>.dSelf*") und in diesem Fall in die Virtual Machine dynamisch gelinkt. Falls nicht, wird die dSelfdatei geladen.

_StringAsByteVector

Konvertiert das Empfängerobjekt vom Typ String in einen ByteVector. Es ist zu Beachten, daß hierbei Unicodezeichen (2 Bytes) umgewandelt werden und die Umwandlung deshalb plattformabhängig ist !

_StringAt:

Gibt einen Integer zwischen 0 und 65535 zurück, der das Unicodezeichen an der durch das Argumentobjekt (vom Typ Integer) angegebenen Position im Empfängerobjekt repräsentiert, welches vom Typ String sein muß. Wenn der Index zu hoch oder zu niedrig ist, wird eine Fehlermeldung ausgegeben.

_StringConcatenate:

Verkettet das Empfängerobjekt vom Typ String mit dem Argumentobjekt, das ebenfalls vom Typ String sein muß und gibt es als Rückgabewert zurück.

_StringGetSubFrom:

Gibt den Suffix des Empfängerobjekts vom Typ String beginnend von der durch das Argumentobjekt (Integer) angegebenen Position zurück. Wenn der Index zu hoch oder zu niedrig ist, wird eine Fehlermeldung ausgegeben.

_StringGetSubFrom:To:

Gibt einen Teilstring aus dem Empfängerobjekt (String) beginnend an der durch das erste Argument (Integer) angegebenen Position bis zu der Stelle, die durch das zweite Argument (Integer) angegeben wird. Wenn der Index zu hoch oder zu niedrig ist, wird eine Fehlermeldung ausgegeben.

_StringPrint

Druckt den Inhalt des Empfängerobjekts auf dem Standardausgabegerät aus. Das Empfängerobjekt kann entweder vom Typ String oder ByteVector sein. Bei ByteVector ist zu beachten, daß hierbei in Unicodezeichen umgewandelt wird und die Umwandlung deshalb plattformabhängig ist !

_StringSize

Gibt die Länge des Empfängerobjekts vom Typ String als ein Objekt vom Typ Integer zurück.

6.7 Primitive Nachrichten für Objekte vom Typ ByteVector

_ByteAt:

Gibt einen Integer zwischen 0 und 255 zurück, der das Byte an der durch das Argumentobjekt (vom Typ Integer) angegebenen Position im Empfängerobjekt repräsentiert, welches vom Typ ByteVector sein muß. Wenn der Index zu hoch oder zu niedrig ist, wird eine Fehlermeldung ausgegeben.

_ByteAt:Put:

Speichert das Argumentobjekt hinter *Put:* (ein Integer) an dem angegebenen Index (ein Integer) im Empfängerobjekt vom Typ ByteVector. Der Rückgabewert ist das durch diese Nachricht modifizierte Empfängerobjekt. Es wird eine Fehlermeldung ausgegeben, wenn der Index zu hoch oder zu niedrig ist, oder wenn der Wert für das übergebene Byte nicht zwischen 0 und 255 liegt.

_ByteSize

Gibt die Anzahl der Elemente des Empfängerobjekts (ein ByteVector) zurück.

_ByteVectorAsString

Konvertiert das Empfängerobjekt vom Typ ByteVector in einen String. Es ist zu beachten, daß hierbei in Unicodezeichen umgewandelt wird und die Umwandlung deshalb plattformabhängig ist !

_ByteVectorCompare:

Es werden zwei Objekte vom Typ ByteVector miteinander verglichen und als Rückgabewert eine -1 (kleiner als), 0 (gleich) oder 1 (größer als) vom Typ Integer zurückgegeben. Wenn beide Vektoren identisch sind, wird eine 0 zurückgegeben, ansonsten werden die Bytes beider Vektoren paarweise von vorne nach hinten miteinander verglichen. Wenn dabei ein Byte des Empfängerobjekts größer ist als das Byte an derselben Position des Argumentobjekts, wird eine 1 zurückgegeben und im umgekehrten Fall -1. Sollten beide Vektoren unterschiedlich groß sein, wird eine -1 zurückgegeben, wenn die Anzahl der Elemente des Empfängervektors kleiner ist als die des Argumentvektors, und 1 wenn sie größer ist.

_ByteVectorConcatenate:

Verkettet das Empfängerobjekt vom Typ ByteVector mit dem Argumentobjekt, das ebenfalls vom Typ ByteVector sein muß und gibt es als Rückgabewert zurück.

_CloneBytes:Filler:

Cloned das Empfängerobjekt vom Typ ByteVector. Das Rückgabeobjekt ist ebenfalls vom Typ ByteVector und besteht aus der durch das erste Argument (Integer) angegebenen Anzahl von Bytes des Empfängerobjekts. Wenn die angegebene Anzahl größer ist als die Anzahl der Bytes im Empfängerobjekt, werden die restlichen Bytes durch das Argumentobjekt hinter *Filler:* aufgefüllt. Das zweite Argument muß zwischen 0 und 255 liegen und vom Typ Integer sein, sonst wird eine Fehlermeldung ausgegeben.

_CopyByteRangeDstPos:Src:SrcPos:Length:

Kopiert die nach *Length:* angegebene Anzahl (ein Integer) von Bytes aus dem nach *Src:* angegebenen ByteVector in das Empfängerobjekt (ein ByteVector) an die Position, die durch das erste Argument (ein Integer) angegeben wird, beginnend an der Stelle, die nach *SrcPos:* angegeben wird (ein Integer). Als Rückgabewert wird das durch diese Nachricht modifizierte Empfängerobjekt zurückgegeben. Es wird eine Fehlermeldung ausgegeben, falls auf einen Index außerhalb des gültigen Bereiches zugegriffen wird.

_StringPrint

Druckt den Inhalt des Empfängerobjekts auf dem Standardausgabegerät aus. Das Empfängerobjekt kann entweder vom Typ String oder ByteVector sein. Bei ByteVector ist zu beachten, daß hierbei in Unicodezeichen umgewandelt wird und die Umwandlung deshalb plattformabhängig ist !

6.8 Primitive Nachrichten für Objekte vom Typ ObjectVector

_At:

Gibt das Objekt an der durch das Argumentobjekt angegebenen Position zurück. Das Empfängerobjekt ist ein Vektor und das Argument vom Typ Integer. Wenn der Index zu hoch oder zu niedrig ist, wird eine Fehlermeldung ausgegeben.

_At:Put:

Speichert das Argumentobjekt hinter *Put:* an dem angegebenen Index (ein Integer) im Empfängerobjekt, welches ein Vektor von Objekten sein muß. Der Rückgabewert ist das durch diese Nachricht modifizierte Empfängerobjekt. Es wird eine Fehlermeldung ausgegeben, wenn der Index zu hoch oder zu niedrig ist.

_Clone:Filler:

Cloned das Empfängerobjekt vom Typ Vektor. Das Rückgabeobjekt ist ebenfalls vom Typ Vektor und besteht aus der durch das erste Argument (Integer) angegebenen Anzahl von Objekten des Empfängerobjekts. Wenn die angegebene Anzahl größer ist als die Anzahl der Objekte im Empfängerobjekt, werden die restlichen Objekte durch das Argumentobjekt hinter *Filler:* aufgefüllt.

_CopyRangeDstPos:Src:SrcPos:Length:

Kopiert die nach *Length:* angegebene Anzahl (ein Integer) von Objekten aus dem nach *Src:* angegebenen Vektor in das Empfängerobjekt (ein Vektor) an die Position, die durch das erste Argument (ein Integer) angegeben wird, beginnend an der Stelle, die nach *SrcPos:* angegeben wird (ein Integer). Als Rückgabewert wird das durch diese Nachricht modifizierte Empfängerobjekt zurückgegeben. Es wird eine Fehlermeldung ausgegeben, falls auf einen Index außerhalb des gültigen Bereiches zugegriffen wird.

_Size

Gibt die Anzahl der Elemente des Empfängerobjekts (ein Vektor) zurück.

6.9 Primitive Nachrichten für Datenobjekte

_AddSlot:ConnectedTo:

Fügt einen Slot zum Empfängerobjekt hinzu, der zur Lobby einer entfernten dSelf Virtual Machine referenziert. Der Name des Slots wird als String als erstes

Argument übergeben und muß lexikalisch einen korrekten Datenslot darstellen. Das zweite Argument ist ebenfalls ein String und muß folgende Form besitzen: '//hostname:port/vm_name'. Also zum Beispiel: '//127.0.0.1:1099/myVM'. Der Name des Hosts und die Nummer des Ports spezifizieren den Ort der RMI-Registry und *vm_name* ist der Name, unter dem sich die entfernte dSelf-VM hat registrieren lassen. Der Hostname und die Portnummer sind optional, so daß standardmäßig der lokale Host und die Nummer 1099 angenommen werden.

_AddSlotIfAbsent:ConnectedTo:

Macht dasselbe wie *_AddSlot:ConnectedTo:*, jedoch nur wenn kein gleichnamiger Slot bereits existiert.

_AddSlots:

Fügt alle Slots des Argumentobjekts zum Empfängerobjekt hinzu. Namensgleiche Slots beim Empfänger werden überschrieben.

_AddSlotsIfAbsent:

Fügt alle Slots des Argumentobjekts zum Empfängerobjekt hinzu. Namensgleiche Slots beim Empfänger werden nicht überschrieben.

_Clone

Gibt einen Klon des Empfängerobjekts zurück, der mittels *shallow copy* erzeugt wird. Wenn *_Clone* auf Integer, Long, Float, Double oder kanonische Strings angewendet wird, wird der Empfänger zurückgegeben und nicht kopiert. Klonen ist in dSelf der einzige Weg neue Objekte zu generieren.

_Define:

Definiert ein Objekt neu als das Objekt, das als Argument angegeben wird. Dazu werden alle bestehende Slots gelöscht und die Slots des Argumentobjekts hinzu kopiert (*shallow copy*).

_Describe

Gibt eine Beschreibung über das Empfängerobjekt aus. Die Beschreibung zum Objekt und die Beschreibungen zu den Slots entsprechen den Anmerkungen (*annotations*), die bei der Deklarierung dieses Objektes angegeben worden sind. Der Rückgabewert ist nil.

_Eq:

Test auf Identität. Gibt *true* zurück, wenn Empfänger und Argument dasselbe Objekt sind.

_GetSlotNames

Gibt einen Objektvektor mit allen Slotnamen des Empfängerobjekts zurück.

_ObjectID

Gibt eine eindeutige Nummer für das Empfängerobjekt zurück. Das Resultatobjekt ist vom Typ Integer.

_Perform:

Sendet die unäre Nachricht, deren Name im ersten Argument als String übergeben wird an das Empfängerobjekt. Beispiel: *x _Perform: 'foo'* ist gleichbedeutend wie *x foo*. Es können keine primitiven Nachrichten, die mit einem Unterstrich ('_') beginnen, versendet werden.

_Perform:With:{With:}

Sendet die *keyword*-Nachricht, deren Name im ersten Argument als String übergeben wird an das Empfängerobjekt. Die Argumente dieser Nachricht werden, je nach Anzahl der Argumente, durch 1 bis maximal 20 *With:* getrennt dahinter angefügt. Beispiel: *x _Perform: 'foo:Bar:' With: 1* With: nil ist identisch zu *x foo: 1 Bar: nil*. Es können keine primitiven Nachrichten, die mit einem Unterstrich ('_') beginnen, versandt werden. Wenn die Anzahl der Argumente nicht mit der erwarteten Anzahl von Argumenten der angegebenen Nachricht übereinstimmt, wird eine Fehlermeldung ausgegeben.

_PerformResend:

Sendet die unäre Nachricht, deren Name im ersten Argument als String übergeben wird an die parent slots des Empfängerobjekts (*indirected resend*). Beispiel: *x _PerformResend: 'foo'* ist gleichbedeutend wie *resend.foo* in Objekt *x*. Es können keine primitiven Nachrichten, die mit einem Unterstrich ('_') beginnen, versandt werden.

_PerformResend:With:{With:}

Sendet die keyword-Nachricht, deren Name im ersten Argument als String übergeben wird an die parent slots des Empfängerobjekts (*indirected resend*). Die Argumente dieser Nachricht werden, je nach Anzahl der Argumente, durch 1 bis maximal 20 *With:* getrennt dahinter angefügt. Beispiel: *x _PerformResend: 'foo:Bar:' With: 1 With: nil* ist identisch zu *resend.foo: 1 Bar: nil* in Objekt *x*. Es können keine primitiven Nachrichten, die mit einem Unterstrich ('_') beginnen, versandt werden. Wenn die Anzahl der Argumente nicht mit der erwarteten Anzahl von Argumenten der angegebenen Nachricht übereinstimmt, wird eine Fehlermeldung ausgegeben.

_Perform:DelegatingTo:

Sendet die unäre Nachricht, deren Name im ersten Argument als String übergeben wird an den Slot des Empfängerobjekts, der im zweiten Argument vom Typ String angegeben wird (*directed resend*). Beispiel: *x _Perform: 'foo' DelegatingTo: 'p'* ist gleichbedeutend wie *p.foo* in Objekt *x*. Es können keine primitiven Nachrichten, die mit einem Unterstrich ('_') beginnen, versandt werden.

_Perform:DelegatingTo:With:{With:}

Sendet die *keyword*-Nachricht, deren Name im ersten Argument als String übergeben wird an den Slot des Empfängerobjekts, der im zweiten Argument vom Typ String angegeben wird (*directed resend*). Die Argumente dieser Nachricht werden, je nach Anzahl der Argumente, durch 1 bis maximal 20 *With:* getrennt dahinter angefügt. Beispiel: *x _Perform: 'foo:Bar:' DelegatingTo: 'p' With: 1 With: nil* ist identisch zu *p.foo: 1 Bar: nil* in Objekt *x*. Es können keine primitiven Nachrichten, die mit einem Unterstrich ('_') beginnen, versandt werden. Wenn die Anzahl der Argumente nicht mit der erwarteten Anzahl von Argumenten der angegebenen Nachricht übereinstimmt, wird eine Fehlermeldung ausgegeben.

_Print

Druckt die Zusammensetzung aller Slots des Empfängerobjekts aus.

_RemoveAllSlots

Entfernt alle Slots des Empfängerobjekts.

_RemoveSlot:

Entfernt den Slot des Empfängerobjekts, dessen Namen als Argument vom Typ String übergeben wird. Wenn kein passender Slot gefunden wird, wird eine Fehlermeldung zurückgegeben.

6.10 Primitive Nachrichten für den Debugger

_DebugCUPOn

Schaltet den internen Debugger des Parsers ein, der vom Parsergenerator (CUP) zur Verfügung gestellt wurde. Es werden beim Parsieren die aktuellen Aktionen (shift, reduce) sowie die dazugehörigen Zustandsnummern ausgegeben. Es empfiehlt sich zusätzlich mittels *_DebugScannerOn* die aktuell gelesenen Token des Scanners darzustellen, da sonst nur die intern verwendeten Symbolnummern angegeben werden.

_DebugCUPOff

Schaltet den Parserdebugger wieder aus.

_DebugScannerOn

Schaltet den Scannerdebugger ein, der jedes erkannte Token ausgibt.

_DebugScannerOff

Schaltet den Scannerdebugger wieder aus.

_DebugSearchPathOn

Schaltet den Suchpfaddebugger ein, der beim delergieren von Nachrichten die Suchwege des *lookup algorithm* (Vererbungsalgorithmus von dSelf) angibt.

_DebugSearchPathOff

Schaltet den Suchpfaddebugger wieder aus.

_DebugFlatParseTreeOn

Generiert beim Parsieren einen Strukturbaum in geklammerter Notation und druckt ihn aus. Generiert beim Parsieren einen Strukturbaum in geklammerter Notation und druckt ihn aus.

_DebugFlatParseTreeOff

Schaltet den Strukturbaumgenerator mit geklammerter Darstellung wieder aus.

_DebugIndentedParseTreeOn

Generiert wie *_DebugFlatParseTreeOn* einen Strukturbaum, gibt ihn aber in einer eingerückten und damit leichter lesbareren Form aus.

_DebugIndentedParseTreeOff

Schaltet den Strukturbaumgenerator mit eingerückter Darstellung wieder aus.

6.11 Sonstige Primitive Nachrichten

_Credits

Gibt eine Meldung zur Virtual Machine aus.

_CurrentTimeString

Gibt ein Objekt vom Typ String zurück, der die aktuelle Zeit und das Datum angibt.

_DirPath

Gibt den aktuellen Verzeichnispfad aus, der als Wurzel für alle Dateien gilt, die mit *_RunScript* geladen werden.

_DirPath:

Setzt den aktuellen Verzeichnispfad auf den im Argument vom Typ String angegebenen Wert.

_GarbageCollect

Weist den Garbage-Collector an, sämtliche Speicherbereinigungen jetzt auszuführen.

_Memory

Gibt die Größe der zur Verfügung stehende Speichermenge zurück. Der Rückgabewert ist vom Typ Long.

_OperatingSystem

Gibt einen String zurück, der das Betriebssystem und den Prozessor der zugrundliegenden Plattform beschreibt.

_Quit

Beendet dSelf direkt.

_TimeReal

Gibt die Anzahl der Millisekunden, die seit Mitternacht am 1. Januar 1970 GMT vergangen sind. Der Rückgabewert ist vom Typ Long.

Kapitel 7

Ausblicke für mögliche Fortsetzungen

Da für diese Diplomarbeit nur eine beschränkte Zeitspanne (6 Monate) zur Verfügung stand, war es natürlich nicht möglich alles zu implementieren, was man sich wünschen würde. Es wurde daher Wert darauf gelegt, daß zumindestens die wichtigsten Sachen zur Verfügung stehen. Desweiteren wurde weniger auf Optimierungen als auf Funktionalität geachtet, da das System nachträglich an einigen Stellen sicherlich noch optimiert werden kann, während fehlende Basisfunktionen im Nachhinein sicher nicht einfach zuzufügen sind. Im folgenden soll daher auf mögliche Erweiterungen für dSelf eingegangen werden sowie die dazu nötigen Umstellungen. Die Betonung liegt hier auf *mögliche*, da einige bei näherer Betrachtung sich als nicht sehr günstig herrausstellen können.

7.1 Priorisierte Mehrfachvererbung und Kapselung

In [3] wurde die priorisierte Mehrfachvererbung (*Prioritized multiple inheritance*) für die Sprache SELF vorgestellt. Prinzipiell unterscheidet man zwischen ungeordneter (*unordered inheritance*) und geordneter Mehrfachvererbung (*ordered inheritance*). Unter geordneter Vererbung versteht man eine festgelegte Reihenfolge bei der Suche durch den *lookup algorithm* nach einem gegebenen Slot. Dies erreicht man, indem die Parentslots durch zusätzliche Sternsymbole unterschiedlich priorisiert werden, wie in diesem Beispiel:

```
object = (|
    parent1*   = (...).
    parent2**  = (...).
    parent3*** = (...).
    ...
|)
```

Hier werden 3 Parentslots mit unterschiedlicher Priorisierung deklariert. Der Slot mit den meisten Sternsymbolen besitzt die niedrigste Priorität und würde daher erst dann besucht, wenn die Suche über *parent1* und *parent2* erfolglos war. Der Grund, warum diese Form der Mehrfachvererbung nicht übernommen wurde, ist schon in Abschnitt 3.1.1 anhand des dortigen Zitates dargestellt worden, deshalb ist eine solche Erweiterung wohl eher nicht zu empfehlen.

Die Kapselung wurde im selben Dokument wie die priorisierte Mehrfachvererbung beschrieben. Dabei handelt es sich um zusätzliche Symbole "^" und "_" um die Sichtbarkeit von Slots einzuschränken. Hierbei entspricht das Circumflex einer *public* und der Unterstrich einer *private* Sichtbarkeit. Für Zuweisungsslots gibt es noch eine Kombination aus beiden Symbolen, wobei eines den Zuweisungsslot selber und das andere den zugehörigen Datenslot betrifft. Hierzu ein Beispiel:

```
object = (|
      ^ slot1 = (^ 42).
      _ slot2 = 'I'm private'.
    ^ _ slot3 <- 1.
    _ ^ slot4 <- 'foo'
|)
```

Hier repräsentiert *slot1* einen öffentlichen Methodenslot und *slot2* einen privaten Datenslot. *slot3* deklariert einen öffentlichen Datenslot *slot3* und einen privaten Zuweisungsslot *slot3:*, sowie *slot4* einen privaten Datenslot mit einem öffentlichen Zuweisungsslot. Die Begründung, warum Kapselung nicht aufgenommen wurde ist dieselbe wie bei der priorisierten Mehrfachvererbung.

7.2 Mirrorobjekte

Mirrorobjekte sind spezielle Objekte in SELF, mit denen man die Beschaffenheit von anderen Objekten untersuchen kann. Man kann solche Mirrors in gewisser Weise mit den Reflectionklassen von Java vergleichen, doch Mirrors bieten noch zusätzlich die Möglichkeit, diese Objekte zusätzlich auch zu modifizieren. Aus zeitlichen Gründen wurden Mirrors nicht mit aufgenommen, da diese in einer verteilten Umgebung nicht trivial zu realisieren sind. Man müßte zunächst untersuchen, wie ein Mirror aussieht, der auf ein entferntes Objekt abbildet, und die Möglichkeit zur Modifikation über die Grenzen der lokalen dSelf Virtual Machine anbieten. Um solche Modifikationen dennoch in dSelf durchführen zu können, gibt es die primitive Methode *_GetSlotName*, durch die man einen Objektvektor mit Strings erhält, die die Namen aller Slots des Empfängerobjekts spezifizieren. Diese Namen kann man zum Beispiel dann in Verbindung mit *_RemoveSlot:* benutzen.Vom Komfort her ist diese Lösung aber sicher nicht mit Mirrorobjekten zu vergleichen.

7.3 Erweiterung um Nebenläufigkeit

In dSelf gibt es keine Prozeße wie in SELF. Es wäre aber erstrebenswert, auch solche Prozeße in dSelf einzuführen. Dazu müßte man die dSelf Virtual Machine um primitive Nachrichten erweitern, die einen neuen Prozeß starten und wie in SELF einen Scheduler implementieren, der diese verwaltet. Desweiteren müssen Konzepte zur Synchronistation wie Locks oder Semaphores eingeführt werden.

7.4 Aufbau einer dSelf-World

In dSelf wird wie in SELF auch von der dSelf Virtual Machine nur eine minimale Umgebung (initiale Welt) zur Verfügung gestellt, die nur die primitiven Objekte und Nachrichten beinhaltet. Alles andere wird über ein Skript hinzugeladen (bei SELF mit init.self). Es gibt in dSelf nur ein Skript namens *smallWorld.dSelf*, das nur eine kleine Anzahl von Objekten und Methoden wie zum Beispiel "+" statt *_IntAdd:* zur Verfügung stellt. Diese Welt ließe sich noch weiter ausbauen.

7.5 Mobile gewöhnliche Objekte

In dSelf verbleiben alle gewöhnlichen Objekte ihre gesamte Lebenszeit dort, wo sie definiert wurden. Dies stellt eine Einschränkung dar, die unterschiedliche Auswirkungen auf das dSelf-System hat.

Die erste Auswirkung betrifft die Effizienz, denn wenn man weiß, daß auf ein Objekt in Zukunft primär von einer entfernten dSelf Virtual Machine zugegriffen werden soll, so wäre es angebracht dieses Objekt auf jene zu verschieben. Die folgenden Zugriffe wären dann lokal und würden nicht durch viel Kommunikation das Netz belasten. Nachdem die Arbeit getan ist, könnte man das Objekt wieder zu seinem Ursprungsort zurückbringen. Es wäre auch eine Automatisierung eines solchen Vorgangs denkbar, indem eine Art Netzmanager statistisch ermittelt, wo die höchste Kommunikation zwischen Objekten besteht und diese dann zusammen auf einen Rechner verlegt.

Eine andere Auswirkung betrifft die Lebensdauer einer dSelf Virtual Machine. Sobald eine dSelf-VM beendet wird, werden mit ihr auch alle ihre Objekte gelöscht. Daher darf man beim derzeitigen dSelf-System eine dSelf-VM erst dann beenden, wenn man keines seiner Objekte mehr benötigt. Bei mobilen, gewöhnlichen Objekten könnte man diese beim Programmende einfach zu der VM bringen, die sie noch benötigt.

Der Grund, warum es so etwas in dSelf nicht gibt ist rein technischer Natur. Wenn man nämlich ein gewöhnliches Objekt verschieben möchte, muß man auch alle Referenzen auf dieses Objekt "umbiegen" auf die neue Position. Um dies zu machen, muß man aber zunächst wissen, von welchen anderen Objekten dieses Objekt überhaupt referenziert wird. Dazu benötigt man dann bidirektional Referenzen, wovon eine (wie bisher) das zu verschiebende Objekt zeigt und eine von diesem Objekt zurück, um diese Referenz auf das neue Ziel zu setzten. Dies erhöht zum einen den Speicheraufwand und zum anderen die Effizienz, da eine zusätzliche Verwaltung notwendig ist. Außerdem muß man sich dann auch um die *Garbage Collection* in Java selber kümmern, da ein nicht mehr referenzierendes Objekt dem referenzierten Objekt Bescheid sagen müßte, daß es sein Rückreferenz löschen soll, damit es vom *Garbage Collector* gelöscht werden kann. Es stellt sich insgesamt also die Frage, ob man nicht eher mehr an Effizienz verliert als gewinnt.

Bei der Idee, Objekte zum Lebensende der dSelf-VM zu kopieren, stellt sich die Frage in welchem Umfang dies geschehen soll. Wenn man ein gewöhnliches Objekt dann verschieben möchte, müßte man auch alle lokalen Objekte, auf die er referenziert ebenfalls mitkopieren. In der Praxis ist es aber häufig so, daß auch die lokale Lobby direkt oder indirekt über mehrere Parentobjekte davon betroffen ist, was bedeutet, daß man unter Umständen fast den kompletten Inhalt der dSelf-VM kopieren muß.

Allgemein betrachtet wird erkennbar, daß die Absichten der Objektorientierung den Erfordernissen von mobilen, gewöhnlichen Objekten widersprechen. Um ein Objekt effizient verschieben zu können, muß es möglichst autonom sein, also so wenig Referenzen zu anderen Objekten wie möglich besitzen. In der Objektorientierung strebt man aber eine Wiederverwertung von möglichst vielen Eigenschaften an, die man dann vererbt. Diese Verebung führt aber zu vielen Abhängigkeiten, die von vielen Objekten geteilt werden und somit zu einer niedrigen Autonomie.

7.6 Aufruf von Javaobjekten in dSelf

In dSelf ist es nicht möglich auf Javaobjekte zuzugreifen, wobei dies sicher wünschenswert wäre. Um dies zu realisieren, müßte man zuerste die Frage beantworten, was zum Beispiel eine Javaklasse im klassenlosen Konzept von dSelf bedeutet und wie man sie anspricht. Da die Sprachen auch syntaktisch unterschiedlich sind, würde es sich wohl anbieten spezielle dSelf-Objekte einzuführen, die eine Art "Wrapper" um ein Javaobjekt bilden. Diese könnten dann für jede Javamethode einen dSelf-Methodenslot anbieten, der die Argumente und Resultate einfach weiterreicht. Dies ließe sich wohl am Besten wie in SELF über Proxyfunktionen erledigen. Des weiteren wäre es überlegenswert, ob man eine Art "Wrappergenerator" entwickeln könnte, der den Quellcode einer Javaklasse einliest und daraus automatisch ein umgebendes dSelf-Objekt erstellt. Diese Art von Objekten hätten dann aber wohl eine andere Semantik als gewöhnliche dSelf-Objekte, da Konzepte wie *dynamic inheritance* nicht auf Java übertragbar sind.

7.7 Optimierung des Compilers

Der Javacode, der vom Compiler erzeugt wird, kann sehr schnell ziemlich groß werden, da er viele Casting-Anweisungen beinhaltet. Dies erzeugt Code für eine Methode, der zum Beipiel so aussieht:

```
return (new ExprList((ExprList)(new ExprList((CodeExpr)
(new PrimMsgExpr(null, 1/* _AddSlots: */, ((CapKeywordExprList)
(new CapKeywordExprList())).getVectorWith((CodeExpr)
(new OrdinarySOExpr((SlotDeclarations)(((SlotDeclarations)
(new SlotDeclarations((SlotDecl)(new DataSlotDecl("lobby",
false, false, false, (CodeExpr)(new PrimMsgExpr(null,
PrimMsg.SELF)), null))))).withSlotDeclarations((SlotDeclarations)
(new SlotDeclarations()))))))), false)))), (CodeExpr)
(new PrimMsgExpr((CodeExpr)(new OrdinaryMsgExpr(null,"lobby")),
1/* _AddSlots: */, ((CapKeywordExprList)(new CapKeywordExprList())).
getVectorWith((CodeExpr)(getObj0())),false))));
```

Wie man sieht, wird hier sehr häufig gecastet, obwohl dies, abgesehen vom letzten Cast in *getVectorWith((CodeExpr)(getObj0())),false))))*, jeweils gar nicht notwendig wäre, wie zum Beipiel in *(CapKeywordExprList)(new CapKeywordExprList())*. Der Grund dafür liegt in der Zerlegeung des Codes in einzelne Methoden. Potentiell kann jeder erzeugte Teilausdruck des Parsers der erste Teilausdruck einer Methode sein. Deshalb muß der jeweilge Typ dieses Teilausdrucks vor dem Methodenaufruf gecastet werden, damit es zu keinen Typfehlern kommt. Daher setzt der dSelf-Compiler vor jeden Teilausdruck prophylaktisch einen Cast-Ausdruck wie hier *(CodeExpr)* vor *getObj0()* und erzeugt dadurch langen Javacode. Um dies zu optimieren muß man allerdings den erzeugten Code nachträglich bearbeiten, also nochmals parsieren (mit einer Javagrammatik) und die überflüssigen Casts entfernen, was kein triviales Problem ist. Desweitern stellt sich die Frage, inwieweit der Compiler von Java optimiert und selber bereits diese Casts herausfiltert.

7.8 Optimierung der dSelf-Nachrichtenverarbeitung

In dSelf werden alle gewöhnlichen Nachrichten durch Strings und primitiven Nachrichten durch Zahlen repräsentiert. Während dies für primitive Nachrichten leicht machbar war, da ihre Anzahl bereits bekannt ist, stellt dies für ihre gewöhnlichen Pendants ein Problem dar.

Lokal betrachtet bietet sich dafür eine Symboltabelle an, wie man sie von anderen Programmiersprachen kennt. Hierbei würde man beim definieren eines Slots in einer Liste von Slotnamen nachschauen, ob dieser Name bereits dort existiert. Ist dies nicht der Fall, dann wird dieser Name neu in die Tabelle aufgenommen und sein Index als ID zurückgegeben. Falls der Name schon existiert wird die bereits existierende ID zurückgegeben und ein Zähler erhöht, der angibt, wieviele Slots unter dem Namen bereits registriert sind. Wenn nun beim *lookup algorithm* ein Slot gesucht werden soll, würde sich der Aufwand auf das Vergleichen von Zahlen beschränken und nicht wie bisher auf Strings, was wesentlich schneller geht. Wenn der Name hingegen gar nicht in der Symboltabelle gar nicht vorhanden ist, kann man sich die Suche sogar ganz sparen, da ein zur Nachricht passender Slot gar nicht gefunden werden kann. Das Suchen einer ID zu einer gegebenen Nachricht ließe sich mit linearem Aufwand realisieren, indem man (ähnlich wie ein Scanner) alle Slotnamen in einer Baumstruktur speichern würde.

Wenn man das Ganze allerdings in einer verteilten Welt betrachtet, dann gibt es auch mehrere dSelf-VM und somit mehrere Symboltabellen. Wenn nun eine solche gewöhnliche Nachricht mit ihrer ID zu einer entfernten dSelf Virtual Machine gesendet würde, hätte seine ID dort keine Bedeutung, da sie auf einer fremden Symboltabelle beruht. Nun müßte man für solche Fälle Vorkehrungen treffen, indem man zum Beispiel die einzelnen Tabellen synchronisiert. Dies wäre allerdings mit hohem Aufwand verbunden, da man alleine für die Konsistenzerhaltung viel Kommunikation betreiben müßte. Eine andere Möglichkeit wäre es, den Selektor der Nachricht beim Verlassen der dSelf-VM wieder in einen String umzuwandeln, der dann auf der entfernten VM wieder in eine ID (basierend auf seiner Symboltabelle) umgewandelt wird. Dabei wäre auch der Fall zu betrachten, bei dem eine Nachricht über mehrere Zwischen-VMs geschickt wird, wobei die zwischengeschalteten VMs den Namen nicht auflösen dürften, da er nicht für sie gedacht ist und somit auch ihre Symboltabelle nicht betrifft.

Literaturverzeichnis

[1] The SELF 4.0 Programmer's Reference Manual von Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, Mario Wolczko. Veröffentlicht von Sun Microsystems, Inc. and Stanford Unversity, 1995

[2] Prototyped-Based Application Construction Using Self 4.0 von Mario Wolczko und Randall B. Smith. Veröffentlicht von Sun Microsystems, Inc. and Stanford Unversity, 1995

[3] Parents are Shared Parts of Objects: Inheritance and Encapsulation in Self von Craig Chambers, David Ungar, Bay-Wei Chang und Urs Hölzle. Veröffentlicht in: LISP AND SYMBOLIC COMPUTATION: An International Journal, 4, 3, 1991

[4] Programming as an Experience: The Inspiration for Self von Randall B. Smith und David Ungar, Sun Microsystems Laboratories

[5] References CUP (v0.10k) LALR Parser Generator for Java von Scott E. Hudson, Graphics Visualization and Usability Center, Georgia Institute of Technology, July 1999. http://www.princeton.edu/~appel/modern/java/CUP/

[6] References JFlex Version 1.3, The Fast Lexical Analyzer Generator for Java von Gerwin Klein, Oktober 2000. http://www.jflex.de