# Squeak I
# Die Reise in eine bunte Welt

Heiko Schröder

22. Februar 2006

## Inhaltsverzeichnis



## Abbildungsverzeichnis


Abbildung 1: Der Urwald von Squeak

# 1 Einleitung

> Das letzte Abenteuer auf unserem Planeten ist noch lange nicht erlebt.
> Heinrich Harrer

Squeak ist wirklich ein Abenteuer. Eine Reise in einen bunten Urwald. In der Abbildung 1 siehst Du, was Dich gleich erwartet. Ja, und was ist das Ganze? Nun, eine ganz trockene Antwort: Squeak ist eine Umgebung zum Programmieren. Ja, und was soll daran so toll sein? Programmierumgebungen gibt es *viele*. Vielleicht ist Squeak aber *anders*. Anders als die Umgebungen anderer Programmiersprachen. Und es stimmt. Genau das wollte ich Dir gerade sagen. Squeak sieht auf den ersten Blick ein wenig »kindlich« aus. Das soll es auch. Squeak ist eine Programmierumgebung *auch* für Kinder. Aber es ist *kein* »Kinderkram«. Vor allem die liebe Squeak-Maus könnte Dich vielleicht auf diesen Gedanken bringen. Das ist aber völlig falsch. Squeak ist ein Programmiersystem, dass sich an Profis richtet. Was denn nun? Kinder oder Profis? Ganz einfach: In kürzester Zeit *werden* Kinder zu echten Profis. So ist das.

Um welche *Programmiersprache* geht es denn? Heißt die Sprache Squeak? Ja, das könntest Du so sehen. *Eigentlich* heißt die Sprache *Smalltalk*. Genauer gesagt: *Smalltalk-80*. Allerdings mit erheblichen Erweiterungen. Vielleicht hast Du diesen Namen schon einmal gehört. Vielleicht wirst Du sogar sagen: »Eine uralte Sprache«. Ja, aber was heißt »uralt«? C++ entstand ja schon drei Jahre nach Smalltalk-80. C ist eher noch älter. Vielleicht fragst Du weiter: »Wer programmiert heute noch mit Smalltalk?« Nun, zum Beispiel jeder, der mit Java zu tun hat. Er merkt es nur nicht. Die Smalltalk-Ideen befinden sich im *Inneren* von Java. Das was nach außen hin sichtbar ist, die Syntax der Sprache, orientiert sich dagegen – leider, leider – an C++.

Was ist denn das Besondere an dieser Sprache *Smalltalk*? Nun, ich könnte wahrscheinlich den größten Teil der etwas älteren unter uns sofort mit dem Begriff *Objektorientierte Programmierung (OOP)* von der weiteren Lektüre abhalten. Dabei hat diese Denkweise sehr, sehr viel mit der Art und Weise zu tun, mit der Vorgänge in der Natur ablaufen. Keine andere Programmiersprache zeigt eine engere Naturverbundenheit als Smalltalk. Aber trotzdem läuft immer noch einigen Programmierern bei OOP ein kalter Schauer über den Rücken. Warum? Nun, weil sie wahrscheinlich nicht das Glück hatten, diese Art der Denkweise mit einer *konsequenten* Sprache zu lernen. C++ oder Java nennen sich zwar objektorientiert, sie sind es aber nicht – jedenfalls nicht wirklich. Auch das sehr schöne Python macht dabei keine Ausnahme.

Die erste Reise, die wir nun unternehmen, findet zunächst nur virtuell, also noch nicht am Computer, statt. Versuchen wir das Prinzip von Smalltalk zu verstehen. Das ist zugleich eine kleine Reise in die Natur.

## 2 Objekte

Wenn wir uns in der Natur umschauen, so sind die aktiven Elemente natürlich *Lebewesen*. Ein Lebewesen reagiert auf Reize, die es von seiner Umwelt erhält. Nicht jedes Lebewesen kann ein und denselben Reiz verarbeiten. So ist zum Beispiel für Fledermäuse die Wahrnehmung eines Ultraschallechos von einem Insekt lebensnotwendig für die Nahrungssuche. Ein Rotkehlchen kann zwar ebenfalls Ultraschall wahrnehmen. Es kann jedoch mit dem Ultraschallecho des Insektes nicht viel anfangen. In der Fledermaus dagegen läuft beim Empfang des Echos sozusagen ein »Programm« ab, mit dem sie genau die Größe, die Art des Insekts und seine Entfernung ermitteln kann. Wenn das »Programm« abgelaufen ist, erfolgt als Antwort eine Reaktion der Fledermaus. Zum Beispiel in der Änderung der Flugrichtung.

Es gehört nun einige Genialität dazu, die einen Alan Kay, einen Dan Ingalls und eine Adele Goldberg auf die Idee bringt, die Art und Weise wie die Natur funktioniert, auf den Computer zu übertragen. Die große Leistung besteht vor allem darin, alle Beobachtungen auf einige wenige grundlegende Prinzipien zurückzuführen:

**Objekte:** Den Lebewesen in der Natur entsprechen *Datensätze*. Das können Zahlen, einzelne Zeichen, Zeichenketten oder aber auch grafische Elemente wie etwa ein Fenster oder eine Zeichnung sein. Diese Datensätze heißen allgemein *Objekte*.

**Nachrichten:** Lebewesen reagieren auf bestimmte *Reize*. Diesen Reizen entsprechen bei den *Objekten* so genannte *Nachrichten*. In der Fachsprache heißen sie natürlich *messages*. Wie solche Nachrichten aussehen, werden wir in Kürze lernen.

**Methoden:** Auf welche Reize ein Lebewesen reagieren kann, ist in seinem Inneren durch die biologischen »Programme« festgelegt, die zu diesen Reizen passen. Entsprechend verfügt ein *Objekt* über einzelne Programme, *Methoden* genannt, über die festgelegt ist, auf welche Nachrichten das Objekt reagieren kann.

Klären wir nun die Frage nach der Art und Weise wie Nachrichten an ein Objekt gesendet werden. Denken wir dabei an eine Familie, die im Mai Ihr erstes Frühstück im Freien wagt. Es ist noch sehr zeitig am Morgen – und warum die »Unterhaltung« bei Tisch ein *wirklicher* Smalltalk ist, können wir uns leicht vorstellen. Marek und Vater Vaclav haben mit der unchristlichen Stunde die meisten Probleme. Mutter Lida und Schwesterchen Sveta sind dagegen sehr viel munterer. Marek folgt ganz der Regel »Bewegung hält jung. Ich bin

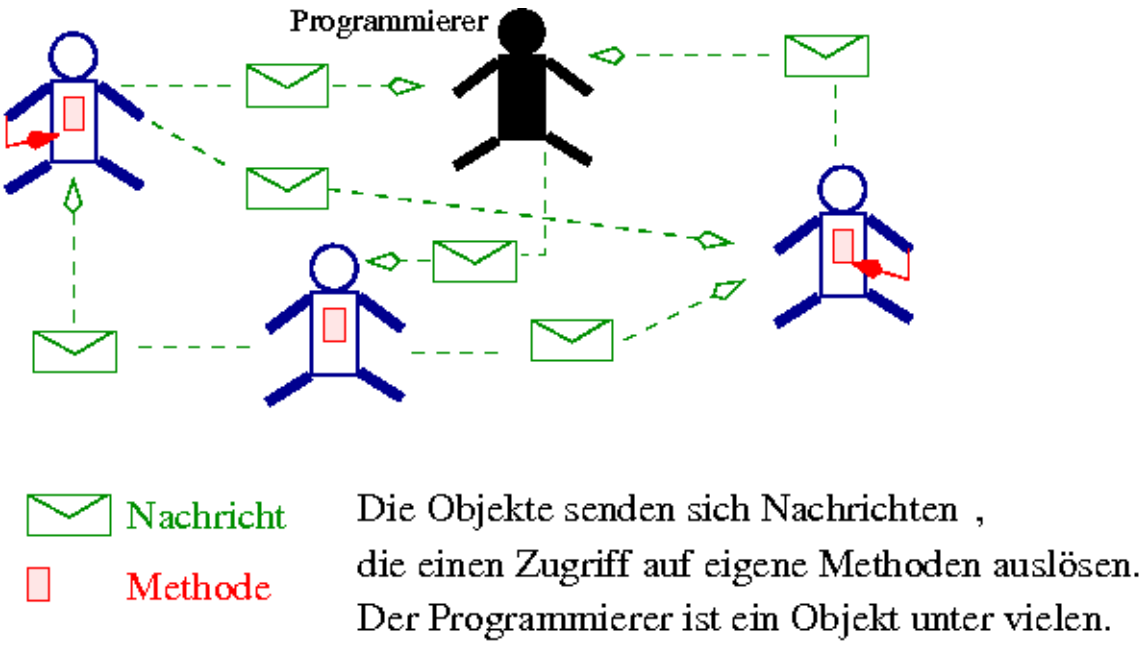


Abbildung 2: Objektorientiertes Programmieren

jung. Also brauche ich mich nicht zu bewegen.« In einem Moment, in dem Sveta still an ihrem Saft nippt, sendet ihr Marek eine Nachricht. »Sveta, gib mir bitte die Butter und den Tee.« In der korrekten Syntax von Smalltalk lautet dies:

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

Schauen wir uns zunächst an, wie die Nachricht gebaut ist. An der ersten Stelle steht immer der Name des angesprochenen Objekts, des *receivers*. Das ist in diesem Falle *sveta*. Das Wort *sveta* ist eine Zeichenkette, die einem Objekt *zugeordnet* ist. Nur über diesen Namen kann die wirkliche kleine Schwester angesprochen werden. Ein solcher Name ist eine *Variable*. Warum *sveta* klein geschrieben wird, hat natürlich seinen Grund. Wir erklären ihn im entsprechenden Kapitel über die Smalltalk-Programmierung genauer. In der Nachricht gibt es weitere Objekte.

Diese stehen jeweils hinter den Doppelpunkten. Es sind also: marek, butter und tee. Diese Objekte werden in der Nachricht als *Argumente* übergeben. Natürlich sind die drei genannten Bezeichnungen ebenfalls *Namen* für real existierende Objekte. Der eigentliche, wesentliche Bestandteil der Nachricht ist der *Nachrichtenselektor*, der in diesem Fall *gib:bitte:und:* lautet. Diese Wortkette heißt deswegen *Selektor*, weil sie aus sätlichen »Methoden«, über die *sveta* im Inneren verfügt, die passende Methode selektiert, also herausfiltert. Im Sinne von Smalltalk existiert im Inneren von *sveta* ein Programm mit der Bezeichnung:

```
gib:erstesArgument bitte:zweitesArgument und:drittesArgument
```

Wie dieses Programm *genau* aussieht, wie es also in *sveta* implementiert ist, ja *das* ist die Kunst der Programmierung, die Du erlernen wirst. Da die Methode *gib:bitte:und:* auch für andere Objekte als *marek, butter* und *tee* funktionieren soll, muss das eigentliche Programm der Methode natürlich mit Ersatz-Namen arbeiten, die hier *erstesArgument, zweitesArgument* und *drittesArgument* heißen.

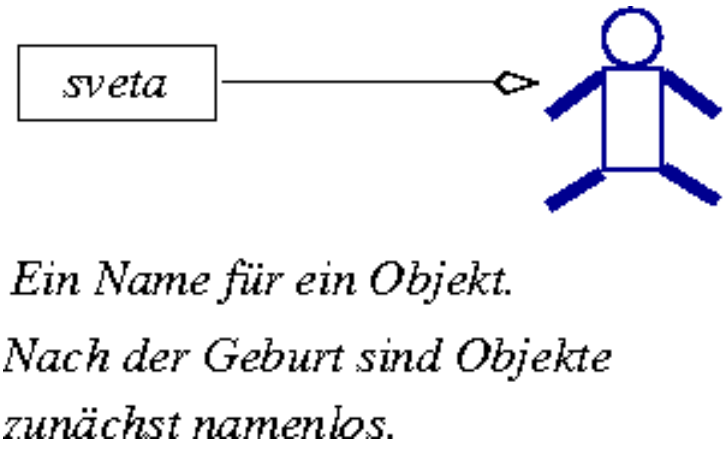


Abbildung 3: Ein Name wird an ein Objekt gebunden.

Jetzt kennst Du den feinen Unterschied zwischen einer *message* und einer *method*. Fassen wir noch einmal zusammen: Eine *message* besteht immer aus drei Teilen: dem Namen des *receivers*, dem *Nachrichtenselektor* mit eventuellen *Argumenten* und dem abschließenden Punkt. Dieser Punkt ist nur notwendig, falls eine weitere Nachricht folgt. Wenn der *receiver* die Nachricht versteht, verfügt er über eine *method*, deren Bezeichnung genau so lautet wie der Nachrichtenselektor, mit dem Unterschied, dass die Namen eventuell übergebener Objekte durch Platzhalter ersetzt sind.

Wie *sveta* auf die Nachricht reagieren wird, hängt sicherlich auch vom Ton der »Unterhaltung« ab. Im einfachsten Fall würde sie die Butter und den Tee reichen. Im Sinne von Smalltalk ist dann das zurückgegebene Objekt eine sogenannte *Kollektion* (Sammlung) von Objekten. In der Smalltalk-Syntax könnte dies zum Beispiel durch

```
#(butter tee)
```

dargestellt werden. *#(butter tee)* ist *kein* Name für dieses Objekt, sondern das Objekt *selbst*. Es gehört zu den wenigen Objekten, die in Smalltalk direkt durch so ein Literal angegeben werden können. Es gibt insgesamt fünf solcher primitiven Objektarten. Grundsätzlich sollte jedes Objekt, das irgendwo auftaucht, sofort mit einem Namen versehen werden, weil sonst der Zugriff darauf verloren geht. In der realen Welt mutet es natürlich seltsam an, wenn Marek die »Kollektion« aus Butter und Tee mit einem Namen versieht. Er könnte vielleicht sehr unfreundlich sein und ausrufen: »Was ist denn das für ein *Kram*?« Nun, hier hat eine so genannte Namenszuweisung stattgefunden, die in Smalltalk so aussieht:

```
kram ←#(butter tee).
```

Gleichzeitig ist es aber wieder eine Nachricht an Sveta:

```
sveta wasSoll:marek mit:kram.
```

Sveta »ruft« jetzt die Methode *wasSoll:senderObjekt mit:objekt* auf, die – wer kann es Ihr verdenken – stets so programmiert ist, dass sie dem Sender stets dieselbe Nachricht zukommen lässt:

```
marek beschwereDichNicht.
```

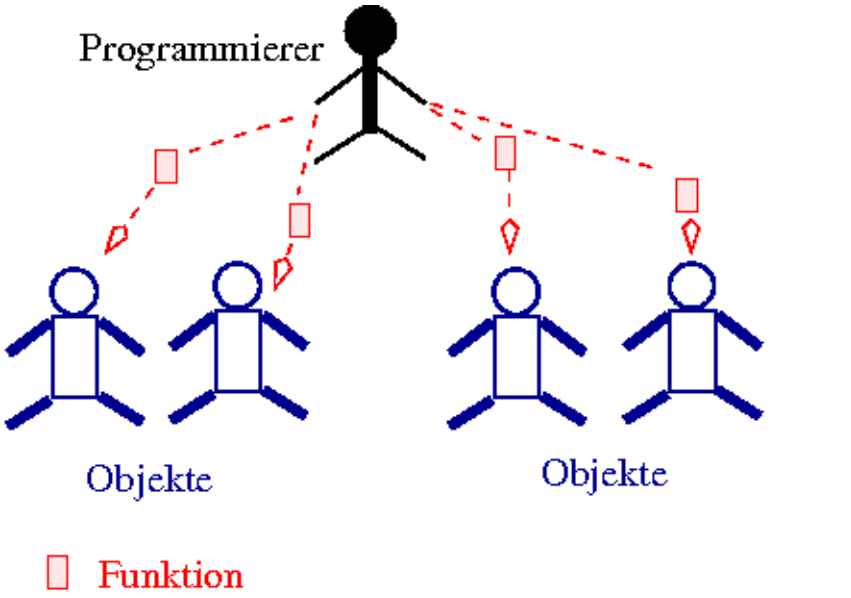


Abbildung 4: Prozedurales Programmieren

Wie Du hier siehst, muss ein Nachrichtenselektor, wie hier *beschwereDichNicht*, nicht immer Argumente besitzen. Diese Nachricht ist allerdings nicht die *Antwort* auf die Nachricht *sveta wasSoll:marek mit:kram.* So merkwürdig es ist: Die Antwort ist *immer* ein Objekt, das zurückgegeben wird. Wenn kein Objekt zurückgegeben werden soll, wie in diesem Fall, so ist der Empfänger *selbst* die Antwort. Diese seltsame Regel stellt sicher, dass nach einer Nachricht für das Folgende ein Ansprechpartner zur Verfügung steht. Wenn *wirklich* nichts zurückgegeben wird, so hat das Objekt die Nachricht nicht verstanden.

Nun machen wir auf unserer Reise durch die Natur eine kleine Pause. Wir setzen die Reise auf der Seite 11 fort. Bis dahin gibt es für die Hungrigsten unter meinen Lesern zwei Abschnitte, in denen etwas genauer auf den Sinn der objektorientierten Programmierung und auf die Frage, in wieweit C++ und Java diesem Konzept genügen, eingegangen wird. Du musst diese beiden Abschnitte nicht unbedingt lesen, um die Reise fortzusetzen. Aber vielleicht ist Einiges doch für Dich von Interesse.

## 3 Prozedurales und objektorientiertes Denken*

Alles was wir eben besprochen haben, ist noch einmal in der Abbildung 2 auf Seite 4 grafisch dargestellt. Ja, die Objekte sprechen auch mit dem Programmierer. Er ist ein Teil des Ganzen. Die ältere, ganz andere Denkweise geht nun von der Vorstellung aus, dass sich die Objekte nicht selbst »bewegen« können. Der Programmierer verwendet den Computer, um etwas *mit* den Objekten auszuführen. Die Objekte sind also passive Gebilde, die sich die Anwendung gewisser *Funktionen* oder *Prozeduren* gefallen lassen müssen (siehe Abbildung 4). Kein Lebewesen in der Natur würde so etwas zulassen. Bei unserem Beispiel würde in einer prozeduralen Sprache das erste Problem durch

```
gibBitteUnd(sveta,marek,butter,tee)
```

gelöst. Dies ist nicht etwa nur eine andere Schreibweise, sondern ein völlig anderer Gedanke! Es gibt hier eine (externe) Funktion mit dem Namen *gibBitteUnd(objekt1, objekt2, objekt3, objekt4)*. Diese Funktion wirkt bei dem Aufruf (der von dem Programmierer stammt!) *auf* die Objekte *sveta, marek, butter* und *tee*. Sie soll dafür sorgen, dass die »Marionette« Sveta der anderen »Marionette« Marek die Objekte Butter und Tee überreicht. Die Funktion oder die Prozedur[1] greift also *von außen* auf die Objekte zu. Eine gewisse Gefahr besteht darin, dass dieselbe Funktion auf Objekte angewendet wird, für die sie nicht gedacht ist. Zum Beispiel würde *gibBitteUnd(sveta,natrium,wasser,benzin)* zu einer einer »größeren Fehlfunktion« führen.

Und doch ist uns das Anwenden von »Werkzeugen« *auf* Objekte scheinbar vertrauter. So rechnen wir zum Beispiel *mit* Zahlen und lassen nicht die Zahlen selbst rechnen. Bei der Addition $3 + 4$ wird eine Funktion $+$ *auf* die Objekte 3 und 4 angewendet. Dabei ergibt sich das neue Objekt 7. Wird dieselbe Funktion $+$ versehentlich auf die Zeichenketten $'3'$ und $'4'$ angewendet, so wird $'3' +' 4'$ wahrscheinlich zu einem Fehler führen, da es sich hier nicht um Zahlen handelt.

Die Vorstellung, Objekte als Lebewesen aufzufassen, ist bei Zahlen auf den ersten Blick kurios. Hier heißt $3 + 4$, dass dem Objekt 3 die Nachricht $+4$ gesendet wird. Der Nachrichtenselektor ist das Zeichen $+$ und das Argument ist das Objekt 4. Jetzt ruft die Zahl 3 im Inneren die entsprechende Methode auf, mit der sie den eigenen Wert um 4 erhöht und gibt das Objekt 7 zurück. Das ist merkwürdig, ja. Und wer Smalltalk-Anweisungen laut liest, wird nicht ohne Grund von seiner Umgebung für seltsam gehalten. Aber nun kommt das Entscheidende: $'3' +' 4'$ führt in Smalltalk *nicht* zu einem Fehler. Denn hier wird nicht dem Zahlobjekt 3, sondern einem ganz anderen Objekt $'3'$, die Nachricht $+'4'$ gesendet. Jetzt ruft $'3'$ die zur Nachricht passende Methode auf, die *völlig verschieden* von der entsprechenden Methode des Zahlobjektes 3 ist. Die Methode *könnte* zum Beispiel die beiden Zeichenketten aneinanderhängen, so dass sich $'34'$ ergibt[2]. Obwohl in beiden Fällen *derselbe* Name für den Nachrichtenselektor verwendet wird, bedeuten beide Nachrichten etwas *Verschiedenes*. Die entsprechenden Methoden sind für Zahlen und Strings unterschiedlich implementiert. Das hat nicht nur den riesigen Vorteil, damit die Anwendung einer »falschen« Funktion auf die falschen Objekte zu verhindern. Es gibt noch einen anderen Grund: Nehmen wir uns wieder die schon etwas strapazierten Namen *sveta* und *marek* her, so ist

```
marek + sveta.
```

eine Anweisung, deren Interpretation der Phantasie Tor und Tür öffnet. Ja, das meine ich durchaus ernst. Zeigen die Namen zum Beispiel auf *Zahlobjekte*, so wird von marek die Methode der gewöhnlichen Addition aufgerufen[3]. Sind *marek* und *sveta* aber Zeichenketten, so wird *marek seine Interpretation* von $+$ zur Addition von Zeichenketten aufrufen. Sind *marek* und *sveta* wirkliche Lebewesen, so ist die Implementation der entsprechenden Methode sicherlich nicht mit den gleichlautenden Methoden für Zahlen oder Zeichenketten zu vergleichen. Der große Vorteil besteht nun darin, dass bei dem Verwenden der Anweisung

[1] Der Unterschied ist für uns nicht wichtig. Aber eine Prozedur ist eine Funktion, die am Ende kein Objekt zurückgibt.

[2] Tatsächlich würde ein solches Objekt als Antwort auf die Nachricht $'3',' 4'$ entstehen. $'3' +' 4'$ wandelt dagegen jedes Objekt in eine Zahl um, sendet dann an das entstandene Objekt 3 die Nachricht $+4$ und wandelt das sich ergebende Objekt 7 wieder in einen String $'7'$ um.

[3] Wobei sich die Methoden in ihrem Ablauf auch nach der Art der Zahl untrscheiden

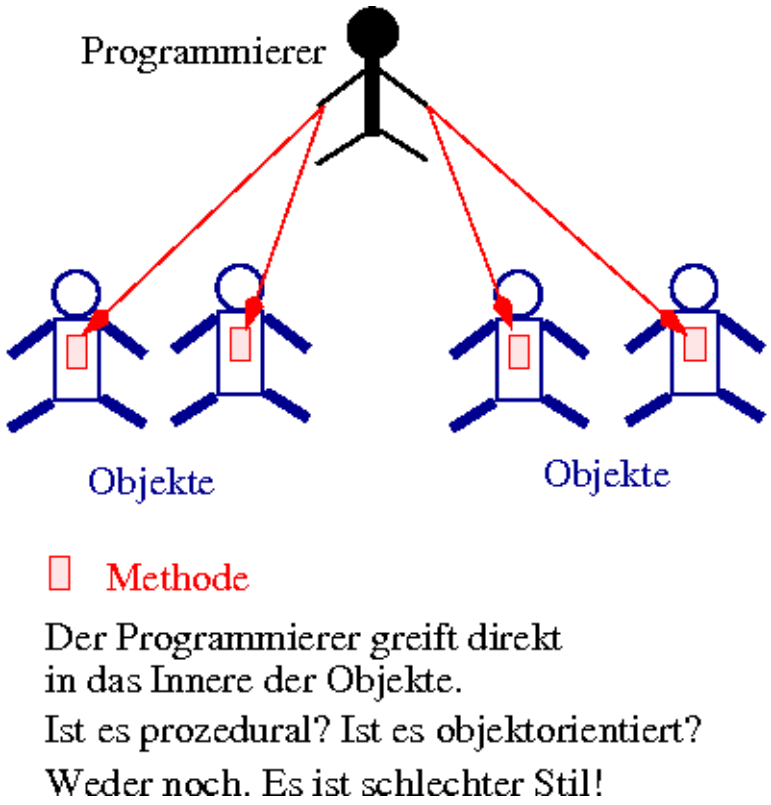


Abbildung 5: Was passiert, wenn eine Sprache nicht rein objektorientiert ist.

*marek+sveta*. völlig egal ist, welche Objekte durch die Namen repräsentiert werden. Welche Interpretation der Nachrichtenselektor + erfährt, hängt von der aktuellen Zuweisung der Namen ab. Der richtige Aufruf ist also immer von der gerade aktuellen Aktion abhängig. Diese Art der Ausführung von Anweisungen heißt *dynamische Bindung* und das Konzept heißt *Polymorphismus* (Vielgestaltigkeit).

Im Unterschied dazu muss in prozeduralen Sprachen sichergestellt sein, dass *marek* und *sveta* Objekte eines ganz bestimmten Typs sind. Nur so kann der Rechner gewährleisten, dass er die richtige Funktion aussucht. Die Festlegung dieser Entscheidung wird am Beginn des Programms[4] durch eine *Typendeklaration* festgelegt, wie zum Beispiel

```
var sveta,marek: INTEGER
```

so dass beide Namen für *ganzzahlige* Objekte reserviert sind. Sie können also nicht an anderer Stelle für Zeichenketten verwendet werden[5]. Mit anderen Worten: rein prozedurale Sprachen müssen alles von vornherein festlegen und arbeiten mit einer *statischen Bindung*.

## 4 Hybridsprachen*

Bevor wir gleich ganz in die Natur abtauchen: Keine Programmiersprache kann den Anspruch erheben, die allein seligmachende zu sein. Es ist im Prinzip möglich, jedes Problem

[4] Bei C++ auch unmittelbar vor der ersten Verwendung.

[5] Es gibt allerdings in vielen prozeduralen Sprachen das umständliche Konstrukt der Typumwandlung. Sie ist der Flexibiliät des Polymorphismus aber unterlegen.

mit jeder beliebigen Sprache zu lösen. Nur unterschiedlich gut. Doch gibt es seit den siebziger Jahren immer noch heiße Diskussionen, welche Sprache sich für den Anfänger am besten eignet. Diese Diskussionen sind durchaus sehr sinnvoll und müssen stattfinden. Es eignet sich nicht *jede* Sprache als erste Plattform. Und es ist auch nicht ratsam, sich auf *eine einzige* festzulegen. Denn es gibt inzwischen ein ganzes Arsenal guter Anfängersprachen.

Leider entwickeln sich die notwendigen Diskussionen oft zu einem Nebeneinander gewisser Glaubenksbekenntnisse, die gemeinsam geführten didaktischen Überlegungen den Weg versperren. Es ist klar: Jede Sprache hat ihre »Fans«. Das muss auch so sein. Selbst diejenigen Programmierer, die mehrere Sprachen beherrschen, sind genau genommen in einer Sprache am liebsten »zu Hause«. Wer aber im wahrsten Sinne des Wortes »zu Hause« ist, führt keinen Finger über jene Leuchttafel, auf der gewisse »Einschaltquoten« verzeichnet sind. Keine Sprache ist allein deswegen vorzuziehen, nur weil »sie eben Praxis« ist[6]. Ein echtes Argument ist dagegen die Konsequenz, mit der eine Sprache ein Konzept verfolgt. Und gerade in diesem Punkt haben so manche populären Sprachen keine guten Karten, weil sie sich – aus welchen Gründen auch immer – als Hybridsprachen präsentieren.

Warum haben sich C++ und Java aber so durchgesetzt? Nun, dazu müssen wir ein wenig die Geschichte bemühen. Smalltalk wurde von Beginn an mit einer grafischen Oberfläche ausgeliefert, wie Du sie bei Squeak vorfindest. Diese Umgebung ist nicht nur ein mitgeliefertes Werkzeug, sondern Bestandteil der Sprache, da sie die Welt darstellt, in der die Objekte leben. Smalltalk entstand aber zu einer Zeit als es nur textorientierte Betriebssysteme gab und der Arbeitsspeicher aus heutiger Sicht lächerlich klein war. Eine grafische Oberfläche verlangt jedoch – wenigstens für damalige Verhältnisse – sehr viel Speicher. Somit war Smalltalk noch Anfang der achtziger Jahre eine ziemlich teure Angelegenheit. Doch entwickelte sich eine kleine, stabile Fangemeinde. Und es ging der Spruch: »Smalltalk macht einfach Spass« in den achtziger Jahren herum. Viele aber konnten sich auf diesen teuren Spass nicht einlassen und kannten Smalltalk-Syteme nur dem Namen nach. Da also der Programmierer sich nur mit Mühe der Sprache zuwenden konnte, kehrte sich der Spieß um und Smalltalk kam – als grafische Betriebssysteme verkleidet – zu *allen* Menschen. Seien sie nun Anwender oder Programmierer. Erst durch Apple und Atari, dann schließlich auch auf dem PC. Verschiebbare Fenster, anklickbare Knöpfe und – nicht zuletzt die Maus. Alles das hat sich durch Smalltalk verbreitet. Die meisten Menschen aber kamen selbst mit der Programmiersprache nicht in Berührung, obwohl von Anbeginn für die Entwickler um Alan Kay die Frage »Was können Kinder mit einem Computer anfangen?« zentrales Anliegen war. Tatsächlich erreichte in der Schule »die Kinder« Smalltalk zunächst nur in Form von Seymour Paperts Turtle-Grafik, die sich zwar noch vor Smalltalk entwickelte, dann aber wesentlicher Bestandteil der Sprache wurde.

Die objektorientierten Ideen waren so überzeugend, dass Bjarne Stoustrup drei Jahre nach dem offiziellen Standard Smalltalk-80 die Sprache C++ aus der Sprache C entwickelte. So konnten objektorientierte Gedanken *unabhängig* von einer grafischen Umgebung realisiert werden. Die Bezeichnung C++ heißt »etwas mehr als C«, also ein »C mit Objekten«, wie die Sprache zunächst auch hieß[7]. Doch – aus welchem Gurnde auch immer – geriet C++ zu einer Sprache, die beide Programmierstile (prozedural und objektorientiert) ermöglichen sollte. Das aber ist eigentlich weder möglich, noch wünschenswert. Neben einigen »Erleichterungen« im Vergleich zu C hält die Sprache an jenen Typendeklarationen fest, die auch im didaktischen Bereich von den Anhängern der einst so beliebten Lehrsprache PASCAL immer noch sehr geschätzt werden. Wie wir vorhin sahen, widerspricht dies ganz klar ob-

[6] Übrigens gibt es in der Informatik »die Praxis« genau so wenig wie »das Problem«.

[7] Tatsächlich entwickelte sich C++ zu einer eigenständigen Sprache und sollte auch unabhängig von C gelehrt werden.

jektorientierter Denkweise (Polymorphismus).

Noch problematischer aber ist die Syntax. Wenn in C++ »rein objektorientiert« gearbeitet wird, so würde unser Beispiel

```
sveta.gibBitteUnd(marek,butter,tee)
```

lauten. Was geschieht hier? Versuche mal einen C++ Programmierer mit dieser Frage auf die Probe zu stellen. Welche Ansicht ist die richtige?

**Falsch:** »Es wird« die in *sveta* implementierte Methode *gibBitteUnd* mit den Argumenten *marek, butter, tee* aufgerufen. Warum ist dies nicht objektorientiert gedacht? Nun, weil in einer echten objektorientierten Sprach eine Methode *niemals* von irgend jemand Anderem aufgerufen wird als vom Objekt selbst. Die Schreibweise legt die Ansicht nahe, dass hier eine Funktion auf drei Objekte angewendet wird (prozedural!), die lediglich *»sveta* gehört« (scheinbar objektorientiert!). Es wird von außen in das Objekt hineingegriffen. Wenigstens gedanklich.

**Richtig:** *sveta* wird die Nachricht *sveta.gibBitteUnd(marek,butter,tee)* übermittelt, wobei die Objekte *marek, butter, tee* übergeben werden. Erst dann ruft *sveta* eine passende Methode auf. Diese Sicht der Dinge legt die Schreibweise überhaupt nicht nahe.

Nun, jetzt könnten wir sagen: Es muss die Schreibweise nur »richtig interpretiert« werden. Gut. Aber tatsächlich läßt C++ den Zugriff von außen *in* ein Objekt hinein zu. Reservierte Worte wie *private, public* weichen das Konzept auf. In einer *wirklich* objektorientierten Sprache ist das Innere eines Objekts *immer* private. Diese *Kapselung* ist ja gerade der Sinn der Sache. Außerdem werden in C++ nicht alle Dinge als Objekte aufgefaßt. Zahlen zum Beispiel nicht. Der Kompormißversuch, prozedural geschulten Programmiererhirnen, nicht *ganz* das prozedurale Denken abzugewöhnen, sondern »nur soviel wie möglich« Objektorientiertes zuzumuten, hat zwar zu einer leistungsfähigen Sprache geführt, die aber mit den Worten einer Eliška Krásnohorská als »bärenhaft« zu bezeichnen ist[8]. Der größte Vorteil von C++ ist die Erstellung schnell ausführbarer echter Programme, die unabhängig von einer festen Umgebung laufen. In Smalltalk gibt es sozusagen keine Programme, sondern reine Objekte, die sich *in* der grafischen Umgebung aufhalten.

Java ist der – gelungene – Versuch, eine Sprache zu schaffen, die nur ein Programmieren in dem Stil zuläßt, wie es unter C++ immer geschehen sollte. Obwohl die Sprache intern extrem stark Smalltalk verpflichtet ist[9], haben sich die Entwickler aus reinen Marketinggründen für die Syntax von C++ entschieden. So verständlich dieser Schritt aus wirtschaftlichen Gründen erscheint: Als Lehrsprache sollte Java mehr als bedenklich erscheinen. Allein ein Konstrukt wie *public static void main* sollte zu denken geben. Zugriffsbeschränker wie *public* und *protected*, sowie jede Art der Typendeklaration haben in einer *streng objektorientierten* Sprache nichts verloren. Was wir bei C++ über Methoden gesagt haben, gilt für Java auch. Ebenso die Inkonsequenz bei der Auffassung von Objekten.

Als Vorteil von Java erscheint vor allem die Ausführbarkeit des Codes in Form von Applets über das Internet. Applets sind sozusagen »unfertige«, aber in einem Zwischencode

[8] So die unglückliche Aussage der Schriftstellerin und Sprachforscherin, nachdem sie Bedřich Smetana das Libretto zur Oper Čertová stěna übergab. Nach Wunsch Smetanas sollte diese – seine letzte – Oper komisch, romantisch und sagenhaft zugleich sein. Also eine Art musikalisches C++ unter den großartigen Bühnenwerken des Meisters.

[9] Eigentlich eher dem Smalltalk-Abkömmling *Self*, wie wir noch sehen werden.

vorliegende, Programme, die durch eine in einem Browser eingebaute »virtual machine« (VM) interpretiert werden. Manche Autoren schreiben die Erfindung der Virtual Machine Java zu. Die Wahrheit ist, dass es die Virtual Machine von Anbeginn an Bestandteil von Smalltalk ist und war. Die grafische Umgebung von Squeak ist in Wirklichkeit auch eine Art Zwischencode, die von der VM interpretiert wird. Sozusagen ist das, was Du bei Squeak unmittelbar siehst, eine Art riesiges Applet. Aber auch Smalltalk hat nicht die VM erfunden. Sie gab es schon wesentlich früher.

Java ist eine gute Sprache. Vielleicht sogar eine sehr gute. Die unkritische Euphorie, die dieser Sprache zuteil wird, nach dem Motto »Es gibt nichts Bessers«, ist aber mehr als fragwürdig. Python ist aus didaktischer Sicht – obwohl auch nicht streng objektorientiert – sehr viel besser, da zumindest die unsinnige Typendeklaration fehlt und der Code extrem gut lesbar ist. Die Syntax selbst gehört aber wieder in den Dunstkreis von C++. Ruby ist – was die Objektorientierung anbelangt – sicherlich noch konsequenter. Da Java (und auch C++) unabhängig von grafischen Umgebungen entwickelt wurden, können grafische Arbeiten nur mit Hilfe leistungsfähiger Bibliotheken durchgeführt werden, die zusätzlich eingebunden werden müssen. Das ist auch bei Python nicht anders, obgleich mit Tk eine sehr einfach zugängliche Bibliothek zur Verfügung steht.

Bei Smalltalk ist der Programmierer von Anbeginn von grafischen Elementen umgeben. Nichts muss extra hinzugebunden werden, denn alles ist da. Viel wesentlicher ist jedoch, dass es keine der genannten Sprachen in der Konsequenz des Konzeptes mit Smalltalk aufnehmen kann. Das erstreckt sich auch auf die Syntax, die sich sehr stark an der Alltagssprache orientiert, wie Du an *sveta gib:marek bitte:butter und:tee.* erkennst. Obwohl Smalltalk nicht die erste objektorientierte Sprache war[10], gibt es zur Zeit keine Sprache, mit der Du besser das objektorientierte Programmieren lernen kannst – und es gibt keine andere, die so eng mit der Natur verbunden ist. Das sagten wir schon früher. Und nun wird es Zeit, dass wir die trockene Diskussion verlassen und endgültig die Natur abtauchen.

# 5 Klassen

Wir müssen nun noch die Frage klären, wie die Lebewesen, die Objekte zu ihren Eigenschaften und Methoden kommen. Wir erinnern uns. Eine Fledermaus benutzt Ultraschall nicht nur zur Orientierung, sondern auch zur Nahrungssuche. Wenn ein Insekt das Echo zurückwirft, kann die Fledermaus diese Nachricht so genau auswerten, dass sie Größe, Entfernung und Art des Insekts feststellen kann. Auch ein Rotkehlchen kann Ultraschall wahrnehmen. Aber es kann die Nachricht eines Ultraschallechos von einem Insekt nicht auswerten. Warum kann die Fledermaus das? Nun, die Antwort klingt fast ein wenig frech: Weil sie eine *Fledermaus* ist. Warum kann ein Rotkehlchen die Nachricht nicht auswerten? Nun, weil es eben ein Rotkehlchen und *keine* Fledermaus ist. Das Rotkehlchen kann sich natürlich auch nicht sagen: »Heute bin ich eine Fledermaus.«, und sich selbst eine Methode einbauen, mit der eine Auswertung des Echos möglich wird.

Damit wird Dir vielleicht klar, dass es nicht die Objekte selbst sind, die ihre eigenen Fähigkeiten festlegen. Es liegt vielmehr »in ihrer Natur« was sie können. Aber es muss ja ein Etwas geben, das diese Dinge festlegt. Dieses Etwas, benennt der Biologe mit Worten wie *Familie, Ordnung* oder *Klasse*. Statt wie der Biologe mehrere, verschiedene Begriffe zu verwenden, spricht Smalltalk *nur* von einer *Klasse*. Die Klasse enthält nun sowohl den *Bauplan* als auch die *Methoden*, über die ein konkretes Objekt verfügen soll. So schreibt die

[10] Das war die norwengische Sprache Simula, die in den sechziger Jahren zur Simulation biologischer Prozesse geschaffen wurde.

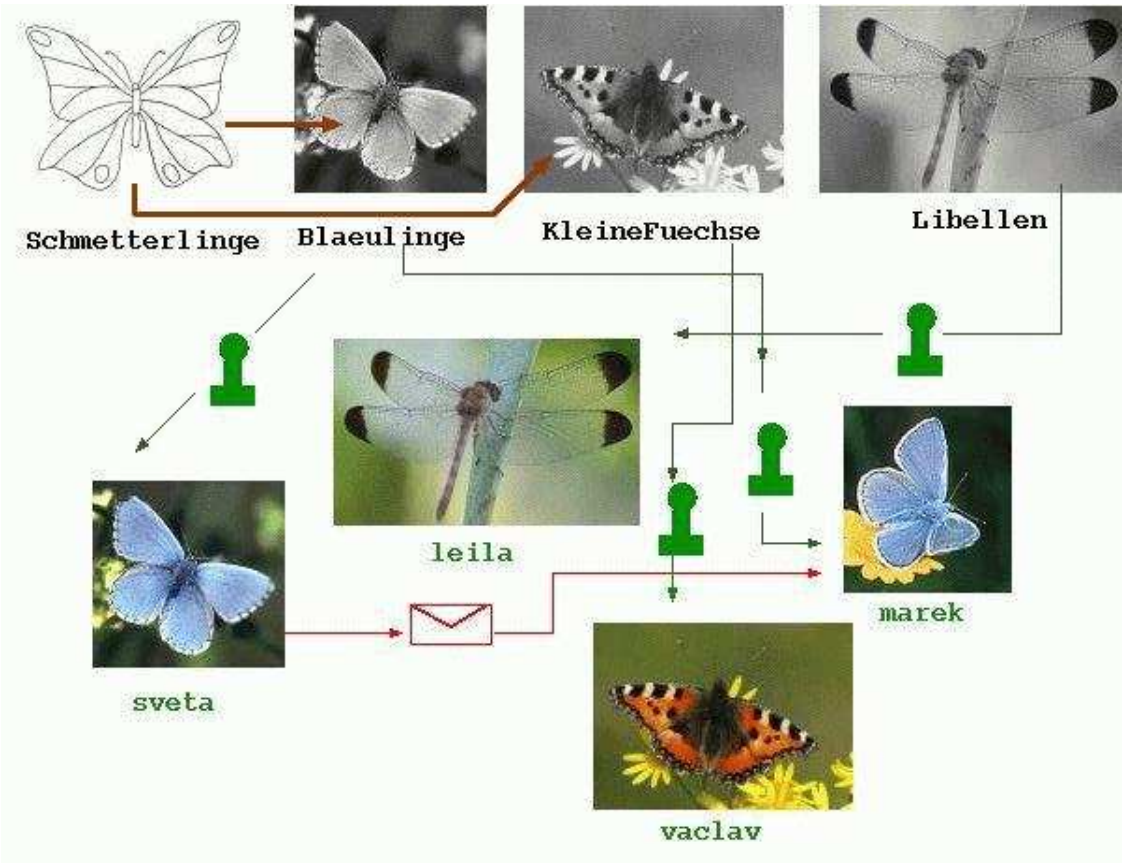


Abbildung 6: Eine Sicht der Natur

Klasse der Fledermäuse[11] zum Beispiel vor, dass eine Fledermaus Flügel und Ohren besitzt. Wie lang die Flügel, wie lang die Ohren bei einem konkreten Objekt sind, kann in vorgegebenen Grenzen variieren. Die Klasse legt also die Eigenschaften des Objekts fest, nicht aber dessen konkrete Ausprägung. Sehr wohl wird aber die Fähigkeit, Ultraschall auszuwerten durch eine Methode festgelegt, über die alle Objekte der Klasse verfügen.

In der Abbildung 6 sind die Klassen durch Schwarzweiß Abbildungen dargestellt. Die konkreten Objekte sind farbige Individuen. Natürlich ist ein Bläuling in der Regel blau. Das wird bereits festgelegt. Aber wie das Blau nun tatsächlich aussieht, kann variieren. Von den Klassen gehen Pfeile aus, die mit einem Stempel versehen sind. Bei der Geburt wird also ein Objekt erzeugt, das wie der Abdruck eines Stempels in die Welt gestzt wird. Aus diesem Grund wird ein Objekt auch *Instanz* einer Klasse genannt. Die Geburt selbst heißt *Instanzierung*.

Wie nun geht bei Smalltalk eine Geburt vor sich? In den meisten Fällen (aber nicht in allen) geschieht das dadurch, dass der Klasse eine Nachricht mit dem Selektor *new* zugesendet wird. So erzeugt

```
Blaeulinge new.
```

einen neuen Bläuling. Aber dieser ist nach der Geburt nicht mehr ansprechbar, wenn wir ihm nicht gleich einen Namen geben. Objekte, auf die kein Name verweist, werden irgend-

[11] In der Biologie ist das zwar keine Klasse, aber wir sagtenja eben, dass wir diesen Begriff für alle Oberbegriffe der Biologie verwenden.

wann von Smalltalks aautomatischer »Müllabfuhr« (*garbage collection*) beseitigt. Um das Neugeborene zu »retten«, muss also gleich die Taufe erfolgen:

```
sveta←Blaeulinge new.
```

setzt den Bläuling *sveta* in die Squeak-Welt. *sveta* enthält dabei alle Methoden und Eigenschaften, die durch die Klasse *Blaeulinge* festgelegt werden. Vielleicht fragst Du Dich, warum die Namen der Klassen groß geschrieben werden. Nun, groß geschriebene Namen sind von jeder Stelle der Squeak-Welt aus ansprechbar. Es sind *globale Variablen*. Klassen *müssen* global sein. Instanzen wie *sveta können* global sein, wenn ihr Name groß geschrieben wird. Das ist aber nur anzuraten, wenn eine universelle Präsenz wirklich notwendig ist. Objekte, deren Namen klein geschrieben wird, existieren nur in derjenigen Umgebung, in der sie geboren werden. Diese Umgebung werden wir bald als Workspace kennenlernen. Wir sehen in der Abbildung, dass sich *sveta* und *marek* wieder einmal Nachrichten senden (roter Brief). Jetzt als Bläulinge. Und die Nachrichten werden sicherlich chemischer Natur sein. Kümmern wir uns noch ein wenig mehr um die Klassen.

Wieso versteht eine Klasse eine Nachricht? Ist sie selbst ein Objekt? Ja, genau so ist es! Und das unterscheidet abermals C++ und Java von Smalltalk. In Smalltalk ist wirklich jedes Element ein Objekt. Woher kann eine Klasse auf die Nachricht *new* antworten? Kann eine Instanz der Klasse auch auf *new* antworten und weitere Instanzen in die Welt setzen? Nein. Die Methode *new* ist eine Klassenmethode, keine Methode der Instanzen. Tatsächlich ist eine Klasse Instanz einer so genannten *Metaklasse*, was wir aber hier noch nicht näher verfolgen werden. Die Methode *new* hat die Klasse tatsächlich von einer anderen Klasse *geerbt*. So ist es zum Beispiel denkbar, dass eines Tages aus der Klasse der Bläulinge eine weitere Klasse entsteht, die *neben* den Eigenschaften und Methoden der gewöhnlichen Bläulinge ihren Instanzen auch noch die Fähigkeit verleiht, die Farbe aus Tarnungsgründen dem Untegrund der Umgebung anzupassen. Diese Unterklasse (*Subclass*) erbt von ihrer Oberklasse (*Superclass*) alle Eigenschaften und Methoden und reichert diesen Pool um gegebenenfalls weitere Eigenschaften und Methoden an oder verändert diese.

Jede Klasse hat die Fähigkeit Unterklassen zu erzeugen. Dazu muss eine recht komplizierte Nachricht an die Oberklasse geschickt werden, die durch den Selektor

```
subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
```

gegeben ist. Natürlich gibt es bei Smalltalk ein Werkzeug, das den Umgang mit einem so langen Nachrichtenselektor vereinfacht. Im Grunde steckt nicht viel dahinter:

**subclass:** erfordert als Argument den Namen der neuen Klasse (zwingend)

**instanceVariableNames:** erfordert als Argument die Namen der geplanten Eigenschaften für die Instanzen; hier müssen nur die *zusätzlichen* Eigenschaften aufgeführt werden, da alle Eigenschaften der Oberklasse vererbt werden. Sind keine zusätzlichen Eigenschaften erforderlich wird eine leere Zeichenkette als Objekt übergeben.

**classVariableNames:** bezeichnet Eigenschaften, die bei allen Instanzen konstant festliegen sollen (ist meistens nicht notwendig)

**poolDictionaries:** bezeichnet Eigenschaften, auf die eine Gruppe von Unterklassen der spezifischen Klasse zugreifen können (brauchst Du noch seltener)

**category:** Bezeichnet nur die Gruppe, zu der die Klasse gehört. Obwohl dies kein echter Bestandteil der Sprache ist, muss die Angabe einer Kategorie erfolgen.

Ist die Klasse erst einmal erstellt, werden gegebenenfalls zusätzliche Methoden eingebaut. Nennen wir die Klasse unserer veränderlichen Bläulinge einfach *VBlaeulinge*, so erbt diese Klasse alle Festlegungen der Klasse *Blaeulinge*, wenn an diese die *subclass...* Nachricht geschickt wird.

Die *Vererbung*, also die Wiederverwendbarkeit von Code ist der mit Abstand größte Vorteil der objektorientierten Programmierung.

Während wir bei der Besprechung von Objekten die drei Prinzipien durch die Worte *Objekt, Nachricht* und *Methode* beschrieben haben, können wir nun bei den Klassen drei weitere ergänzen.

**Klasse:** Eine Klasse legt die Variablen und die Methoden der Objekte (Instanzen) fest.

**Subklasse:** Jede Klasse, mit Ausnahme der obersten Klasse *Object*, ist Subklasse einer Superklasse. Und zwar jeweils nur einer einzigen (anders als bei C++). Eine Superklasse kann beliebig viele Subklassen in die Welt setzen, selbst aber nur Subklasse einer einzigen Superklasse sein. Alle Klassen sind Subklassen von *Object*. Durch Subklassenbildung werden Variablen und Methoden vererbt. Die Klasse *Object* vererbt zum Beispiel die Klassenmethoden *new* und *subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:*.

**Metaklasse:** Jede Klasse ist ein Objekt und damit Instanz einer Metaklasse. Da der Umgang mit Metklassen eine Art Beschäftigung mit dem »Jenseits« von Squeak darstellt, werden wir erst im konkreten Kapitel über Smalltalk davon etwas erfahren.

Das war nun das ganze Geheimnis, was sich hinter der Worten Objekt und Klasse verbirgt. Schauen wir noch ein letztes Mal in die Abbildung 6 auf Seite 12. Die Vererbungslinien sind dort durch braune Pfeile dargestellt. Die Klasse *Schmetterlinge* ist also Superklasse der Subklassen *Blaeulinge* und *KleineFuechse*, aber natürlich nicht der Klasse *Libellen*. Vielleicht fällt Dir auf, dass die Klasse *Schmetterlinge* selbst keine Instanzen erzeugt. Ihre einzige Aufgabe besteht lediglich darin, zu vererben. Tatsächlich gibt es in der Natur ja auch nicht »den Schmetterling«. In der Biologie wird eine Klasse, die Instanzen erzeugt, im allgemeinen als *Familie* bezeichnet. Eine Klasse, die keine Instanzen erzeugen soll, heißt *abstrakte Klasse*. Der echte Klassenbegriff der Biologie entspricht also einer abstrakten Klasse. Die Klasse Vögel legt zum Beispiel fest, was ein Vogel ist. Aber es gibt natürlich nicht *den Vogel*, schon eher aber *das Rotkehlchen*. Die Familie der Rotkehlchen ist also eine Klasse, die Instanzen erzeugt, die Klasse der Vögel ist dagegen im Sinne von Smalltalk eine abstrakte Klasse. Natürlich ist die schon erwähnte oberste Klasse *Object* eine abstrakte Klasse.

Die Abbildung legt allerdings vielleicht ein Mißverständnis nahe: Eine Instanz ist *keine* Kopie des Klassenobjektes, bei dem nur einige Eigenschaften verändert sind. Die Klasse ist ein völlig *anderes* Objekt als seine Instanzen. Die grafischen Elemente bei Squeak (Morphic) arbeiten aber genau mit dieser Vorstellung der Kopie eines Prototyps. Sie folgen auch im Grunde nicht der Programmiersprache *Smalltalk*, sondern einer ihrer Dialekte mit dem Namen *Self*, der also nicht *ganz* konsequent das objektorientierte Prinzip umsetzt. Der Unterschied ist marginal und wird Dich nicht allzu sehr beschäftigen müssen. Vielleicht ist es aber für Dich von Interesse, dass Java ein Self-Abkömmling ist.

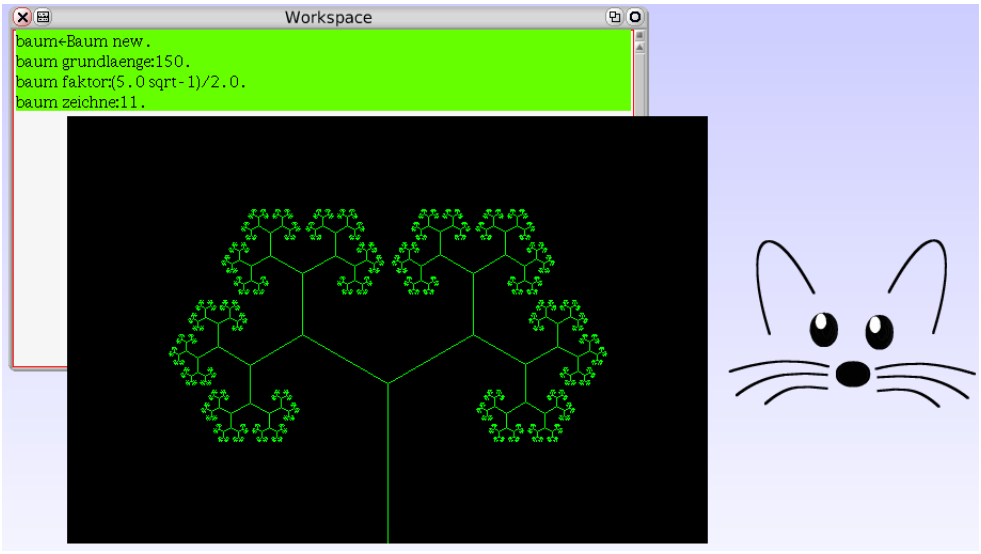


Abbildung 7: LINDEMAYER-Systeme mit Turtle-Grafik

# 6 Turtle-Grafik

Ein Vorläufer der objektorientierten Programmierung ist Seymour PAPERTS Turtle-Grafik, die in den frühen siebziger Jahren am MIT (Massachussetts Institue for Technology) entwickelt wurde. Der Roboter, der hier gesteuert wurde, war in der Tat zunächst eine Schildkröte, die in der Mitte einen Stift trug. Je nach Befehl (Nachricht) an diese Schildkröte konnte sie sich vorwärts bewegen, drehen, oder den Stift senken und heben. Natürlich wurde später die Schildkröte auf dem Bildschirm nur noch simuliert. Die dazugehörige Programmiersprache *Logo* hat nicht nur zufällig den gleichen Klang wie *Lego*. Mit dieser Turtle-Grafik können kinderleichte Grafiken als auch sehr anspruchsvolle Projekte erstellt werden. Die Abbildung 7 zeigt ein solches Besipiel, wo ein Petersilien-Baum durch fortwährendes Zeichnen eines Y bei stetiger Verkleinerung erstellt werden kann. Pflanzen, die durch eine solche Vorschrift erzeugt werden können, heißen nach dem norwegischen Botaniker Aristid LINDEMAYER entsprechend Lindemayer-Systeme. Die Turtle, die dieses Bild erzeugte, ist auf dem Bild nicht zu sehen. Es ist sozusagen ein unsichtbarer Stift der Klasse *Pen*.

In Squeak wimmelt es vor Turtle-Grafik. Natürlich sind nicht *alle* Grafiken auf der Turtle-Technik aufgebaut. Besonders aber die zugänglichsten. Das erste praktische Kapitel mit dem Namen EToys stellt eine solche stark erweiterte Grafik vor. An einer späteren Stelle lernst Du eine noch eindrucksvollere Implementation unter dem Namen StarSqueak oder StarLogo kennen, bei dem Hunderte oder Tausende von Turtlen gleichzeitig laufen und biologische Abläufe simulieren können.

# 7 Das Ende der Reise ist der eigentliche Anfang...

Wir sind am Ende unserer ersten Reise angelangt. Sie fand bisher nur auf dem Papier und im Kopf statt. Ich verspreche Dir: es ist die *einzige* Reise, die so theoretisch abläuft. Aber nun weißt Du ein wenig mehr, was Dich erwartet. Du bist bestens für die nächsten Ausflüge gerüstet, da Du das Konzept der objektorientierten Programmierung – und damit auch die Simulation der Natur – in den Grundzügen verstanden hast. Es ist absolut unmöglich, *alles*

in Squeak zu beschreiben, da sich das System ständig erweitert. Jedes Kapitel geht nur so weit, dass Du selbst weiterforschen kannst. Und das ist auch der Sinn der Sache. Wenn wir einen Urwald mit allen seinen Arten und Gefahren bis in jedes Detail beschreiben wollten, so bräuchten wir gar nicht mehr auf Entdeckungsreise zu gehen. Aber es ist das Ziel, Dich zum Entdecker *und* Erfinder auszubilden. Es lohnt sich und wird Dir hoffentlich sehr viel Spaß bereiten! Die Squeak-Maus wartet jedenfalls auf Dein nächstes Abenteuer.

The best way to predict the future is to invent it.
Alan KAY

# 8 Kleine Kontrollfragen

**Frage 1** *Versuche genau zu beschreiben, welche Elemente ein Objekt auszeichnen.*

**Frage 2** *Was ist der Unterschied zwischen einer Methode und einer Nachricht (Message)?*

**Frage 3** *Welche Aufgaben haben Klassen? Was bedeutet das Wort Instanz?*

## Index