Squeak 4
Smalltalk-Hacking

Heiko Schröder

22. Februar 2006

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Sicherlich möchtest Du gleich loslegen. Wir werden das jetzt auch tun. Aber nicht, ohne eine Warnung auszusprechen: Wenn es irgendeinen großen Fehler gibt, den Du machen kannst, so ist dies Code-Hacking. So etwas ist aus anderen Sprachen, wie zum Beispiel C, bekannt. Natürlich funktiniert das auch auf den ersten Blick in Squeak. Aber, wie Mark GUZDIAL in seinem sehr guten Buch *Squeak, Designing Multimedia Applications* sagt: die Probleme tauchen später auf, wenn es darum geht, den geschriebenen Code zu verwalten und anderen verfügbar zu machen.

Doch durchbrechen wir einmal den guten Stil und schauen uns an, dass Sqeuak sich so verhalten kann, wie wir es gerne hätten. Dem Smalltalk-Balloon geht es dabei nicht besonders gut, wie Du siehst. Aber – naja. Es ist sicherlich wichtig, die einzelnen Werkzeuge einmal kennenzulernen. Schließlich sieht ein Tischler auch nicht erst dann eine Kreissäge zu ersten Male, wenn er ein konkretes Projekt vor sich hat.

Warum sollen wir nicht einmal statt einer Reise in die Wildnis, eine wilde Reise durchführen. Eine Art »Last Minute Tour«. Wir werden erstaunlich viel erleben. Doch wollen wir bescheiden bleiben. Am Ende wissen kennen wir sozusagen die Töne der Tonleiter, wissen aber noch nicht, wie man ein gutes Stück schreibt. Allerdings: Wenn wir sozusagen, in irgendeines der Squeak-Instrumente etwas hineinblasen, kommt ein Ton heraus.

2 Erinnerung an die Grundlagen

Wir erinnern uns an die Dinge, die wir im Kapitel Kapitel 1 besprochen haben:

```
sveta gib:marek bitte:butter und:tee.
```

An Objekte werden *Nachrichten* gesendet. In diesem Falle ist es das »Objekt« *sveta*. Der Name des Empfängers (*receivers*) steht immer am Beginn der Nachricht. Der zweite Teil ist der Nachrichtenselektor. In diesem Falle *gib:bitte:und:*. Hinter den Doppelpunkten stehen die Namen der für die Nachricht wichtigen Objekte, die mit ihr übergeben werden (*Argumente*). Ein Nachrichtenselektor muss nicht unbedingt Argumente besitzen. Zum Schluss folgt noch der Punkt. Obwohl er nur dann notwendig ist, wenn eine weitere Nachricht folgt, werden wir ihn zunächst *immer* setzen.

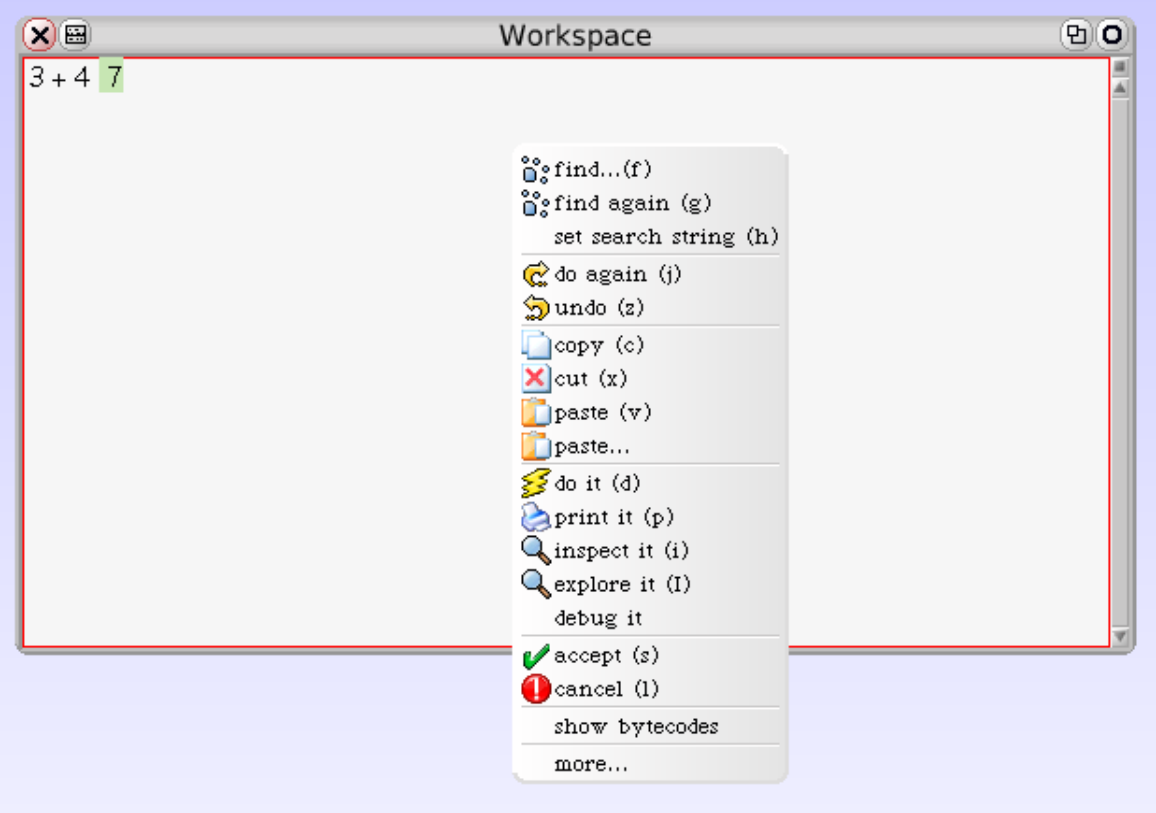

Abbildung 1: Workspace

Erstens werden zunächst zu klären haben, mit welchem Werkzeug wir an Objekte Nachrichten versenden können. Dies ist der so genannte *Workspace*.

Nach dem Absetzen der Nachricht ruft *sveta* in ihrem Inneren ein Programm aus, das zu dem Selektor *gib:bitte:und:* passt. Dieses Unterprogramm ist *svetas* Methode. Sie trägt die Bezeichnung *gib:objekt1 bitte:objekt2 und:objekt3*, wobei *objekt1, objekt2* und *objekt3* wieder Namen als *Platzhalter* für die übergebenen Objekte darstellen.

Zweitens werden wir also zu klären haben, wie wir in *sveta* eine Methode einbauen können. Du erinnerst Dich aber: Jedes Objekt ist eine Instanz einer Klasse. Nicht in *sveta* werden die Methoden eingebaut, sondern in die Klasse. Wir müssen also zwei Dinge klären: Wie erzeugen wir eine Klasse und wie bauen wir in sie eine Methode ein. Dabei lernen wir das zweite Werkzeug, den *System-Browser* kennen.

Schließlich erinnern wir uns, dass *sveta* grundsätzlich antwortet, wenn sie die Nachricht versteht. Drittens werden wir uns also darum kümmern müssen, wie wir die Antwort betrachten. Dafür gibt es zwei Möglichkeiten: Das *zum Schluss* zurückgegebene Objekt kann abermals im Workspace erscheinen. Aber eben nur das letzte dieser Unterhaltungen, falls mehrere Nachrichten abgesetzt wurden. Soll *sveta* zwischendurch eine Information in Form einer Zeichenkette (Text) zurückgeben, so geschieht dies in einem separaten Fenster, dem Transcript.

Nun sind wir endlich so weit, dass wir beginnen können.

3 Workspace

Ziehe nun den Workspace aus der Lasche »Werkzeugleiste« auf den Desktop. Alternativ kannst Du auch ALT-k ausführen. Das sieht noch nicht sehr eindrucksvoll aus. Ein graues Fenster, in dem der Cursor leise vor sich hinblinkt. Keine Buttons zum Anklicken. Nichts. O, doch. Die Fähigkeiten sind natürlich in dem Kontextmenü versteckt, das Du über @g Dir holen kannst.

Machen wir zunächst etwas *ganz* Einfaches und schauen nach, ob Squeak rechnen kann. Schreibe in den Workspace die weltbewegende Rechnung

```
3+4.
```

Wieso passiert nichts? Ja, woher *weiß* Squeak, dass Du diese Nachricht absetzen willst? Zunächst steht Deine Absicht ja nur auf dem Bildschirm. Es ist vergleichbar mit einer Notiz auf einem Einkaufszettel. Du stellst Dich sicherlich auch nicht an einen Gemüsestand und wartest, bis die Bedienung errät, was Du auf Deinem Zettel stehen hast.

Die »Bedienung« von Squeak ist die Virtual Machine (VM). Um ihr etwas zu sagen, musst Du zwei Schritte vornehmen:

1. Zunächst ist derjenige Teil des Textes zu markieren, der der VM mitgeteilt werden soll. Dies kann auf dreierlei Art und Weise geschehen:
   (a) Du ziehst die Maus wie in jeder gewöhnlichen Textverarbeitung über den Text.
   (b) Du wählst die ganze Zeile mit ALT-c aus.
   (c) Du wählst den ganzen im Workspace befindlichen Text mit ALT-a aus.
2. Danach folgt die eigentliche Mitteilung. Dies kann auf zweierlei Art geschehen:
   (a) Du sendest den Text der VM durch ein so genanntes *doIt* mit ALT-d.
   (b) Du sendest den Text der VM durch ein so genanntes *printIt* mit ALT-p

Führe nun mit unserer Rechnung diese Schritte durch. Falls Du als Erstes ALT-d ausprobierst, wirst Du vielleicht ein wenig enttäuscht sein. Es passiert *gar* nichts. Doch, es passiert sehr wohl etwas. Die VM hat dem Objekt 3 die Nachricht +4 gesendet. Dieses Objekt hat auch geantwortet, denn sonst wärst Du über diesen Mißstand unterrichtet worden. Dass *kein* Fehlerfenster aufgetaucht ist, stellt also ein gutes Zeichen dar.

Ach, Du wolltest das zurückgegebene Objekt auch *sehen*. Ja, das musst Du der VM natürlich sagen. Das letzte Objekt wird nur dann zurückgegeben, wenn Du ein *printIt* mit ALT-p durchgeführt hast. Denke wieder an den Vergleich mit dem Einkaufszetter. Die Objekte – und in diesem Falle die VM – können keine Gedanken lesen.

Gib jetzt eine zweite Rechnung *3-4* ein, wobei Du die erste bitte nicht löschst. Der Workspace soll also so aussehen[1]:

```
3+4.
3-4
```

[1] Falls noch die 7 von eben in dem Workspace leuchtet, lösche sie bitte.

Bei der zweiten Rechnung haben wir den Punkt weggelassen. Er ist ja nur dann nötig, wenn danach noch eine weitere Nachricht folgt. Wähle jetzt den ganzen Text aus und führe ein *printIt* durch. Ein kleines Lehrstück: es wird *nur* das letzte Objekt zurückgegeben. Die Nachricht *printIt* wird immer an die VM gesendet. Und diese liefert den Service, das ganz zuletzt zurückgegebene Objekt zu zeigen, damit Du eine Kontrolle darüber hast, womit Du weiterarbeitest.

3.1 Lokale Variablen

Ja, was heißt »mit einem Objekt weiterarbeiten«? Haben wir denn zum Beispiel noch irgendeinen Zugriff auf die 7 der ersten Rechnung. Nein, *so* nicht. Du erinnerst Dich: Auf ein Objekt hast Du nur dann einen Zugriff, wenn es einen *Namen* trägt. Wenn eine so genannte *Variable* auf das Objekt verweist. Die Zahlen 3 und 4 sind aber keine *Namen* für Objekte, sondern die Objekte *selbst*. Sie verschwinden sofort wieder aus Deinem Zugriffsbereich, wenn Du keine Namen an sie bindest.

Wenn wir die 7 der ersten Rechnung retten wollen, müssen wir sie zum Beispiel *marek* nennen. Ändere die erste Zeile in dem Workspace ab. Du benötigst dafür das Zeichen ←, dass Du mit dem Unterstrich _ erhältst. Alternativ kannst Du auch das Zeichen := verwenden[2]. Der Text sieht jetzt so aus:

```
marek ← 3+4.
3-marek
```

Führe wieder ein *printIt* nach Auswahl des gesamten Textes durch. In Zukunft sagen wir das nicht mehr jedesmal dazu. Solange wir den Namen *marek* nicht anders verwenden, zeigt er immer auf das Objekt 3. Merwürdigerweise funktioniert auch die Zeile:

```
marek
```

Sie ist ja gar keine vollständige Nachricht. Aber da *marek* nichts durchführen soll, gibt er sich selbst als Antwort zurück. Übrigens kannst Du auf die Markierung des Textes verzichten, wenn Du mit *printIt* oder *doIt* nur eine einzige Zeile ausführen willst. Du musst nur darauf achten, dass der Cursor in dieser Zeile steht. Funktioniert auch die folgende Zeile?

```
sveta
```

Anscheinend wird ein ganz merkwürdiges Objekt zurückgegeben. Der Name *nil* bedeutet das Objekt »Nichts«. Es ist dasjenige Objekt, auf den ein Name zeigt, wenn er keinem anderen Objekt des Systems zugewiesen wurde. Da stellt sich natürlich die Frage, ob *marek* nun auch in einem anderen Workspace auf das Objekt 7 zeigt.

Hole Dir einen zweiten Workspace. Du wirst feststellen, dass ALT-k anscheinend kein Workspace holt, sondern die Schriftgröße des vorhandenen anpassen will. Das ist richtig, wenn Du Dich *in* dem Workspace bereits befindest. Klickst Du auf den Desktop und führst trotz des World-Menüs ALT-k aus, wird ein neuer Workspace geholt. Natürlich kannst Du ihn auch aus der Lasche ziehen.

Ist *marek* in den zweiten Workspace zugewiesen worden? Nein. Ein Workspace *heißt* eben deshalb Work-*Space*. Er grenzt einen Gültigkeitsbereich aus dem Squeak-System ab. Die Namen gelten nur *lokal* in einem Workspace so lange, bis dieser geschlossen wird.

[2] Squeak scheint es zu bevorzugen. Jedenfalls wird in den Methoden der Pfeil stets in := verwandelt. Das ist allerdings nicht für Smalltalk üblich. Der Pfeil ist *sehr* viel einprägsamer. Wir werden ihn hier auch ausschließlich verwenden.

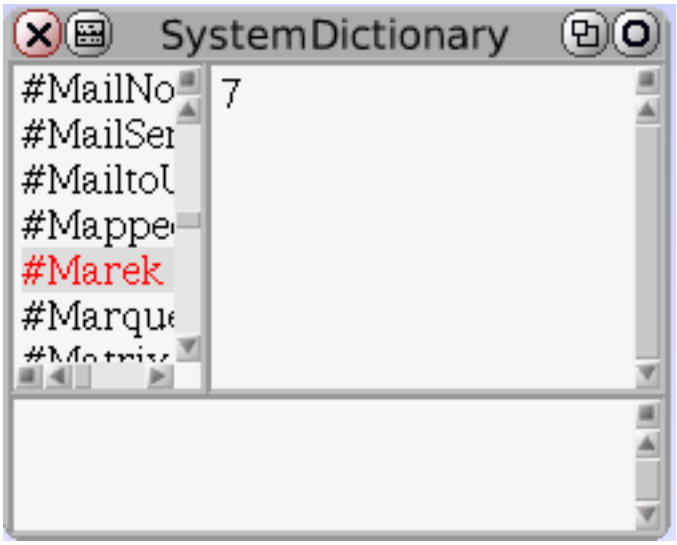

Abbildung 2: Der Inspektor

3.2 Globale Variablen, Inspektor und Explorer

Etwas anders sieht es aus, wenn Du den Namen *Marek* groß schreibst. Führe in einem Workspace

```
Marek←7
```

aus. Du wirst ein Meldefenster von Squeak erhalten: »Unknown variable: Marek«. Darunter findest Du ein paar Vorschläge, wie Du die Sache richtigstellst. Das erste ist »define class«. Offenbar werden groß geschriebene Namen vorwiegend für Klassen verwendet. Klicke jetzt aber nicht auf diesen Eintrag. Sondern wähle denjenigen darunter: »declare global«. Überzeuge Dich nun davon, dass

```
Marek
```

nun in *jedem* Workspace auf das Objekt 7 verweist. Das sieht auf den ersten Blick wie ein Vorteil aus. Aber eine globale Variable ist in dem System gespeichert. Selbst wenn Du alle Workspaces schließt und erst dann einen neuen öffnest, ist *Marek* noch zugewiesen.

Wie wirst Du eine globale Variable wieder los? Tja, so *ganz* einfach ist das an dieser Stelle nicht. Es gibt ein globales Objekt mit dem Namen *Smalltalk*. Wenn Du

```
Smalltalk
```

in einen Workspace eingibst, so ist nach einem *printIt* die Antwort etwas »frech«: *a SystemDictionary (lots of globals)*. Das Objekt ist einfach zu groß, um es auf dem Bildschirm darzustellen. Dennoch wollen wir nun hineinsehen. Dafür gibt es zwei Möglichkeiten. Achte darauf, dass der Cursor wirklich hinter dem Wort *Smalltalk* steht. Gib dann ALT-i ein. Es öffnet sich ein kleines, graues Fenster, in dem Du nur im linken Teil zunächst Einträge siehst. Wenn Du nur lange genug suchst, findest Du *Marek* irgendwann. Was das Doppelkreuz # vor dem Namen soll, erklären wir gleich. Klickst Du auf diesen Namen, so erscheint

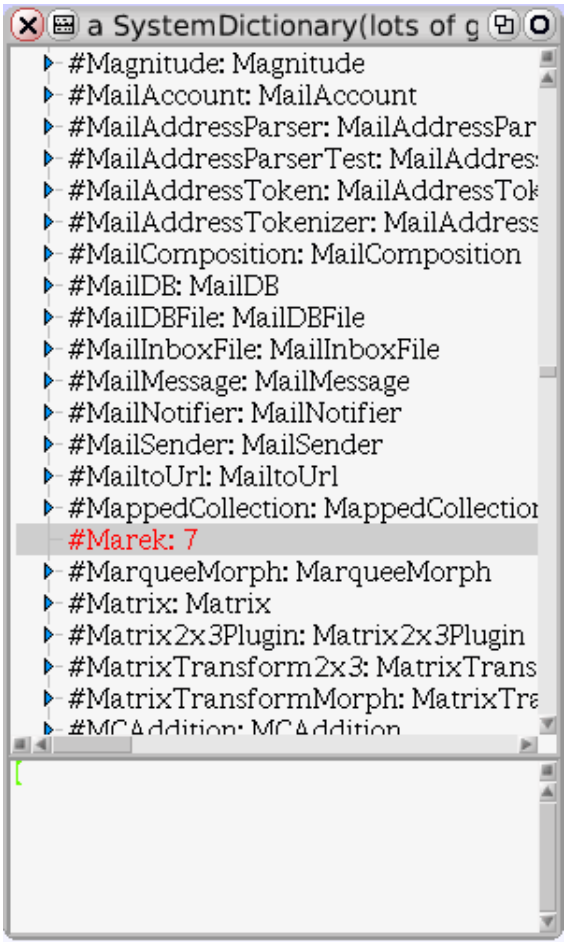

Abbildung 3: Explorer

im rechten Fenster das zugewiesene Objekt. Natürlich kannst Du das Fenster auch vergrößern. Was Du hier kennengelernt hast, ist der so genannte *Inspektor*. MIt einem Inspektor kannst Du in alle Objekte hineinsehen. Einen Inspektor auf *Marek* zu öffnen ist wesentlich weniger interessant. Du findest dort nur die Einträge *self* und *all inst vars*, die bei allen Objekten als Einträge vorhanden sind. Verstehen wir sie? *self* bedeutet sicherlich »selbst«. Ja. Der Name *self* zeigt immer auf das Objekt selbst. Was soll das? Nun: Jeder kann mit sich selbst Gespräche führen. So auch unsere Objekte. *all inst vars* ist vielleicht weniger selbsterklärend und wir wollen diesen Eintrag zunächst auch übergehen. Vielleicht erinnerst Du Dich noch daran, dass es so etwas wie Instanz-Variablen gibt.

Es gibt noch eine andere Möglichkeit, in eine Objekt hineinzuschauen. Statt eines *inspectIt* mit ALT-i kannst Du auch ein *exploreIt* mit ALT-I absetzen. Zunächst einmal sieht das Ergebnis nicht sonderlich ermutigend aus. Wieder erscheint ein Fenster mit dem Eintrag ▷ *root: a SystemDictionary (lots of globals)*. Das kleine Dreieck weist aber auf eine verzweigte Struktur hin. Klicke auf dieses Dreieck und Du siehst den Inhalt des Objekts *Smalltalk* in voller Schönheit. Irgendwo ist unser *Marek* versteckt. Ob nun der Inspektor oder der *Explorer* die bessere Wahl ist, um in ein Objekt hineinzusehen, erscheint Dir sicherlich als eine reine Frage des Geschmacks. Übersieh jedoch nicht die blauen Dreiecke bei den meisten Einträgen. Bei Marek fehlt so ein Eintrag, weil Marek eben »nur« eine globale Variable, aber keine *Klasse* ist. Vielleicht erinnerst Du Dich, dass Klassen Subklassen bilden können. Anders als beim Inspektor liefert der Explorer die Struktur der Abhängigkeiten. Hinter den blauen Dreiecken verbergen sich Unterklassen.

Allein an der Anzahl der blauen Dreiecke erkennst Du, dass die meisten globalen Variablen tatsächlich Klassen sind. So etwas wie unser *Marek* sollte eine absolute Ausnahme bleiben. Angeblich soll es helfen, sich ein Sparschwein auf den Tisch zu stellen und für jede globale Variable 50 oder – in besonders hartnäckigen Fällen – auch 100 Kronen einzuzahlen.

Jetzt wissen wir immer noch nicht, wie wir *Marek* loswerden. Natürlich müssen wir dazu eine Nachricht an *Smalltalk* senden. Die Nachricht heißt:

```
Smalltalk removeKey:#Marek.
```

Danach sollte *Marek* in keinem Workspace mehr auf ein Objekt zeigen. Noch eine kurze Bemerkung zum Selektor. *Smalltalk* ist ein so genanntes Dictionary. Die Klasse *Dictionary* erzeugt Instanzen, die Einträge in einer Kette wie Perlen an einer Schnur anordnen. Die einzelnen Einträge sind wiederum Paare aus zwei Objekten. Das erste, ist der sogenannte »Schlüssel« oder »Key«. Das zweite ist das so genannte *Wertobjekt* oder *Value*. Der Schlüssel ist dem Wertobjekt zugeordnet, wie auch in einem Lexikon unter einem Schlüsselwort dessen Bedeutung nachgeschlagen werden kann. Wird der Schlüssel entfernt, so kann dessen Bedeutung natürlich nicht »in der Luft hängen«. Also wird das ganze Paar gelöscht. Und genau das haben wir eben getan.

3.3 Symbole, Strings, Character

Vielleicht hast Du Dich eben gefragt, wieso der Schlüssel *#Marek* in dem Dictionary ein *Objekt* sein soll. Sind Namen denn Objekte? Nein, das sind sie nicht. Sie *zeigen* auf Objekte. Mehr nicht. Im Gegensatz zu Objekten müssen Namen auch nicht *erzeugt* werden. Sie »existieren« grundsätzlich wie wir uns auch im täglichen Leben beliebig sinnvolle oder weniger sinnvolle Namen ausdenken können.

#Marek ist aber auch kein Name, sondern eine Zeichenkette. Und zwar eine *eindeutige* Zeichenkette. Vielleicht kennst Du die Bezeichnung *Strings* von anderen Programmiersprachen. Aber *#Marek* ist *kein* String, sondern ein *Symbol*. Symbole sind eindeutig, Strings dagegen nicht. Bei Strings verwendest Du nicht das Pfundzeichen #, sondern Du schließt die Zeichenkette in Hochkommata ein. Zum Beispiel so: *'Marek'*.

Was bedeutet nun *Eindeutigkeit*? Bereite die folgenden Zuweisungen in einem Workspace vor:

```
marek←'Marek'.
sveta←'Marek'.
```

Damit zeigen beide Namen auf ein String-Objekt 'Marek'. Ja, wie ist es nun? Zeigen beide Namen auf *ein und dasselbe* Objekt oder sind es zwei *verschiedene* Objekte? Führe dazu die folgende Nachricht aus:

```
marek=sveta.
```

Das Ergebnis ist (hoffentlich) ein so genanntes *boolesches Objekt* »true«. Genauer gesagt: *true* ist der *reservierte* Name für dasjenige Objekt, das den Wahrheitswert »Wahr« repräsentieren soll. Du kannst daher den Namen *true* nicht frei verwenden.

Führe nun die Nachricht

```
marek==sveta.
```

aus. Ist das Objekt *false* eine Überraschung? Nun, um die Bedeutung der Selektoren *=sveta* und *==sveta* für *marek* herauszufinden, führe dieselben beiden Nachrichten *marek=sveta* und *marek==sveta* noch einmal aus, wobei Du zuvor die Zuweisung

```
marek←#Marek.
sveta←#Marek.
```

Vorgenommen hast. Jetzt sollte in *beiden* Fällen das Rückgabeobjekt *true* sein. Der Selektor *=sveta* testet also auf Gleichheit der Werte. Er fragt sozusagen, ob *marek* und *sveta* Zwillinge sind. Der Selektor *==sveta* testet aber auf Gleichheit der Objekte. Er fragt sozusagen, ob *marek* und *sveta* In Wirklichkeit dieselbe »Person« bedeuten.

Und jetzt siehst Du den Unterschied zwischen Symbolen und Strings. Symbole sind Zeichenketten, von denen es im ganzen System *keinen* Zwillingspartner geben kann. Von Strings dagegen können beliebig viele Kopien existieren. Symbole werden also immer dann verwendet, wenn die Eindeutigkeit eines Namens sichergestellt werden soll. Sie sind sozusagen Kontrollobjekte »im Hintergrund«.

Ansonsten unterscheiden sich Strings und Symbole kaum. Sowohl

```
#marek size.
```

als auch

```
'marek' size.
```

ergeben die Länge der Zeichenkette als Rückgabeobjekt.

Was führen

```
#marek at: 3.
```

und

```
'marek' at:3.
```

durch?

Nun, dabei erhältst Du mit *$r* einen weiteren Objekttyp. Dieses ist ein einzelner Buchstabe, ein so genannter *Character*. Du fragst Dich möglicherweise, warum nicht einfach *#r* oder 'r' ausgegeben werden, was doch naheliegt. Nun, einzelne Zeichen verstehen Messages, die Symbole und Strings nicht verstehen. So kannst Du zum Beispiel mit

```
$r asInteger.
```

den so genannten ASCII-Wert des Buchstabens $r ermitteln, was 114 ergibt. Natürlich versteht die Zahl 114 auch das Umgekehrte:

```
114 asCharacter.
```

gibt als Rückgabeobjekt $r. Jede Zahl ist über den so genannten ASCII-Code einem darstellbaren Zeichen zugeordnet. Das macht zum Beispiel die Erstellung von Geheimschriften sehr einfach, da das Problem lediglich auf ein Rechnen mit Zahlen verlagert werden kann. Das werden wir noch sehen.

Versuche nun, einmal den Selektor *asInteger* auf das Symbol *#r* und danach auf den String 'r' anzuwenden. Immerhin *verstehen* die beiden Objekte die Nachricht. Aber in beiden Fällen wird das Objekt *nil* zurückgegeben. Mit anderen Worten: Der Nachricht liefert eine gleichgültige Reaktion der Objekte.

Kommen wir noch einmal auf *'marek' at: 3* zurück. Was ist zu tun, wenn die Ausgabe wirklich als String 'r' erfolgen soll? Nun, dann muss dem zurückgegebenen Character die Nachricht *asString* noch zusätzlich mitgegeben werden. Wie gehen wir vor? Nun, führen wir zuerst

```
buchstabe←'marek' at:3.
```

aus. Der Name *buchstabe* zeigt dann auf $r. Danach führe die Zeile

```
buchstabe asString.
```

aus. Das Ergebnis sollte wie gewünscht erscheinen. Aber sind zwei Messages nicht ein wenig umständlich. Können wir uns nicht auf eine Zeile beschränken? Etwa:

```
'marek' at:3 asString.
```

Versuche es. Ups, das erste Fehlerfenster. Schaue zunächst nur auf den Rahmen. Dort steht: »Only integers should be used as indices.« Ist das nicht eine Art Orakelspruch? Versuche grundsätzlich Fehlermeldungen immer zu verstehen. Auch wenn es Dir nicht leicht fällt. Nur so lernst Du.

Es ist tatsächlich die Frage, welchem Objekt wir die Anweisung *asString* geschickt haben. War es nun dem entstehenden Character $r oder der Zahl 3. Es ist Letzteres der Fall. Tatsächlich war die Zahl 3 das erste angesprochene Objekt. Auf die Nachricht *3 asString.* hat sie daraufhin das Objekt *'3'* zurückgegeben. Danach folgte die Nachricht *'marek' at: '3'.* und das ist natürlich Unsinn. Denn als »Index« wird hier ein String verwendet und keine ganze Zahl (Integer). Der Selektor *at:* wurde von 'marek' zwar verstanden, aber das übergebene Objekt passte nicht.

Wie ist das Ganze zu erklären? Nun, als Anfänger bist Du gut beraten, wenn Du grundsätzlich Dich pro Zeile auf *eine* Nachricht beschränkst. Spare nicht mit der Verwendung von Namen (!). Es gibt unendlich viele, die Dir zur Verfügung stehen. Als Fortgeschrittenerer (bemerke den Komparativ!) solltest Du Dir die Regel merken, dass ein Selektor *ohne* Argument (*unärer Selektor*) immer Vorrang von einem Selektor *mit* Argumenten besitzt (*Keyword-Selektor*). Dazwischen stehen noch die so genannten *binären Selektoren*. Dies sind Nachrichten, die aus einem Sonderzeichen und *einem* Argument bestehen, wie zum Beispiel *+3*, *=sveta*, *<4* und so weiter. Grundsätzlich haben *unäre* Selektoren Vorrang vor *binären* und diese wieder Vorrang von *Keyword*-Messages. In der absolut verrückten Nachricht:

```
'marek' at:'3'asInteger-1.
```

wird also zuerst dem String '3' die Nachricht *asInteger* zugesandt. Danach wird das Objekt *3* zurückgegeben. Dieses erhält die Nachricht *-1*, so dass das mit dem Selektor *at:* an *'marek'* übergebene Objekt die Zahl *2* ist. Soll die »Vorfahrtsregel« geändert werden, müssen Klammern gesetzt werden. Bei unserem fehlgeschlagenen Versuch müssen wir die Nachricht lediglich in

```
('marek' at:3) asString.
```

verbessern.

Es kann nicht ausdrücklich genug darauf hingewiesen werden, dass die Lesbarkeit des Codes das Wichtigste überhaupt ist! Für kryptischen, zusammengedrängten Code sind Fans aus der C- und Perl-Welt besonders anfällig. Dabei geht es aber auch um andere Ziele, wie zum Beispiel hardwarenahes, zeitunkritisches Verhalten. Wer eine Programmiersprache erlernt, tut sich selbst und anderen damit keinen Gefallen, an der falschen Stelle zu sparen. Daher zu Beginn: Möglichst wenig Nachrichten pro Zeile und nicht mit Namen sparen.

3.4 Arrays

Arrays sind – wie Strings und Symbols – eine Sonderform einer *Collection*. Während Symbole und Strings nur Anordnungen von Character-Objekten sind, wobei Symbole auch keine Leerzeichen zulassen, fassen Arrays *beliebige* Objekte zu einer Einheit zusammen. Auch hier wird – sehr seltsam – das Pfundzeichen verwendet:

```
einArray←#(3 'text' bedrich).
```

Denke unbedingt an die Leerschritte zwischen den einzelnen Komponenten. Sehr merkwürdig erscheint Dir sicherlich *bedrich* innerhalb des Arrays. Was ist denn das? Ein Name, der noch nicht zugewiesen wurde? Nein, es handelt sich um ein *Symbol*. Innerhalb der Array-Klammern darf das Pfundzeichen nicht auftreten. Dies gilt auch für eingebettete Arrays selbst:

```
einArray←#(3 'text' bedrich (4 'Text' gabriela))
```

Es juckt einem in den Fingern, die Einträge durch ein Komma abzugrenzen. Ein Komma ist aber ein Zeichen, das natürlich einen Selektor einer Nachricht darstellt. Ein Komma kann zum Beispiel dazu verwendet werden, zwei Strings mit einander zu verbinden (*Konkatenation*):

```
'dobry',' den'.
```

Dies gilt auch für andere Kollektionen. Also auch für *Symbole* und also auch für *Arrays*. So ergibt

```
#(1 2),#(3 4).
```

das Rückgabeobjekt *#(1 2 3 4)* . Integers verstehen aber den Selektor *,* nicht. Daher würde *3,'text'* in einer Antwort *Message not understood* enden. Aus diesem Grunde müssen zwischen die Objekte eines Arrays Leerzeichen gesetzt werden.

Symbols, Strings und Arrays haben eine feste Größe. In einem Array können aber Komponenten ersetzt werden. So kannst Du zum Beispiel durch

```
einArray ←#(1 2).
einArray at:2 put:'vier'.
einArray
```

das Array in das Objekt *#(1 'vier')* überführen (*einArray* muss danach noch einmal abgefragt werden, um etwas zu sehen. Daher die letzte Zeile).

Auf einzelne Komponenten greifst Du natürlich über den Selektor *at:* zu. Was ergibt

```
einArray←#(3 'text' bedrich (4 'Text' gabriela)).
einArray at:4.
```

Nun, es wird das Array *#(4 'Text' gabriela)* zurückgegeben. Wie können wir nun *'Text'* »holen«? Die Lösung ist doch wohl:

```
einArray at:4 at:2.
```

Nicht wahr? Leider nein. Hier wird dem Array der Selektor *at:at:* gesendet, der von einem Array nicht verstanden wird. Gemeint war ja etwas Anderes. Dem zurückgegebenen Objekt *#(4 'Text' gabriela)* soll beim zweitenmal der Selektor *at:2* geschickt werden. Daher muss die Zeile wieder geklammert werden, und zwar so:

```
(einArray at:4)at:2.
```

Ohne Klammern würde die Nachricht als *eine einzige* Nachricht für *einArray* aufgefaßt werden.

3.5 Zahlen

Nun noch ein paar Anmerkungen zu Zahlen. Es gibt unterschiedliche Arten. Mit Hilfe der *class* Nachricht können wir herausfinden, welche Arten es gibt. Führe die folgenden Zeilen jeweils *einzeln* aus:

```
3 class.
3.0 class.
(3/4) class.
(3.0/4) class.
(3.0/4.0) class.
3.0 class.
3e5 class.
3e9 class.
3e-5 class.
3.0e-5 class.
```

Es gibt also verschiedene Zahlen vom Typ Integer (ganze Zahlen), Floats (Gleitkommazahlen) und Fraction (Brüche aus ganzen Zahlen). Bei der dritten, vierten und fünften Zeile siehst Du wieder die Notwendigkeit der Klammer. Ansonsten wird zuerst dem Nenner die Nachricht *class* geschickt und erst dann dem Zähler der Selektor / mit einem falschen Argument.

Alle Zahlen reagieren auf die Selektoren +, -, und *. Die Bruchzahlen (Fraction) sind dabei besonders interessant:

```
(1/2) +(2/3).
```

ergibt wie zu erwarten das Fraction-Objekt (7/6). Sind die Klammern notwendig? Probiere es aus:

```
1/2 +2/3.
```

ergibt seltsamerweise (5/6). Wie ist das möglich? Nun, sowohl / als auch + sind *binäre* Nachrichtenselektoren. Also haben sie denselben »Rang«. Die Nachricht wird damit von links nach rechts abgearbeitet. Zuerst wird der Zahl 1 die Nachricht /2 gesendet. Das ergibt das Objekt (1/2). Dann erhält dieses Objekt die Nachricht +2. Das ergibt natürlich (5/2). Diesem Objekt wird zum Schluss die Nachricht /3 gesendet. Und das ergibt natürlich (5/6). Smalltalk weiß nicht, dass 2/3 *vor* der Addition ausgeführt werden sollte, da alle Selektoren gleichberechtigt sind.

Mit anderen Worten: Bei Smalltalk gilt (sehr konsequent!) keine Punkt-vor-Strich-Rechnung. Es müssen mathematisch unnötige Klammern gesetzt werden. Manche Anhänger anderer »objektorientierter« Sprachen halten das für ein Problem. Tatsächlich ist es ganz konsequent und den eigentlichen Vorgängen in einem Computer viel näher. Dass ein Computer Punkt-vor-Strich beherrscht, ist ein nicht nötiger Service. Es sein denn, es wird die Sprache vorwiegend zum Rechnen benutzt, wie es zum Beispiel bei FORTRAN der Fall ist.

Obowohl Smalltalk primär keine Rechensprache ist, bietet sie doch viel mehr Konstrukte, von denen wir in anderen Sprachen nur träumen können. Dazu gehört zum Beispiel die Darstellung einer Zahl in einem beliebigen Zahlensystem. Die Addition einer Zahl im Vierersystem und einer solchen im Fünfersystem ist überhaupt kein Problem. Die Basis muss lediglich durch ein Präfix der Form *<Basiszahl> r* kenntlich gemacht werden:

```
5r421+4r30.
```

ergibt die Zahl *123* im Dezimalsystem. Soll diese in einem anderen Zahlensystem, zum Beispiel 16 dargestellt werden, ist die Nachricht

```
123 radix:16.
```

erforderlich, was das String-Objekt *'7B'* ergibt.

Beliebige Potenzen und damit auch Wurzeln werden mit dem Selektor *raisedTo:* erzeugt. Für Quadratwurzeln gibt es den unären Selektor *sqrt*. Und vieles mehr.

Wir sollten noch auf die Division eingehen. Grundsätzlich ist das Rechnen mit Integer dem Rechnen mit Floats aus Gründen der Genauigkeit vorzuziehen! Bei Floats tauchen grundsätzlich Rundungsfehler auf. Bei der Division ist aber Vorsicht geboten.

/ erzeugt aus *zwei* Integer *grundsätzlich* ein Objekt vom Typ *Fraction*. Es handelt sich hierbei *nicht* um die Integer-Division. Ist eine der Zahlen ein Float, ist dies die gewöhnliche Division. Beispiele: *2/3* ergibt *(2/3)* aber *2.0/3* ergibt *0.66666...*.

// ist in *allen* Fällen die gewöhnliche Integerdivision. Das heißt: Es wird gezählt, wie oft der Nenner in den Zähler hineinpasst und das Ergebnis als Integer ausgegeben (auch bei Floats). Beispiele: Sowohl *3//2* als auch *3.4//2.1* ergeben *1*.

\\ ist in *allen* Fällen die Modulodivision. Es wird wiederum gezählt, wie oft der Nenner in den Zähler hineinpasst. Es wird der *Rest* als Objekt zurückgegeben. War der Zähler ein Float, so ist auch das zurückgegebene Objekt ein Float. Beispiele: *3\\2* ergibt *1*. *3.4\\2.1* ergibt *1.3*.

Alle Zahlen verstehen den Selektor *rounded* für das Runden. Du fragst nach Zufallszahlen? Nun, das kommt jetzt.

3.6 Objekte ohne Literale

Wir haben bis jetzt nur Objekte kennengelernt, die sich *direkt* erzeugen lassen. Anhand einer bestimmten Form erkennt die Virtual Machine (VM), mit welcher Klasse sie »reden« soll, um ein ganz bestimmtes Objekt zu erzeugen. Diese bestimmten Formen heißen *Literale*. Die VM erkennt zum Beispiel mit Hilfe des Literals *'marek'*, dass der Klasse *String* eine Nachricht zugesendet werden soll, die sie veranlaßt, ein String-Objekt zu erzeugen und dieses mit dem Wert 'marek' zu belegen. Am Literal *$a* erkennt die VM, dass sie jezt nicht mit der Klasse String, sondern mit der Klasse Character »reden« soll.

Außer Zahlen, Character, Symbols, Strings und Arrays gibt es keine weiteren Objekte, die durch Literale erzeugt werden[3]. In allen anderen Fällen muss mit der zugehörigen Klasse *explizit* geredet werden.

Das erste Beispiel dieser Art ist die Klasse *Random* zur Erzeugung von Zufallszahlen. Die meisten Klassen verstehen den Nachrichtenselektor *new*. Wenn die Klasse *Random* die Nachricht *Random new.* erhält, so erzeugt sie ein Objekt (Instanz) dieser Klasse. Denke daran, dass Du jedem neuen Objekt einen Namen geben mußt. Sonst ist es für Dich verloren[4]. Die Nachricht *Random new.* allein reicht also nicht.

```
zufall←Random new.
```

Jetzt verfügst Du über einen Zufallsgenerator mit dem Namen *zufall*. Er versteht alle Nachrichten, die Zufallsgeneratoren eben verstehen. So erzeugst Du Dir eine Zufallszahl zwischen 0 und 1 mit der Nachricht

```
zufall next.
```

Soll die Zufallszahl eine ganze Zahl, zum Beispiel zwischen 1 und 20 sein, so verwendest Du die Nachricht:

[3] Die Bezeichnungen *nil, true* und *false* sind keine Literale. Es sind vielmehr Namen für bereits existierende Objekte, die nicht geändert werden können. Daher heißen diese Variablen auch *Pseudovariablen*.

[4] Es ist noch nicht wirklich »weg«. Aber da der Zugriff fehlt, belegt es als so genannte »Speicherleiche« einen Teil des Arbeitsspeichers. Daher beseitigt Smalltalks automatische Müllabfuhr (*garbage collection*) die Speicherleichen, sobald der Platz benötigt wird.

```
zufall nextInt:20.
```

Ein zweites Beispiel sind die uns bekannten Arrays. Sie lassen sich zwar über ein Literal erzeugen. Aber es könnte zum Beispiel sein, dass Du zu irgendeinem Zeitpunkt ein leeres Array brauchst, dass erst später aufgefüllt werden soll. Die VM erzeugt mit dem Literal *#()* zwar wirklich ein leeres Array. Aber dieses hat zu allen Zeiten die Länge 0. Es ist hier nicht möglich, etwas einzufüllen. Wenn Du zum Beispiel ein leeres Array benötigst, dass aber drei Einträge speichern kann (Arrays haben immer eine feste Länge!), musst Du der Klasse *Array* die Nachricht *Array new:3* senden. Danach kann dann das Array mit dem Selektor *at:put:* gefüllt werden.

```
sveta←Array new:3.
sveta at:1 put:'erstens'.
sveta at:2 put:'zweitens'.
```

Das Array ist noch nicht ganz gefüllt. Schaust Du Dir seinen Inhalt an, so erkennst Du, dass die dritte Komponente *nil* enthält. Das Array *#()* dagegen würde *gar keinen* Eintrag enthalten, und damit auch keinen Eintrag ermöglichen. Natürlich gibt es Kollektionen, die nicht so eingeschränkt wie ein Array sind. Diese werden im übernächsten Kapitel 6 Smalltalk II (Smalltalk-Erweiterungen) sehr genau besprochen[5] .

Nun wird es ein wenig Zeit für Grafik. Um eine Turtle zu erhalten, musst Du mit der Klasse *Pen* reden:

```
turtle ← Pen new.
```

Die Turtle ist allerdings unsichtbar. Sie befindet sich in der oberen linken Ecke. Lassen wir sie zu einem Punkt (200 | 100) laufen. Einen Punkt erstellst Du, indem Du an die x-Komponente 200 die Nachricht @100 schickst.

```
punkt←200@100.
```

Jetzt sollte

```
turtle goto:punkt.
```

einen schwarzen, dünnen Strich aufblitzen lassen. Dort, wo sich Dein Workspace befindet, ist der Strich unterbrochen. Überhaupt »radierst« Du durch Bewegen des Fensters Teile des Striches wieder aus. Der Grund: Jedes Fenster ist ein Morph. Aber eine Turtle-Zeichnung ist *kein* Morph. Sie kann allerdings *auf* einem Morph ausgeführt werden und ist dann gegen »Radierprozesse« geschützt.

Es gibt nun eine Klasse *TurtleCanvas*, die unsere Wünsche erfüllen wird. Diese Klasse ist nun das erste Beispiel für eine abweichende Erzeugung von Instanzen. Hier genügt nicht nur die Angabe von *new*, sondern es gibt *newFromUser:aColor*, *new:aColor from:aPoint to:anotherPoint* und *new:aColor width:anInteger height:anInteger*. Die Namen für die Attribute hinter den Schlüsselwörtern zeigen Dir, was dort eingegeben werden muss. Wir

[5] Übrigens ist die Nachricht *new* nicht dasselbe wie *new:*. Die Klasse *Array* verwendet für die Instanzvariablen *indizierte Variablen*. Daher muss bei dem Slektor *new:* ein Integer-Objekt übergeben werden, das die Anzahl der Variablen festlegt.

verwenden der ersten Selektor *newFromUser:aColor*, weil dieser mit Abstand am interessantesten ist. Vorher müssen wir ein wenig über die Klasse *Color* erfahren. Zunächst müssen wir uns ein Farbobjekt erzeugen. Dies geschieht ebenfalls nicht mit *new*, sondern der Selektor entspricht dem englischen Namen der gewünschten Farbe:

```
gruen←Color green.
schwarz ←Color black.
```

Unser Turtle-Canvas soll schwarz sein. Also senden wir gleich die Nachricht *TurtleCanvas newFromUser: schwarz.* Vorher musst Du wissen, dass sich Dein Mauscursor verändern wird. Ziehe Dir mit festgehaltener roter Maustaste einen Bereich auf dem Desktop auf. Dieser wird Dein *TurtleCanvas* werden.

```
sveta←TurtleCanvas newFromUser:schwarz.
```

Falls Du diese Nachricht mit *printIt* übergeben hast, wirst Du als Rückmeldung nur mitgeteilt bekommen, dass das Objekt existiert. Aber wir sehen es noch nicht.

```
sveta openInWorld.
```

Jeder Morph muss mit dieser Nachricht sichtbar gemacht werden. Jetzt sollte Dein Rechteck oben links auf dem Desktop erscheinen. Du kannst den Morph beliebig verschieben.

Nun brauchen wir noch die Turtle.

```
marek ←Pen newOnForm:sveta.
marek color: gruen.
marek up.
marek goto:0@0.
marek down.
marek goto:200@100.
```

Na, wo ist er denn, der *marek*? Klicke den Morph einmal an. Jetzt solltest Du einen grünen Strich sehen können, wenn Dein TurtleCanvas nicht zu klein war.

So geht es im Prinzip. An dieser Stelle soll das erst einmal genügen. Du wirst den Morph natürlich wie jeden Morph über den Halo wieder los.

4 System Browser

Jetzt kommen wir zum interessantesten Teil dieser Reise. Wie erzeugen wir nun unsere eigenen Klassen, unsere eigenen Objekte? Denken wir wieder an unsere Familie.

```
sveta gib:marek bitte:butter und:tee.
```

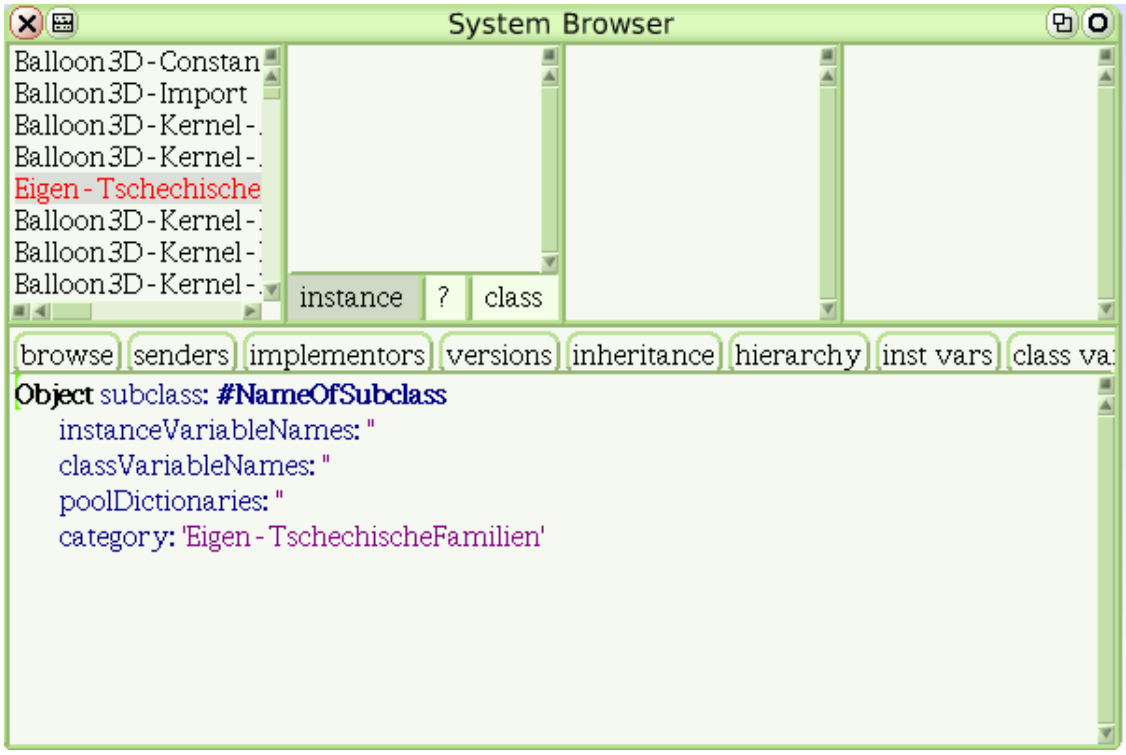

Abbildung 4: Klassenkategorien

wird natürlich nicht funktionieren. Wir haben den Namen *sveta* nun wahrhaft häufig strapaziert. Aber kein Objekt, das bisher *sveta* hieß, versteht diese Nachricht. Machen wir uns auf den Weg, das zu ändern. Zunächst einmal müssen *marek* und *sveta* zu einer *Klasse* gehören. Wir haben schon früher gesagt, dass Klassen so etwas wie (biologische) Familien sind. Nehmen wir das jetzt einmal wörtlich und betrachten die Familie *Svoboda*. So soll unsere Klasse heißen. Sicherlich sollten wir uns auch überlegen, welche *Eigenschaften* die Familienmitglieder auszeichnen. Wir wollen uns dabei nur auf die Attribute *alter* und *vorname* beschränken. Los geht es.

Wir benötigen ein neues Werkzeug: den so genannten *System-Browser*. Der System-Browser gehört – wie auch der Workspace – zu *jeder* Grundausstattung eines Smalltalk-Systems. Du findest ihn in der Lasche unter der nicht gerade vielsagenden Bezeichnung *Browser*. Es gibt in Squeak eine Unmenge von Browsern. Der System-Browser ist tatsächlich *der König* unter den Browsern. Alternativ erhältst Du ihn auch mit ALT-b, wenn Du Dich in keinem anderen Fenster befindest.

Auf den ersten Blick sieht das Ding unglaublich kompliziert aus. Aber keine Angst. Wir kümmern uns zunächst nur um die vier kleinen Fenster. Sehr wahrscheinlich siehst Du nur im linken Fenster überhaupt irgendwelche Einträge. Das sind die so genannten Klassen-Kategorien. Klickst Du auf irgendeine dieser Kategorien füllt sich das zweite Fenster mit den zugehörigen Klassen. Ferner findest Du im unteren Hauptfenster eine Textvorlage, die gleich wichtig werden wird. Falls Du Dich erinnerst, dass wir im ersten Kapitel schon einmal darüber sprachen, bist Du sehr, sehr gut präpariert. Das ist aber nicht unbedingt notwendig.

4.1 Einrichten einer Kategorie

Führe jetzt im ersten den vier kleinen Fenster links einen gelben Klick (@g) aus. Es wird ein Kontextmenü sichtbar. Wähle den Eintrag *add item...* aus. Überschreibe in dem kleinen

Fenster den Eintrag *Category-Name* mit *Eigen-TschechischeFamilien*. Im Fenster springt der Balken auf den neuen Eintrag und färbt den Namen der Kategorie rot. Im unteren Fenster erscheint jetzt die Textvorlage. Das Bild sollte so aussehen wie in der Abbildung 4 auf der vorherigen Seite.

4.2 Einrichten einer Klasse

Jetzt legen wir unsere Familien-Klasse *Svoboda* an. Dies geschieht *nicht* über das Kontextmenü im zweiten Fenster. Sondern im Hauptfenster. Was Du dort siehst ist natürlich wieder eine Nachricht. An wen? Nun, der Receiver steht immer am Anfang. Es ist eine Klasse mit dem Namen *Object*. An diese Klasse wird eine Nachricht mit jenem komplizierten Selektor

subclass:instanceVariableNames:classVariableNames:category:

gerichtet, an den Du Dich vielleicht noch erinnerst. Hinter *subclass* tragen wir nun das Symbol (!) *#Svoboda* ein. Du erinnerst Dich? Symbole sind *eindeutig*. Es wird also in dem ganzen Squeak-System keine andere Familie mit dem Namen *Svoboda* geben. Hinter dem zweiten Schlüsselwort *instanceVariableNames:* gibst Du einen String an. Kein Symbol. Hier hinein gehören jetzt die beiden Namen *alter* und *vorname* und zwar durch einen Leerschritt (*kein* Komma!) getrennt. Hinter dem Schlüsselwort *instanceVariableNames:* sieht es also wie folgt aus:

```
instanceVariableNames:'alter vorname'
```

Das war es schon. Ja, den Rest lassen wir unverändert. Entweder benötigen wir ihn nicht (*classVariableNames:* und *poolDictionaries:*) oder er ist schon eingetragen (*category:*) . Lösche aber bei *classVariableNames:* und *poolDictionaries:* nicht etwa die leeren Strings. Die sind notwendig, da hinter jedem Schlüsselwort *ein Objekt* übergeben werden muss. Und nicht etwa überhaupt nichts.

So, nun müssen wir diese Nachricht wieder der VM übergeben. Aber diesmal weder mit *doIt*, noch mit *printIt*, sondern mit einem *acceptIt*. Wähle zunächst den *ganzen* Text mit ALT-a aus. Und dann wird das *acceptIt* mit ALT-s oder über das Kontextmenü (@g!) mit dem Eintrag OK ausgelöst. Wenn alles funktioniert hat, sollte das Ganze so wie in der Abbildung 5 auf der nächsten Seite aussehen. Im dritten Fenster erscheinen nun zwei Einträge, über die wir gleich sprechen. Das Hauptfenster hat sich unterteilt. In der unteren Hälfte kannst Du einen Kommentar über die Klasse loswerden. Hier kann ein beliebiger Text stehen. Schreiben wir also hinein: *Ich bin unsere kleine Beispielfamilie, die ein Gespräch bei einem Frühstück an einem schönen Maimorgen simulieren soll.*

Während sonst in der Programmiersprache keine Umlaute erlaubt sind, sieht es bei Kommentaren anders aus. Diese sind nur für den Benutzer gedacht. Mache von aussagekräftigen Kommentaren intensiv Gebrauch. Vermeide unbedingt nichtssagende Kommentare, wie *Dies ist die Familie Svoboda*. Was sowieso klar ist, sollte nicht unnötig erklärt werden. Bedenke aber auch: Falsche Kommentare sind schlimmer als gar keine. Vielleicht sieht es ein wenig albern aus, wenn sich die Klasse mit »Ich bin ... « selbst vorstellt. Aber das entspricht genau den Gepflogenheiten. Die Klasse ist selbst ein »lebendes Ding«. Auch den Kommentar musst Du auswählen und mit ALT-s akzeptieren.

Unter dem zweiten Fenster siehst Du den Knopf mit dem Fragezeichen. Dieser blendet alles aus, was nicht zur Erklärung der Klasse beiträgt. Der Knopf mit der Aufschrift *class*

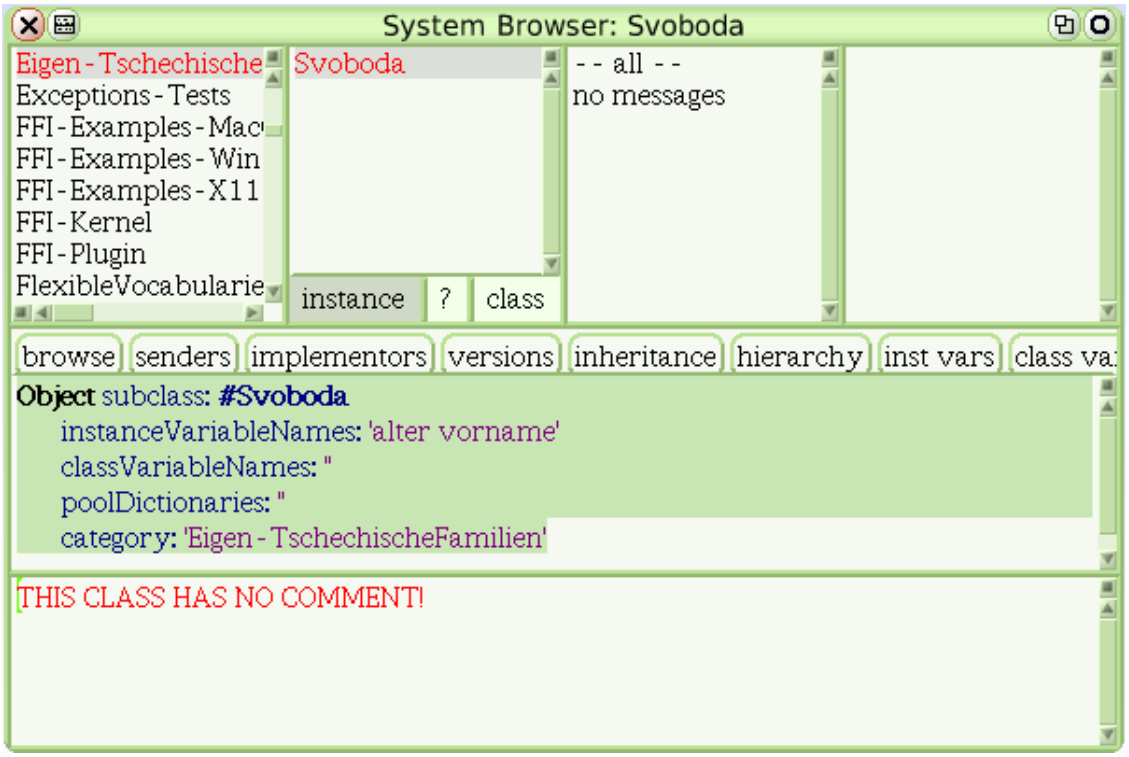

Abbildung 5: Klassen im System-Browser

informiert über Eigenschaften (*classVariableNames*) und Methoden, die nur der Klasse gehören. Es scheint keine Klassenmethoden zu geben. Aber das ist falsch. Denn *Svoboda* ist wenigstens in der Lage auf die Nachricht *Svoboda new.* zu reagieren. Probieren wir es im Workspace aus:

```
sveta←Svoboda new.
```

Das funktioniert. Aber trotzdem versteht *sveta* die Nachricht

```
sveta gib:marek bitte:butter und:tee.
```

noch nicht. Das wollen wir gleich ändern. Zuvor ist aber noch die Frage, warum *Svoboda* auf den Selektor *new* reagieren kann. Tatsächlich erbt die Klasse *Svoboda* die entsprechende Methode von der Superklasse *Object*. Dass *Svoboda* eine Unterklasse von *Object* ist, haben wir ja mit der Einrichtungsmessage

```
Object subclass:#Svoboda
```

so vorgesehen. *Jede* Klasse ist Unterklasse von Object. Auch wenn sie Unterklasse einer anderen Klasse ist, so ist doch wenigstens eine in diesem »Stammbaum« ein direkter Nachfahre von *Object*. Nur *Object* ist kein Nachfahre von irgendeiner Klasse. Bevor wir die sehr wichtige Frage beantworten, wie wir über eine Klasse Informationen erhalten, vervollständigen wir unsere Familie *Svoboda* erst einmal. Gehe wieder zurück in den Browser und auchte darauf, dass unter dem zweiten Fenster der Knopf *instances* aktiviert ist (grau).

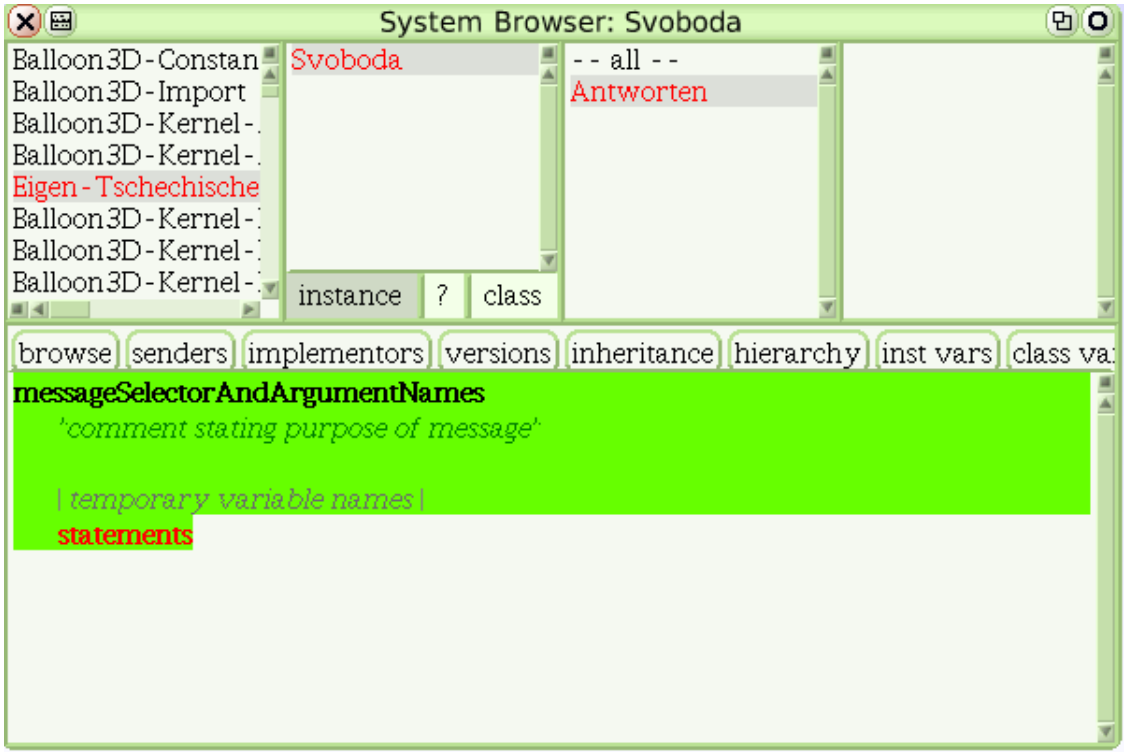

Abbildung 6: Methodenkategorien

4.3 Erzeugen einer Methodenkategorie

Das dritte Fenster ist dem ersten vergleichbar. Zwei Einträge sind schon vorhanden, die aber keine Methodenkategorien sind. Der Eintrag *no messages* sagt zum Beispiel, dass die Klasse auf keine Nachrichten reagieren kann, die sie nicht schon geerbt hat. Hier erzeugen wir einen Eintrag wieder über das Kontextmenü und dem Eintrag *new category...*. Es öffnet sich abermals ein Fenster mit der Bezeichnung »Add category«. Das Menü schlägt Dir im unteren Teil einige Bezeichnungen vor, die häufiger benötigt werden. Wir wählen *new...*. Wir nennen unsere Methodenkategorie einfach *Antworten*. Hast Du alles richtig gemacht, sollte das Ganze so aussehen wie in der Abbildung 6 .

Die Methodenkategorien sind nicht *unbedingt* nötig wie die Klassenkategorien. Sie dienen nur der besseren Übersicht. Natürlich kann eine eingebaute Methode auch *nachträglich* einer Kategorie zugeordnet werden.

4.4 Erzeugen einer Methode

Wir wollen nun sicherstellen, dass *sveta* auf die schon mehrmals beschriebene Nachricht ganz brav antwortet. Wenn *sveta* nun wirklich die Objekte zurückgeben soll, so müssen erst einmal Butter und Tee als Instanzen der entsprechenden Klassen vorhanden sein. Das haben wir noch nicht simuliert. Wir wollen zunächst eine sehr viel einfachere Aufgabe lösen: Sveta soll lediglich den Satz »Hier, bitte, Marek. Da sind Butter und Tee.« antworten. Wenn sie das nur *sagen* soll, so müssen Butter und Tee genaugenommen nicht existieren. Es reicht also, dass Sveta weiß wie die Dinge zu benennen sind. Statt dass in der Nachricht

```
sveta gib:marek bitte:butter und:tee.
```

existierende Objekten als Attribute zur Sprache kommen, reichen einfache Strings.

```
sveta gib:'marek' bitte:'butter' und:'tee'.
```

Vielleicht ist es überflüssig zu betonen, dass es nicht um die Eleganz der Programmierung oder gar den Sinn dieses Beispiels geht. Das Ziel ist es, etwas verständlich zu machen. Stelle Dir vielleicht vor, die Nachricht wird *gesagt* und *sveta* soll die Antwort sozusagen aufschreiben. Damit sie überhaupt etwas tut, zwingen wir sie, die Eigennamen in Großbuchstaben zu verwandeln. Los geht es. Die Textvorlage für die Methode besteht aus vier Teilen, die wir der Reihe nach abarbeiten.

In der ersten Zeile findest Du *messageSelectorAndArgumentNames.* Diesen Text ersetzt Du durch

```
gib:ersterString bitte:zweiterString und:dritterString.
```

Du erinnerst Dich: Die Argumentnamen *ersterString, zweiterString* und *dritterString* sind Platzhalter für die zugewiesenen Zeichenketten.

In der zweiten Zeile findest Du einen Kommentar *"comment stating purpose message"*. Ja, doppelte Anführungszeichen. Ein Kommentar innerhalb einer Methode muss in doppelten Anführungszeichen stehen. Ersetze diesen Text durch

```
"Eine symbolische Antwort".
```

Lasse am Besten die Anführungszeichen so stehen und überschreibe nur den Text.

Als dritten Eintrag siehst Du nach einer Leerzeile */temporary variable names/*. Hier stehen Namen, die innerhalb der Methode benötigt werden, um die Objekte zu »retten«, weil mit ihnen gearbeitet werden soll. Oft wird erst im Verlaufe des letzten Teils klar, welche Namen benötigt werden. Ihre Angabe ist deshalb erforderlich, um einen möglichen Konflikt mit dauerhaften Namen zu vermeiden. Für Instanzen der Klasse *Svoboda* sind zum Beispiel die Namen *alter* und *vorname* bereits reserviert und können nicht vergeben werden. Falls es zu viele reservierte Namen gibt, kannst Du Dir die Liste über den Knopf *inst vars* anzeigen lassen.

Wir lassen erst einmal diese dritte Zeile unverändert. Und lassen uns bei der eigentlichen Implementierung überraschen, welche *zusätzlichen* Namen notwendig sind. Gehen wir gleich zu dem vierten Teil über, der mit dem roten Wort *statements* versehen ist. Gehen wir ganz langsam vor. Ein Symbol wie *erstesSymbol* versteht den Selektor *capitalized.* Dadurch wird der erste Buchstabe in ein Großbuchstabe verwandelt. Eine solche Verwandlung können wir durch die Nachricht

```
ersterString←ersterString capitalized.
```

vornehmen. Was geschieht hier? Zuerst wird dem String *ersterString* die Nachricht *ersterString capitalized.* gesendet. Das Rückgabeobjekt wird dann für die Zukunft dem Namen *ersterString* zugewiesen. So etwas ist in Ordnung, wenn dadurch sichergestellt ist, dass Du auf das alte, kleingeschriebene Objekt, auf das *ersterString* vorher zeigte, nicht mehr verwenden möchtest. Denn der Zugriff darauf ist dadurch verloren gegangen.

Grundsätzlich ist es aber nicht anzuraten, die Verbindungen der Platzhalter zu ihren Objekten zu ändern. Daher sollte im allgemeinen ein neuer Name verwendet werden. Das kann zum Beispiel dadurch geschehen, dass Du dem Namen des Platzhalters einfach ein *T* für *temporär* anhängst. Du kannst aber auch einen beliebigen anderen Namen verwenden, wenn er nicht mit den Namen der Instanzvariablen übereinstimmt. Gib also in Deine Methode an Stelle von *statements* die Nachrichten

```
ersterStringT←ersterString capitalized.
zweiterStringT←zweiterString capitalized.
dritterStringT←dritterString capitalized.
```

ein. Achte bitte auf die sorgfältige Einrückung. Gib beim Zeilenwechsel nicht nur die RETURN-Taste, sondern STRG-RETURN ein. Dies ist zwar für den Ablauf nicht unbedingt erforderlich. Aber dadurch bleibt der Text übersichtlich. Du wirst erkennen, dass die Namen der neuen temporären Variablen rot geschrieben sind. Das ändert sich, wenn Du sie in der dritten Zeile durch

```
| ersterStringT zweiterStringT dritterStringT |
```

festlegst. Jetzt erzeugen wir noch das Antwortobjekt. Und wir brauchen dafür einen weiteren Namen *antwort*, der ebenfalls in der dritten Zeile deklariert werden muss.

```
antwort←'Bitte, ',ersterStringT,'. Hier ist die ',zweiterStringT,'
und der ',dritterStringT,'.'.
```

Versuche, diese Zeile absolut fehlerlos einzugeben. Du erinnerst Dich? Das Komma »klebt« zwei Strings zusammen. Allerdings gibt es auch ein Komma, das Teil des Textes ist. Versuche es, zu finden.

Vergiß vor allem nicht den letzten Punkt. Denn es kommt noch eine letzte Anweisung. Bis jetzt hat die Methode einen String mit dem Namen *antwort* erzeugt. Es sind aber mit *ersterStringT* usw. auch andere Objekte erzeugt worden. Welches Objekt soll nun am Ende zurückgegeben werden?

Dazu lernst Du nun neben ← das letzte werkzeugartige Zeichen kennen. Dies ist der Rückgabepfeil ↑, den Du mit dem Caret-Zeichen ^ erhältst.

Die letzte Zeile Deiner Methode lautet also:

```
↑antwort
```

Da sie die letzte Nachricht darstellt, ist ein Punkt nicht notwendig. Da es so wichtig ist, wiederholen wir noch einmal den Ablauf des Ganzen:

1. Wenn ein Objekt *sveta* der Klasse *Svoboda* die Nachricht *sveta gib:'marek' bitte:'butter' und:'tee'.* erhält, so ruft *sveta* die Methode *gib:ersterString bitte:zweiterString und:dritterString* auf. Die Platzhalter *ersterString, zweiterString* und *dritterString* werden der Reihe nach den Objekten *'marek'*, *'butter'* und *'tee'* zugeordnet.

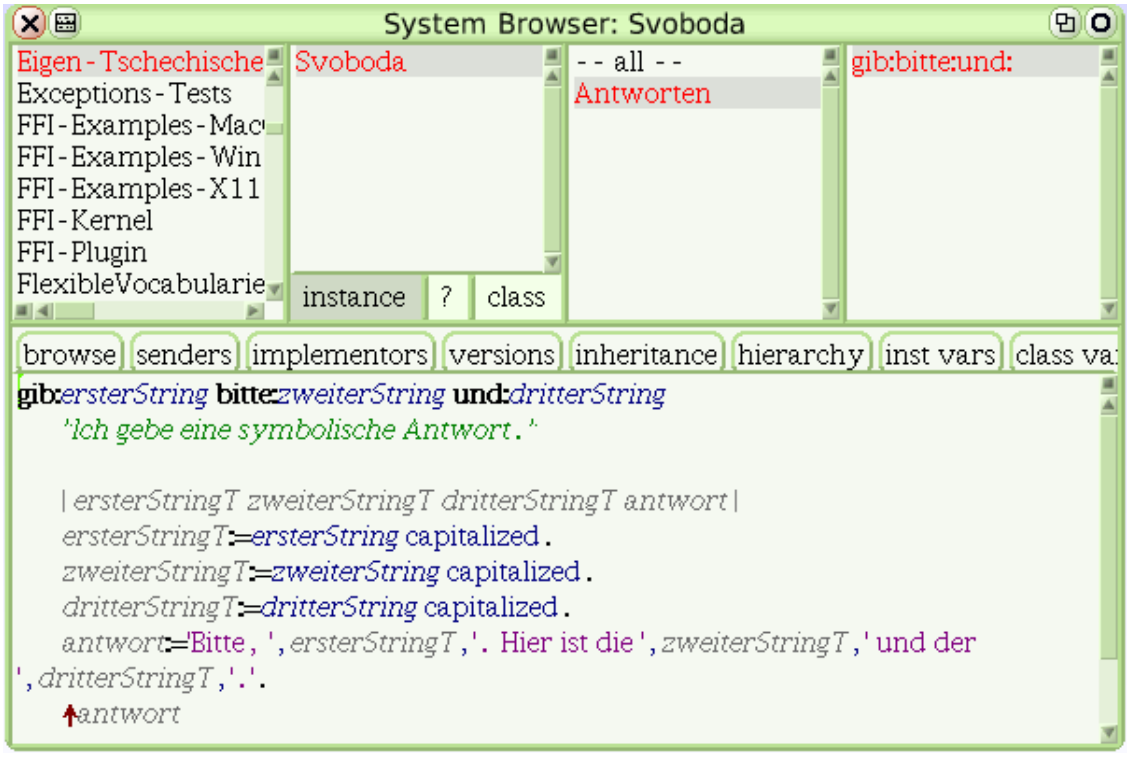

Abbildung 7: Methoden im System-Browser

2. Die zweite Zeile der Methode ist nur ein Kommentar für den Programmierer. Die dritte Zeile dagegen ist ein Kommentar für die VM. Hier stehen die Namen derjenigen Variablen, die *zusätzlich* benötigt werden.

3. Die ersten drei Zeilen des eigentlichen Programms wandeln die empfangenen Zeichenketten so um, dass der erste Buchstabe groß erscheint.

4. Die nächste Zeile generiert einen langen String, wobei mit Hilfe des Kommas die einzelnen veränderten Strings, auf die die Variablen *ersterStingT, zweiterStringT* und *dritterStringT* zeigen, einfach aneinandergehängt werden. Der neue String trägt den Namen *antwort*.

5. Die letzte Zeile läßt *sveta* diesen String als Objekt zurückgeben.

Gut. Es ist wieder der gesamte Text auszuwählen und mit ALT-s zu akzeptieren. Es kann sein, vielleicht auch schon vorher, dass Du dabei nach Deinem Namen oder Deinem Kürzel gefragt wirst. Du weißt, wozu das gut ist.

Wenn alles wie verabredet funktioniert hat, so sollte das Ganze wie in der Abbildung 7 aussehen. Jetzt kommt der große Moment des Testens. Erzeuge in einem Workspace unsere *sveta*, durch

```
sveta ←Svoboda new.
```

Falls das schon vorher geschehen ist und Du den Workspace nicht geschlossen hast, musst Du *sveta* nicht neu erzeugen. Auch das unterscheidet Smalltalk von anderen Sprachen. Änderungen werden sofort bei allen bereits erzeugten Objekten wirksam.

Gib nun im Workspace

```
sveta gib:'marek' bitte:'butter' und:'tee'.
```

ein und führe ein *printIt* aus. Natürlich kannst Du auch als Argumente andere Zeichenketten übergeben. *sveta* wird richtig antworten. Übrigens kannst Du mit

```
misa←Svoboda new.
```

auch eine kleine Schwester erzeugen. Die kann genau dasselbe.

4.5 Private Methoden*

Falls es für Dich zu viel wird, gehe bitte zum Abschnitt 6 auf Seite 37 über und hole die mit Sternchen versehenen Abschnitte später nach.

Unsere kleinen Svobodas haben weder *alter* noch einen *vornamen* zugewiesen bekommen. Oder besser gesagt: Sie wissen nichts darüber.

Erzeuge nun eine neue Methodenkategorie mit dem Namen *privat*. In dieser Kategorie erzeugst Du zunächst die zwei primitiven Methoden *alter* und *alter:einInteger*. Der Sinn der Sache ist folgender: *sveta* soll auf die Nachricht *sveta alter:10* ihre eigene Instanzvariable mit dem Objekt 10 belegen. Auf die Nachricht *sveta alter.* dagegen soll sie den Inhalt der Variablen ausgeben. Störe Dich nicht daran, dass die Rückgabemethode genau so heißt wie die Instanzvariable. Implementiere zuerst die Methode

```
alter:einInteger
   alter←einInteger
```

und dann die Methode

```
alter
   ↑alter
```

Vergiß niemals, jede der Methoden mit ALT-s zu akzeptieren. Üblicherweise werden bei diesen Methoden keine Kommentare gesetzt, da sie sich von selbst verstehen. Ferner benötigst Du keine temporären Variablen. Jetzt kann einer kleinen *misa* gesagt werden, wie alt sie ist:

```
misa alter:4.
```

Und anschließend weiß sie es auch:

```
misa alter.
```

Beachte bitte, dass diese letzte Nachricht *nicht* heißt, dass Du auf die Variable *alter* im Inneren von *misa* zugreifst. Du sendest *misa* vielmehr eine Nachricht, so dass sie mit der entsprechenden Methode ihr eigenes Alter abfragt. Warum haben wir die Methoden *privat* genannt? Nun, weil sie eigentlich nur das Objekt selbst verwenden sollte. Baue jetzt entsprechend die beiden anderen Methoden *vorname:einString* und *vorname* in die Klasse *Svoboda* ein.

4.6 self*

Wie gesagt. Es ist kein guter Stil, mit privaten Methoden *von außen* das Objekt zu belästigen. Wir bauen jetzt eine Methode ein, die jedem *Svoboda* sagt, wie er heißt und wie alt er ist. Wir können den Methodentext jetzt verknappen, denn Du weißt inzwischen, wie eine Methode eingebaut werden muss.

liebe:*einString* duBistJetzt:*einInteger*

```
"Ich weise einem Objekt einen Namen und das Alter zu."
self vorname: einString.
self alter: einInteger.
```

Du siehst hier etwas Neues. *self* ist genau so wie *nil, true* oder *false* eine Pseudovariable. Sie zeigt immer auf den Empfänger der Nachricht. Wenn *sveta* jetzt also die Nachricht

```
sveta liebe:'sveta' duBistJetzt:12.
```

erhält, ruft die Methode die *eigenen* Methoden *vorname:* und *alter:* auf. Eine Methode kann aber nur als Reaktion auf eine *Nachricht* aufgerufen werden. An wen richtet *sveta* also die beiden Nachrichten? An sich selbst. Wenn also eine Methode eine andere Methode aufruft, die zu demselben Objekt gehört, muss diese Nachricht an *self* gesendet werden. Entsprechend muss die unäre Methode

stelleDichBitteVor

```
| antwort |
antwort←'Ich heiße ',self vorname,' Svoboda und bin ',
self alter asString,' Jahre alt.'.
↑antwort
```

mit der Pseudovariablen *self* arbeiten. Wir haben uns hier nicht um die Methodenkategorien gekümmert. Falls Du die Methoden unter *–all–* erstellt hast, werden sie zugleich der Scheinkategorie *as yet unclassified* zugeordnet. Du kannst jederzeit eine neue Kategorie erstellen. Durch einen gelben Klick auf den Namen der Methode im vierten Fenster erhältst Du das Kontexmenü. Hinter dem Eintrag *more...* findet sich ein Fortsetzungsmenü mit dem Eintrag *change category...*. Denke Dir für die eben geschriebenen Methoden eine Kategorie aus, zum Beispiel *Hoefliches* und fuege sie dementsprechend zu.

Im Workspace ist jetzt ein nettes Gespräch mit Mitgliedern der Familie *Svoboda* möglich.

4.7 Automatische Initialisierung einer Instanz*

Es ist wünschenswert, dass die Instanzen einer Klasse nicht ohne Namen und ohne Alter auf die Welt kommen. Eine Möglichkeit besteht darin, eine Methode mit dem Namen *initialize* einzubauen, wie zum Beispiel

initialize

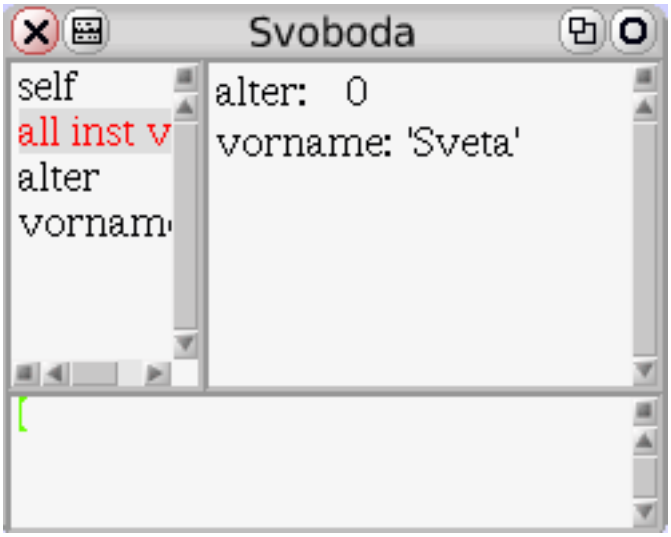

Abbildung 8: Überprüfung eines Objekts im Inspektor

```
self alter:0.
self vorname:'Sveta'.
```

Dass dies wirklich so ist, kannst Du daran überprüfen, dass Du einmal mit

```
marika←Svoboda new.
```

eine ganz neue *Svoboda* einrichtest. Stelle dann einmal den Cursor direkt hinter *marika* und öffne den Inspektor mit ALT-i. Auf dem Rahmen siehst Du die Angabe der Klasse. Der Eintrag *all inst vars* dürfte jetzt klar sein. Du kannst mit dem Inspektor (oder auch mit dem Explorer über ALT-I) direkt in das Objekt hineinsehen. Im Browser siehst Du ausßerdem den Reiter *inheritance* rot aufleuchten, wenn Du im vierten Fenster den Balken auf der Methode *initialize* stehen hast. Klicke einmal auf diesen Reiter. Es öffnet sich ein weiteres Browser-Fenster, dass Dir zeigt, dass die Methode bereits von der Klasse *Proto-Object* vererbt wurde. Du hast sie jetzt überschrieben, so dass die vererbte Methode nicht mehr direkt wirksam ist. Überschreiben einer Methode (*Overriding*) ist eine sehr häufige Technik in der OOP.

4.8 Erweiterte *new*-Methode*

Was machen wir mit dem Vornamen? Er soll nicht von vornherein festliegen. Wir wollen ihn erst während der Geburt mitteilen. Der Schlüssel für dieses Problem liegt in einer erweiterten *new* Methode. Wir wollen in Zukunft über *Svoboda new: 'sveta'* dem neuen Objekt mitteilen wie es heißt. Wie Du siehst, muss die Nachricht von der *Klasse* verstanden werden, aber nicht von der Instanz, die dadurch ja erst entsteht. Wenn Du jetzt die folgende Methode in den Browser eingibst, musst Du dafür sorgen, dass unter dem zweiten kleinen Fenster der Knopf *class* gedrückt ist:

newSvoboda:*einString*

```
"Ich erzeuge eine Instanz und belege die Variable vornamen."
| instanz |
instanz←self new.
instanz vorname:einString.
↑instanz
```

Falls Du versuchst, die Methode *new:einString* einzurichten, erhältst Du eine Warnung. Damit würdest Du einer sehr wichtige Methode überschreiben. Also verwenden wir den Methodenselektor *newSvoboda:*. Danach sollte aus der Methode *initialize* die Zeile *self vorname:'Sveta'* natürlich entfernt werden. Wenn Du jetzt in einem Workspace

```
mairi ←Svoboda newSvoboda:'mairi'.
```

ausführst und Dir danach *mairi* im Inspektor anschaust, wirst Du sehen, dass unser Ziel erreicht ist.

4.9 Einrichten einer Subklasse*

Vielleicht ist es Dir vorhin schon aufgefallen: Die Methode *stelleDichBitteVor* passt nicht für weibliche Mitglieder der Familie *Svoboda*. Üblicherweise wird an den Familiennamen die Endung *ova* angehängt. Natürlich gibt es viele Möglichkeiten, durch Hinzufügen einer neuen Instanzvariablen *geschlecht* oder einer entsprechenden Veränderung der Methode das Problem zu lösen.

Wir werden aber einen anderen Weg wählen. Wir bilden eine *Subklasse* der Klasse *Svoboda*, die *Svobodova* heißt. Wie patriarchalisch! Die »männliche« Klasse steht in der Hierarchie über der »weiblichen«. Ja, aber die »weibliche« kann mehr. Sie erbt alle »Fähigkeiten« und »Eigenschaften« der Superklasse und fügt diesen eventuell noch einige weitere hinzu. Andererseits können geerbte Methoden auch durch Überschreiben abgeändert werden.

Statt nun irgendeinen selbst definierten Realitätsbezug zu erzwingen, sollten wir die Sicht der Dinge noch einmal überdenken: Jede *Svobodova* ist auch eine *Svoboda*. Das umgekehrte gilt nicht. Gut. Aber die Klasse *Svoboda* kann gar nicht entscheiden, ob es sich um ein weibliches oder männliches Mitglied handelt. Die Unterklasse *Svobodova* soll genau dies können. Im Falle eines Mädchens sollen zusätzliche »Fähigkeiten« zur Verfügung stehen. Mit anderen Worten: Die Klasse *Svobodova* soll auch »männliche Instanzen« bilden können. Aber diesen sollen dann keine zusätzlichen Fähigkeiten zur Verfügung stehen. Auf jeden Fall benötigt *Svobodova* eine weitere Instanzvariable mit der Bezeichnung *geschlecht*.

Erinnerung An dieser Stelle sei Dir noch einmal darüber im Klaren: Was wir im Moment tun, ist Smalltalk-Hacking. Wir wollen ein paar grundlegende Gedanken verstehen. Mehr nicht. Es könnte sein, dass Du die Idee einer Subklasse für übertrieben oder umständlich hältst. Darin stimme ich Dir sogar zu. Aber: Wir lernen sozusagen erst die Tonleiter und die »Spielregeln« kennen. Die Komposition eines guten Stückes ist für das nächste Kapitel vorgesehen. Objektorientiertes Design kann einen sehr großen Spaß machen! Aber es ist eine *echte* Kunst!! Das Verstehen-Lernen, was wir gerade betreiben, ist dagegen zwar notwendig, aber *keine* wirkliche Kunst. Wer sich darauf beschränkt, gelangt über das Hacking-Niveau nicht hinaus. Insofern sieh dieses Kapitel und vor allem das Beispiel bitte nicht als der Weisheit letzter Schluß an. Allein, dass wir *während* des Programmierens uns überlegen, was wir eigentlich wollen, ist nur in dieser ersten Phase akzeptabel.

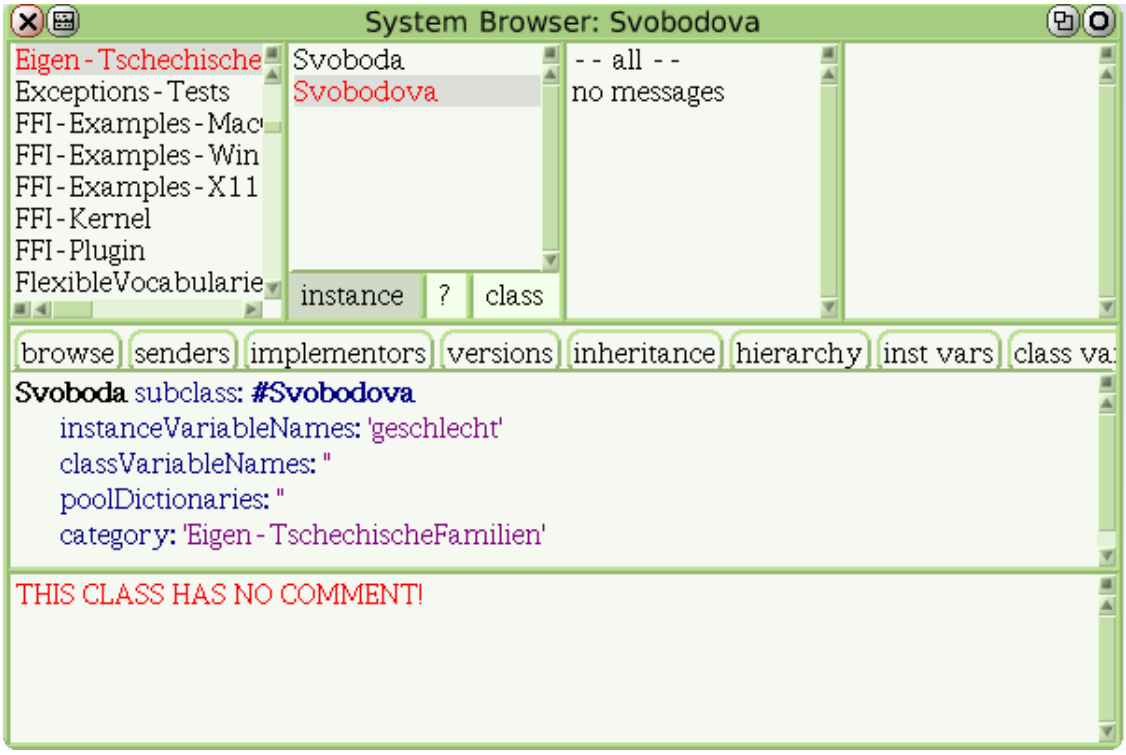

Abbildung 9: Eintragen einer Subklasse

Das Einrichten einer Subklasse ist denkbar einfach. Wir haben ja schon mit *Svoboda* in Wirklichkeit eine Subklasse der Superklasse *Object* gebildet. Jetzt gehen wir ganz genau so vor. Wir ersetzen nur in der Textschablone das Wort *Object* durch *Svoboda*. Die Textschablone für die Einrichtung einer Klasse erhältst Du durch Anklicken der Kategorie *Eigen-TschechischeFamilien* im linken kleinen Fenster des Browsers. Dazu kannst Du *irgendeine* Kategorie anklicken und dann einfach anfangen, die Bezeichnung der Kategorie zu schreiben (keine Großbuchstaben verwenden!). Der Balken wird dann in die Nähe der Kategorie springen.

Achte unbedingt darauf, dass Du nicht den Text der schon fertigen Klasse *Svoboda* verwendest. Wo Du den Namen *Svobodova* und die neue Instanzvariable *geschlecht* eintragen musst, weißt Du schon. Zusätzlich musst Du aber das Wort *Object* ersetzen. Denn die neue Klasse soll eine Subklasse von *Svoboda* sein[6]. Vergiß nicht, Dein Werk mit ALT-s bekannt zu machen. Wenn alles wie abgesprochen durchgeführt wurde, sollte das Ganze so aussehen wie in der Abbildung 9 . Den Kommentar kannst Du Dir selbst ausdenken.

Jetzt kannst Du auf die herkömmliche Weise Instanzen erzeugen. Obwohl wir in die Unterklasse noch keine Methoden eingebaut haben, funktioniert

```
sveta←Svobodova newSvoboda:'sveta'.
```

wunderbar. Schaue Dir auch Deine erzeugte Instanz mit dem Inspektor an. Die Variable *geschlecht* weist momentan noch auf kein Objekt. Nun, es fehlt dreierlei:

1. eine private Methode *geschlecht:einCharacter*, die der Variablen *geschlecht* die Buchstaben $w oder $m zuordnet.

[6] Damit ist sie allerdings auch automatisch eine Subklasse von *Object*, denn *Svoboda* ist ja direkt von *Object* abgeleitet.

2. eine private Methode *geschlecht*, die das Objekt zurückgibt, auf das die Variable *geschlecht* verweist,

3. eine neue *Klassen(!)methode* mit der Bezeichnung *newSvoboda:einString genus:einCharacter*, die sowohl den Namen als auch das Geschlecht zuweist.

Die Methoden *geschlecht* und *geschlecht:einCharacter* solltest Du jetzt selbst eingeben können. Für die Klassenmethode schalte den Browser wieder auf *class* um. Dann baue die folgende Methode ein:

newSvobodova:*einString* genus:*einCharacter*

```
| instanz |
instanz←self newSvoboda:einString.
instanz geschlecht:einCharacter.
↑instanz
```

Der Aufruf

```
mairi←Svobodova newSvobodova:'mairi' genus:$w.
```

weist nun *allen* Variablen Objekte zu. Du hast sicherlich ein paar Fragen:

1. Warum kann zur Instanzbildung *Svobodova newSvoboda:einString* verwendet werden? Erzeugt *newSvoboda* denn nicht ein Objekt mit nur *zwei* Variablen *alter* und *vorname*? Antwort: Dass die Methode von *Svoboda* vererbt wurde, ist Dir sicherlich klar. Aber eine Instanz mit nur zwei Variablen wird mit der Nachricht *Svoboda newSvoboda:einString* erzeugt! In diesem Fall wird die Nachricht aber *Svobodova* gesendet. *Svobodova new.* erzeugt eine Instanz mit *drei* Variablen. Im Prinzip unterscheidet sich *newSvoboda:einString* von *new* aber nur dadurch, dass *zusätzlich* die Variable *vorname* belegt wird.

2. Warum kann *Svobodova* die Variable *alter* belegen? Antwort: Weil die Klasse *Svoboda* die Methode *initialize* vererbt. Natürlich wird sie auch bei der Instanzbildung von *Svobodova* automatisch aufgerufen.

Es mag Dir gespensterhaft erscheinen, dass eine Instanz Methoden aufrufen kann, die sie gar nicht besitzt. Diese Sichtweise ist aber nur halb richtig. *Tatsächlich* ist zum Beispiel die Methode *stelleDichBitteVor* nicht in der Klasse *Svobodova* implementiert. Wenn *Svobodova* nun eine Nachricht erhält, für die es in der Klasse keine Methode gibt, sucht die VM in der Superklasse *Svoboda*. Wenn die VM auch dort keine Methode findet, sucht sie in der Superklasse von *Svoboda*. Das ist die Klasse *Object*. Spätestens hier muss sie aber eine Methode finden, denn *Object* besitzt keine Superklasse. Wenn auch dort keine passende Methode vorliegt, wird die Nachricht *Message not understood* ausgeben.

4.10 Blöcke und Verzweigungen*

Als letzte Tat werden wir nun die geerbte Methode *stellDichVor* überschreiben. Für Instanzen von *Svobodova* soll die Methode im Wesentlichen dasselbe durchführen. Im Falle einer weiblichen Instanz soll jedoch noch ein zusätzlicher Satz ausgegeben werden. In diesem Zusammenahng begegnet uns zum ersten Male eine *Verzweigung* in Smalltalk. Eine Verzweigung ist eine Entscheidungsabfrage. Wenn irgendetwas richtig ist, soll eine bestimmte Sache ausgeführt werden. Sonst nichts, oder irgendetwas Anderes.

Smalltalk hat für diesen Zweck ein enorm starkes Konstrukt: den so genannten *Block*. Ein Block besteht aus einer beliebigen Zahl von Nachrichten, die durch eckige Klammern eingeschlossen sind. So ist zum Beispiel

```
[3+4.]
```

ein besonders einfacher Block. Anweisungsblöcke gibt es auch in anderen Programmiersprachen. Weder in C++, noch in Java oder Python sind Blöcke aber selbst Objekte. Ein Block wird erst dann ausgeführt, wenn an ihn eine Nachricht gesendet wird, die den Selektor *value* enthält. In unserem Beispiel wird die innere Nachricht also erst dann ausgeführt, wenn Du an den Block die Nachricht

```
[3+4] value.
```

sendest. Probiere es aus.

Blöcke können nun unter anderem für Entscheidungen (Verzweigungen) eingesetzt werden. Führe einmal die folgende Nachricht aus:

```
true ifTrue:[3+4.].
```

Das boolesche Objekt *true* versteht den Nachrichtenselektor *ifTrue:einBlock*. Das Objekt *true* sendet an den Block die Nachricht *[3+4.] value*. Das boolesche Objekt *false* versteht diesen Selektor auch. Aber es reagiert auf die Nachricht nicht. Probiere es aus.

Interessanter ist der Selektor *ifTrue:einBlock ifFalse:nochEinBlock*. Wenn Du

```
true ifTrue:[3+4.] ifFalse:[1-2.].
```

ausführst, wird dem *ersten* Block die *value* Nachricht geschickt. Das Objekt *false* sendet folglich bei diesem Selektor dem *zweiten* Block die *value*-Nachricht. Entsprechend gibt es auch den Selektor *ifFalse:ifTrue:*.

Selbstverständlich werden die booleschen Objekte nicht explizit angesprochen, sondern durch eine entsprechende Nachricht erzeugt. So liefert zum Beispiel die Nachricht

```
2>1.
```

das Objekt *true*. Eine Entscheidung kann nun so ausgedrückt werden:

```
(2>1) ifTrue:[↑'Das ist richtig.'].
```

Die runden Klammern sind nicht unbedingt notwendig. Sie erhöhen jedoch die Lesbarkeit. Es würde bei

```
2>1 ifTrue:[↑'Das ist richtig.'].
```

nicht die Gefahr bestehen, dass der Zahl 1 die *tfTrue:* Nachricht geschickt wird, denn der binäre Selektor > wird zuerst ausgewertet. Sehr wohl gibt es aber im folgenden Fall Konflikte:

```
array←#(5 2).
5=array at:1 ifTrue:[↑'Das ist richtig.'].
```

Hier würde dem Objekt *array* der Selektor *at:ifTrue:* gesendet. Diese Nachricht wird von einem Array aber nicht verstanden. Die folgende Zeile

```
(5=array at:1)ifTrue:[↑'Das ist richtig.'].
```

ist schon besser, aber auch noch nicht fehlerfrei. In der runden Klammer hat der Selektor = Vorrang vor *at:* . Es wird also eine Zahl mit dem Objekt *array* verglichen, was nicht möglich ist. Die korrekte Zeile lautet:

```
(5=(array at:1)) ifTrue:[↑'Das ist richtig.'].
```

Wir werden die Verzweigung gleich für das folgende »Problem« verwenden:

```
geschlecht←$w.
(geschlecht=$w) ifTrue:['Genau genommen heiße ich eigentlich Svobodova.'].
```

4.11 super*

Jetzt ändern wir für die Klasse *Svobodova* die geerbte Methode *stelleDichBitteVor* ein wenig ab. Damit *überschreiben* wir die Methode. Wir werden zunächst die neue Methode noch nicht vollständig implementieren, sondern zunächst nur so:

stelleDichBitteVor

```
antwort←''.
zusatz←'Genau genommen heiße ich eigentlich Svobodova.'.
(self geschlecht=$w) ifTrue:[antwort←antwort,zusatz.].
↑antwort
```

Wenn Du nun ein Mädchen mit

```
mairi←Svobodova newSvobodova:'mairi' genus:$w.
```

einrichtest, so erscheint nach der Message

```
mairi stelleDichBitteVor.
```

der Satz *Genau genommen heiße ich eigentlich Svobodova.* Wenn Du aber ein Objekt der Klasse *Svobodova* einrichtest, das männlich ist, so erscheint gar keine Reaktion. Die vererbte Methode kann nicht aufgerufen werden, da sie überschrieben wurde. Nun ändern wir unsere neue Methode ab:

stelleDichBitteVor

```
antwort←super stelleDichBitteVor.
zusatz←'Genau genommen heiße ich eigentlich Svobodova.'.
(self geschlecht=$w) ifTrue:[antwort←antwort,zusatz.].
↑antwort
```

Was ist in der ersten Zeile geschehen? Hier wird eine Nachricht an die fünfte Pseudovariable *super* gesendet (die anderen vier sind: *nil, true, false* und *self*). Die Variable *super* zeigt genau so wie *self* auf das Empfängerobjekt. Der einzige Unterschied ist der Startpunkt der Methodensuche. Die VM beginnt mit der Methodensuche bei Nachrichten an *self* immer in der Klasse der Empfängers. Wird dagegen eine Nachricht an *super* gesendet, so beginnt die Methodensuche in der *Superklasse* des Empfängers. Mit anderen Worten: *super stelleDich-BitteVor* wird mit einem Aufruf der »alten« in der Klasse *Svoboda* befindlichen Methode beantwortet. Das Ganze hat den Effekt, dass sich jetzt jede Instanz von *Svobodova* genau so vorstellt wie eine Instanz der Klasse *Svoboda*. Bei »weiblichen Objekten« wird jedoch auch noch der Zusatzspruch ausgeführt.

Mit Hilfe von *super* gelingt es also, überschriebenen Code doch noch verfügbar zu machen, weil die Methodensuche in der Superklasse des Empfängers beginnt.

5 Informationen abfragen

Bei dem gewaltigen Gewirr von Klassen im System stellst Du Dir sicherlich die Frage, wie ein Programmierer sich da noch zurechtfinden soll. Die Hauptfragen sind sicherlich:

1. Ich habe etwas ganz Bestimmtes vor. Gibt es eine Klasse, die meinen Wunsch erfüllen kann?
2. Ich habe eine konkrete Klasse vor mir. Wie finde ich die Superklassen heraus, von der meine Klasse etwas erbt?
3. Ich habe eine konkrete Klasse vor mir. Ich möchte nicht alle Superklassen durchsehen. Wie finde ich heraus, welche Methoden die Instanzen meiner Klasse beherrschen?
4. Ich habe eine konkrete Klasse vor mir. Wie finde ich heraus, welche anderen Klassen diese Klasse benutzen?
5. Ich habe *keine* konkrete Klasse vor mir, sondern ich suche eine bestimmte Methode. Wie finde ich heraus, welche Klasse diese Methode verwendet?

Irgendwie gelangst Du *immer* an das, was Du suchst. Natürlich hilft Dir dabei am meisten die Erfahrung. Zu Beginn darfst Du Dich nicht enrmutigen lassen, dass das Suchen noch recht zeitintensiv sein kann.

5.1 Suche nach einer Klasse

Nehmen wir einmal an, Du möchtest Dich in die Programmierung von Fenstern beschäftigen. Gibt es irgendeine Klasse, die *Window* oder so ähnlich heißt? Der Weg führt über den System-Browser. Einerseits kannst Du natürlich zunächst Dir die einzelnen Kategorien anschauen. Es lohnt sich, denn es sind so viele nun auch wieder nicht. Effektiver ist es aber vielleicht, das Kontextmenü im linken kleinen Fenster zu aktivieren und mit *find class...* oder ALT-f den Namen *Window* einzugeben. Du wirst dann eine ganze Reihe von Klassen angeboten bekommen.

Schon recht konkret wäre die Suche nach einem Zufallsgenerator. Hätten wir nicht schon früher über ihn gesprochen, hättest Du sicherlich von selbst nach *Random* gesucht.

5.2 Informationen über eine konkrete Klasse

Jetzt stellen wir uns vor, Du hast eine konkrete Klasse vor Dir. Zum Beispiel *Svoboda*. Es gibt nun mehrere Möglichkeiten, etwas über die Klasse zu erfahren.

Natürlich kannst Du wie eben im Browser suchen. Es ist aber auch möglich, in den Workspace den Namen *Svoboda* zu schreiben und den Cursur hinter diesen Namen zu stellen. Wenn Du dann ALT-b drückst, öffnet sich der Browser schon an der richtigen Stelle.

Um zu erfahren, welche Superklassen und welche Subklassen *Svoboda* hat, klicke im Browser einfach auf den Knopf *hierarchy*. Es öffnet sich ein ganz ähnlich aussehender Browser, der im linken kleinen Fenster den Stammbaum anzeigt. Hier siehst Du, dass *Svoboda* Subklasse von *Object* ist und selbst wieder eine Unterklasse *Svobodova* besitzt. Verwechsle den Knopf *hierarchy* nicht mit *inheritance*. Der Knopf *inheritance* ist nur bei überschriebenen Methoden aktiv. Es ist über ihn dann möglich die entsprechenden Methoden der Superklassen einzusehen.

Eine andere Möglichkeit besteht darin, im Hauptfenster die Hierarchie anzuzeigen. Dazu muss Du auf der Klasse *Svoboda* über das Kontextmenü den Eintrag *show hierarchy* anklicken.

Wichtiger mag Dir erscheinen, welche *Methoden* die Instanzen der Klasse beherrschen. Dafür gibt es den Menüpunkt *browse protocol*, den Du auch mit ALT-p aktivierst. Dann erscheint der Protokoll-Browser. Im linken oberen Fenster sind alle Kategorien aufgelistet. Im rechten findest Du die Methoden. Die speziell zur Klasse gehörigen Methoden sind fettgedruckt. Aller vererbten Methoden sind in normaler Schrift angegeben.

Um herauszufinden, welche anderen Klassen die vorliegende Klasse benutzen, kannst Du über das Kontextmenü unter dem Eintrag *class refs* mehr erfahren. Alternativ kannst Du auch ALT-n eingeben. Unsere Klasse *Svoboda* wird von keiner anderen Klasse verwendet. *Svobodova* ist zwar Unterklasse von *Svoboda*. Aber sie sendet keine Nachrichten an die Klasse.

Die Reiter in dem System-Browser dienen ausschließlich dazu, etwas über die Methoden zu erfahren. Um *senders* und *implementors* benutzen zu können, muss eine Methode der Klasse bereits ausgewählt sein. In jedem Falle öffnet sich ein weiterer Browser. *senders* gibt an, welche Methoden der Klasse die aktuelle Methode in Nachrichten verwenden. *implementors* gibt an, wie oft die Methode in dem Vererbungsbaum eingerichtet wurde. Wenn Du die Methode *stelleDichVor* untersuchst, so findest Du, dass diese Methode sowohl in der Klasse *Svoboda* als auch in der Klasse *Svobodova* implementiert wurde.

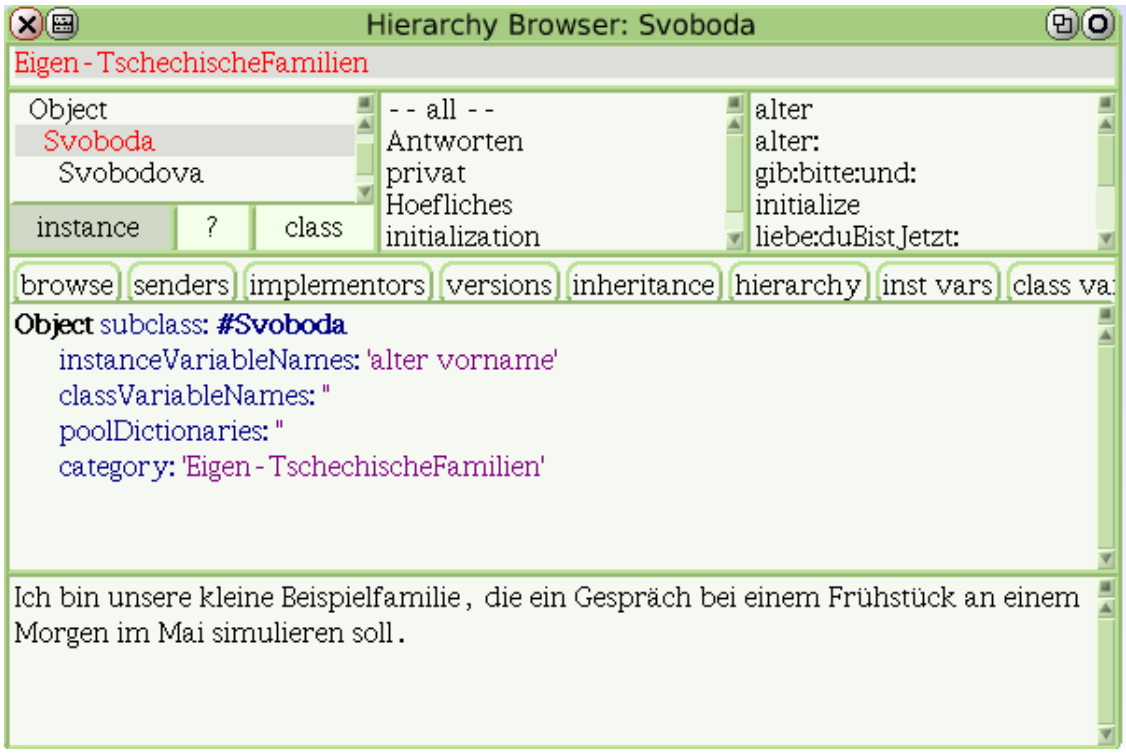

Abbildung 10: Hierarchie-Browser

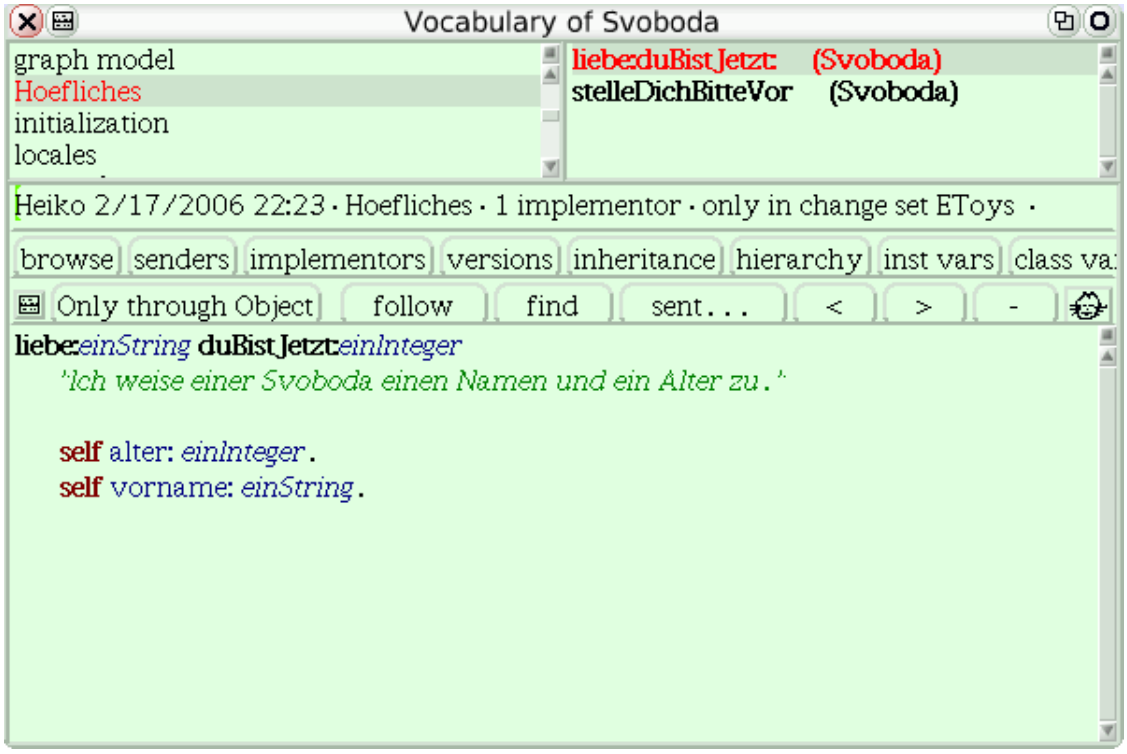

Abbildung 11: Der Protokoll-Browser

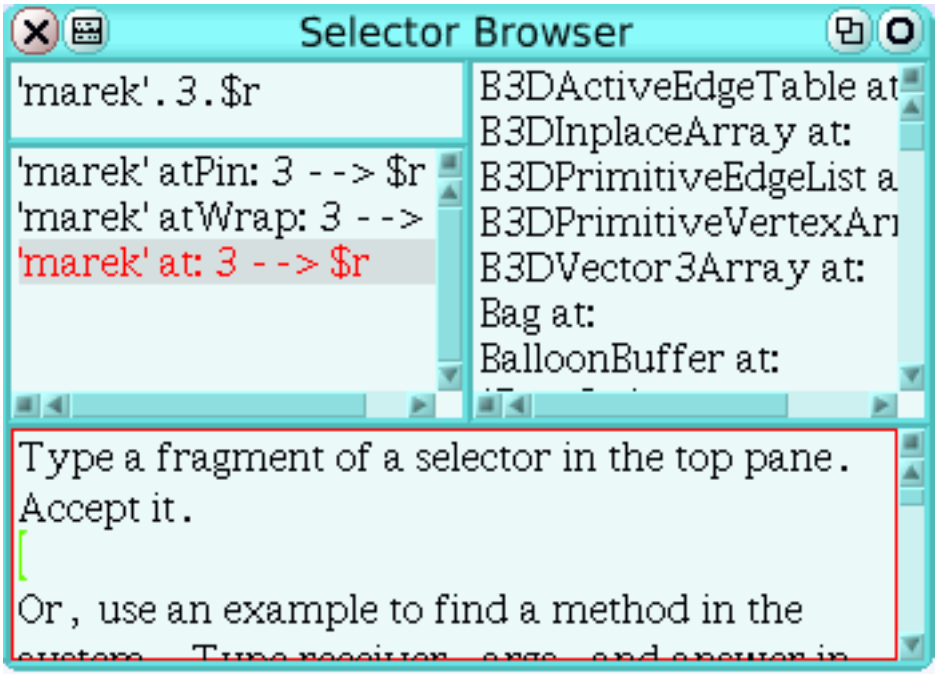

Abbildung 12: Der Methoden-Sucher

5.3 Methodensuche bei unbekannter Klasse, aber bekanntem Methodennamen

Nehmen wir an, Du kennst den Namen einer bestimmten Methode. Denken wir zum Beispiel an *size*. Du weißt aber nicht, welche Klassen die Methode verwenden. Dazu gibt es in der Werkzeugleiste zwei Werkzeuge: den *Message-Finder* und den *Method-Finder*. Wenn Du mit dem Nachrichten-Sucher (*message finder*) nach der Methode *size* suchst, so erhältst Du alle Methoden, die das Wort *size* irgendwie enthalten. Im rechten Fenster tauchen die expliziten Nachrichten auf und im Hauptfenster der genaue Code der entsprechenden Methode.

Der Method-Finder ist noch mächtiger. Hier kannst Du irgendein Bruchteil des Namens der Methode angeben. Im Großen und Ganzen gleicht er bis zu diesem Punkt dem Nachrichten-Sucher. Doch im nächsten Abschnitt kommt das Besondere.

5.4 Methodensuche bei unbekanntem Methodennamen

Der Methoden-Sucher verfügt über ungeheure Fähigkeiten, die in so manchen anderen Smalltalk-Systemen fehlen. Er kann Methoden mit Hilfe von Beispielen finden. Gibst Du im oberen linken Fenster *'marek'.5* ein, so zeigt er Dir im darunterliegenden Fenster mögliche Messages. Darunter findest Du ebenfalls *size*. Gib nun *'marek'.3.$r* an, so findet er tatsächlich den Selektor *at:*. Es gibt noch eine weitere Möglichkeit, die im Hauptfenster ganz unten beschrieben ist. Lies Dir den Text einmal durch und führe die Beispiele aus. Verwende auch unsere beiden Beispiele.

6 Transcript

Bei allen unseren Beispielen haben wir den Workspace als Empfänger für Antworten verwendet. Wenn Objekte aber Strings ausdrucken sollen, ohne dass dieser String zugleich das

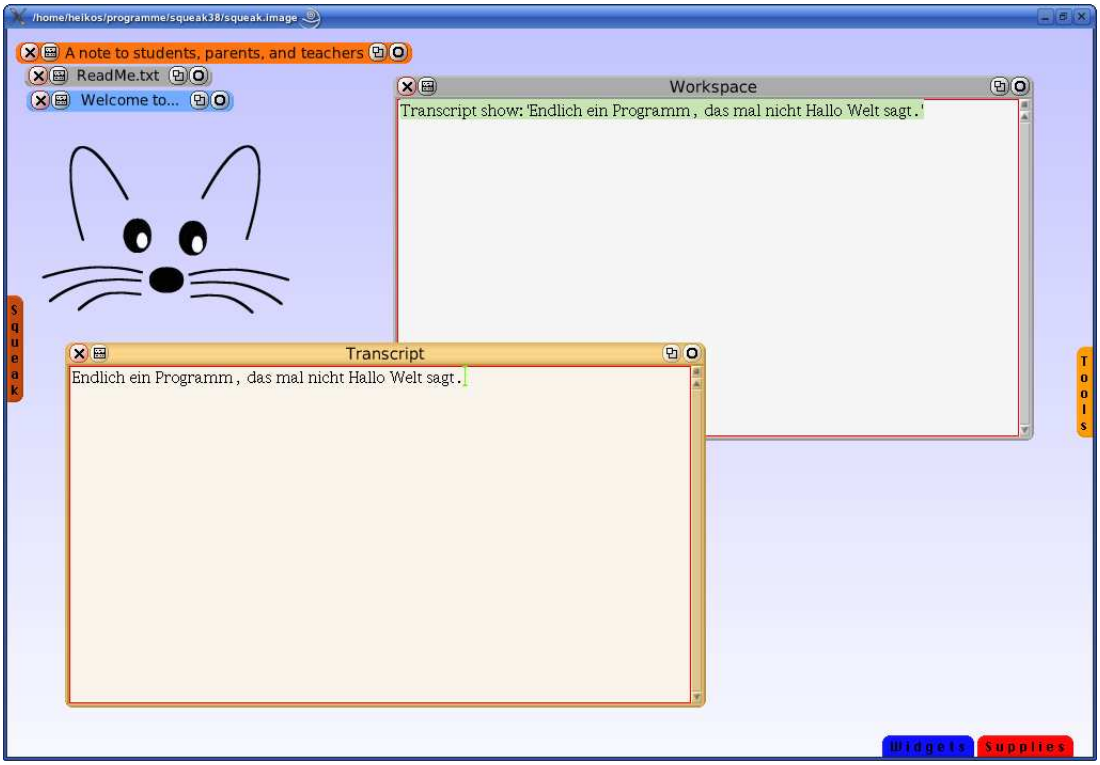

Abbildung 13: Das Transcript-Fenster

zurückgegebene Objekt ist, benötigen wir ein spezielles Ausgabefenster: Das sogenannte *Transcript*. Ein solches Transcript-Fenster kannst Du mit ALT-t öffnen. Das Fenster ist allerdings weitestgehend *passiv*. Du kannst nichts hineinschreiben. Aber es können Nachrichten an dieses Fenster gesendet werden. Natürlich sendest Du diese Nachrichten über den Workspace.

```
Transcript closeAllViews.
```

schließt alle offenen Transcript-Fenster. Die Nachricht

```
Transcript open.
```

öffnet natürlich ein solches Fenster wieder. Um einen Text in dem Fenster auszudrucken, sendest Du zum Beispiel:

```
Transcript show:'Endlich einmal ein Programm, das nicht Hallo Welt schreibt.'
```

Bei der nächsten *show*-Nachricht wird allerdings der folgende Text angehängt. Um das zu verhindern, kannst Du einen Zeilenumbruch durch

```
Transcript cr.
```

erreichen. Ja, und um den Text zu löschen, ohne das Fenster zu schließen gibt es – na, rate mal –

```
Transcript clear.
```

7 Schleifen

Wir können nun am Ende unserer *wilden Reise* noch das zweite, wichtige Konstrukt neben den Verzweigungen ansprechen: die Schleifen. Über Verzweigungen und Schleifen verfügt *jede* Programmiersprache. Eine Schleife wiederholt einen Auftrag so lange, bis eine Abbruchsnachricht die Schleife beendet.

Die einfachsten Schleifen führen einen Auftrag eine bestimmte Anzahl von Malen aus. In den meisten Programmiersprachen sind das die so genannten *For-Schleifen*. In Smalltalk wird dagegen eine Nachricht an eine Integer-Zahl gesendet. Zum Beispiel führt

```
4 timesRepeat: [Transcript show:'Dobry den.';cr].
```

viermal den originellen Gruss aus. Genau genommen sendet die Zahl 4 viermal eine *value*-Nachricht an den Block. Innerhalb des Blocks siehst Du etwas Neues. *Transcript* ist das Objekt, an das eine Nachricht gesendet wird. In Wirklichkeit sind es zwei Nachrichten. Die erste enthält den Selektor *show:* und die zweite den Selektor *cr*. Das Semikolon erspart es uns, das Empfängerobjekt *Transcript* noch einmal zu schreiben. Ein Selektor, der auf ein Semikolon folgt gehört zu einer Nachricht, die an dasselbe Objekt gesendet wird.

Die zweite Schleifenart nennt sich in den meisten Sprachen *While-Schleife*. Eine While-Schleife sendet eine *value*-Nachricht so lange an einen Block, bis eine bestimmte Bedingung sich verändert. Dafür gibt es die Selektoren *whileTrue:einBlock* und *whileFalse:einBlock*. Auf den ersten Blick scheinen sie mit den Selektoren *ifTrue:einBlock* und *ifFalse:einBlock* verwandt zu sein. Das ist aber nicht richtig. Boolesche Objekte verstehen die *while*-Nachrichten nicht, sondern nur Blöcke. Schauen wir uns ein Beispiel an:

```
zaehler←0.
[zaehler<10] whileTrue: [
    zaehler←zaehler+1.
    Transcript show:'Ich bin noch nicht fertig.';cr].
Transcript show:'Jetzt bin ich fertig.'.
```

Probiere diesen Code in einem Workspace aus. Auch hier siehst Du etwas Neues. Wenn ein Block mehrere Nachrichten enthält, so ist es üblich, nach der öffnenden Klammer [in die nächste Zeile zu wechseln. Jede Nachricht des Blocks wird in eine neue Zeile geschrieben *und* eingerückt. Erst nach der schließenden Klammer] wird mit der alten Einrückung fortgefahren. Die letzte Nachricht ist also kein Bestandteil des Blocks.

Was passiert genau? Die While-Nachricht wird an den ersten Block *[zaehler<10]* gesendet. Dieser sendet sich daraufhin selbst die Nachricht *value*. Wenn das übergebene Objekt *true* ist, so wird dem zweiten Block eine *value*-Nachricht geschickt. Genauergesagt, schickt der Block dem entstehenden Booleschen Objekt eine *ifTrue*-Nachricht, wobei der zweite Block übergeben wird.

So sind While-Schleifen gebaut. Noch einmal: Gehe Smalltalk nicht in die Falle, indem Du eine While-Nachricht direkt an ein Boolesches Objekt schickst. Diese verstehen die Nachricht nicht! Eine While-Nachricht muss *immer* an einen Block gesendet werden.

8 Kleine Kontrollfragen

Wir sind endlich am Ende unserer *wilden Reise*. Es wäre schön, wenn Du Dich nicht im Gestrüpp verloren hast. Es war sicherlich ein bißchen viel auf einmal. Aber das ist gut so. Denn nun wird es wirklich sehr angenehm. Die *echten* Smalltalk-Kapitel vertiefen nicht nur das, was Du jetzt schon weißt. Sie werden Dich in die Kunst des OOP einführen. Und dann beginnt das Ganze hoffentlich richtig Freude zu machen.

Frage 1 *Welche Objekte lassen sich durch Literale darstellen?*

Frage 2 *Was ist eine Instanz? Wie werden Instanzen erzeugt?*

Frage 3 *Welcher Unterschied besteht zwischen Symbolen und Strings?*

Frage 4 *Welche Arten von Variablen hast Du kennengelernt?*

Frage 5 *Noch einmal: Worin besteht der Unterschied zwischen einer Message und einer Methode?*

Frage 6 *Welches Objekt gibt die Nachricht 3+4/6 zurück?*

Frage 7 *Besitzen wirklich alle Klassen Superklassen?*

Frage 8 *Von wie vielen Klassen kann eine Klasse Subklasse sein?*

Frage 9 *Das Objekt mairi verwendet in einer Methode jeweils eine Nachricht an das Objekt self und an das Objekt super. Auf welches Objekt verweisen self und super?*

Index