Seminar Report

Smalltalk
Specialization Seminar
99 Bottles of Beer

Armin Gufler
armin.gufler@student.uibk.ac.at

18 February 2011
Supervisor: Sarah Winkler

Abstract

This seminar report is about the programming language Smalltalk. After a short motivation and introduction to present Smalltalk, the history of this quite old language and the process of its development is described, pointing out the reasons and the purpose of developing it and what languages influenced this process. In the following sections the main features, syntactical constructs, concepts and control structures of the language are presented. First important characteristics like object-orientation and message passing within Smalltalk are described. Second the most important syntactical rules and constructs are explained. The description of the key concepts like classes and objects and structures like loops then shows the great possibilities of Smalltalk, comparing it constantly to modern languages such as Java or C++. To show how a program is developed and executed, the Smalltalk Environment is explained. Finally the usage of Smalltalk in the past, its usage now and its influence on other languages nowadays is illustrated.

Contents

1 Introduction

This section should be a motivation and give a short description of what Smalltalk is.

The programming language Smalltalk is object oriented, dynamically and strongly typed and based on message passing between objects. It is considered the first pure object programming language with the first real Integrated Development Environment (IDE). The main concept behind Smalltalk is, that everything within the language is an object and these objects communicate with each other simply by sending and receiving messages. The main philosophy of Smalltalk can be summed up by five points, which developer Alan Kay formulated [8].

- Everything is an object
- Objects communicate by sending and receiving messages
- Every object has its own memory
- Every object is an instance of a class
- The class declares the shared behavior for all its instances

Smalltalk was designed to be a simple, elegant and easy to learn language. It was especially developed for a wide range of people, rather than a small group of experts and so one main goal was that Smalltalk would become a language for children and other unexperienced people to become familiar with computers and there functions.

This report is structured such that first the history and creation of the language will be described. The features mentioned above will be described beginning with Section 3. In Section 4 the basic syntax of the language is explained, followed by Section 5 where the most important control structures and concepts are described in detail. To show a current Smalltalk Environment, *Squeak* [3] is presented within Section 6, containing an example of how to model a concrete class. Section 7 shortly depicts the usage of Smalltalk now and in the past.

2 History of Smalltalk

Within this section the most important milestones in the history of Smalltalk and the different versions that were developed will be explained.

The language Smalltalk was mainly designed by Alan Kay, an American computer scientist. Kay developed his idea of an object-oriented language starting in the late sixties. The development and the language itself were influenced by LISP, Simula, Sketchpad and also Logo [8].

2.1 The Idea of Object Orientation

The idea of an object oriented programming (OOP) style came to Kay already in the late sixties as he observed and participated in several projects, involving the programming and development for the Air Force and later on for ARPA.

The motivation for Kay was to stop dividing the computer into weaker parts like procedures and data structures, but to split it up into thousands of little computers (objects), each simulating and offering something useful. With this thoughts the idea of OOP was born.

In 1966 he got confronted with a document named *Sketchpad: A man-machine graphical communication system* [12], which already contained big ideas concerning the graphical user interface and interaction of the user with the computer in general. Together with Simula, a language for simulation purpose, Sketchpad influenced Alan Kay in his later development of Smalltalk. Furthermore Logo and LISP were languages Kay worked with and had influence on his development process [8].

2.2 Smalltalk-71 & 72

In 1971 the first concrete version of the language, called Smalltalk-71, was implemented by Kay's colleague Dan Ingalls. They were both working for Xerox PARC, an important research company from California. Kay and a few colleagues worked on a conceptional computer system called *Dynabook*, which was already similar to tablet PCs nowadays. As a first prototype of the *Dynabook* the system *miniCOM* was developed. In his design of *miniCOM*, which offered a bit-map display and a pointing device, Kay also included this first version of Smalltalk.

The name *Smalltalk* was based on Kay's vision that "programming should be a matter of Smalltalk" and "children should program in Smalltalk" [8]. Kay also wanted to react against the tide that many systems back then got names from ancient gods like Zeus, Thor etc.

Smalltalk-71 then got enhanced further and so Smalltalk-72 was created, which Kay calls "the first real Smalltalk" [8]. In 1972 a computer prototype called *Xerox Alto* was developed, which already had a graphical user interface and was in fact the first computer with such an interface. The *Xerox Alto* included an environment for Smalltalk-72 and was used already to do actual research work. With features like the pointing device (mouse) and the graphical user interface which already used windows, icons etc. it was a very important milestone in the development of modern computers. In fact a lot of these ideas came from the group that developed Smalltalk.

In the seventies Alan Kay also worked with children and adults, which he wanted to teach Smalltalk and an understanding of computers in general. He always saw Smalltalk as a language which should be very intuitive and easy to learn even for young children and indeed his work showed that people were able to solve easy problems with Smalltalk and to learn it fast [8].

In the following years Smalltalk-72 was consecutively improved to the versions Smalltalk-74 and Smalltalk-76.

2.3 Smalltalk-80

As a consequence of the development Smalltalk-80 was created and this version was the first one to be released outside the research company. It represented a temporary standard for the language and was given to companies like HP or Apple and also to the university UC Berkeley.

A final standard for Smalltalk came in 1983 with the book *Smalltalk-80: The Language and its Implementation* [7]. At the same time also Smalltalk-80 Version 2 was released and made available for everyone. The version came as an image with object definitions and a virtual machine and was also platform independent. Later and current versions of Smalltalk are still based on this release.

2.4 Current Versions

Nowadays there are many different implementations of Smalltalk and behind some of them stands a big community supporting and updating it continuously. The most popular implementations are *VisualWorks Smalltalk* [4] and *Squeak Smalltalk* [3]. The latter one is an open source implementation and has a strong developer team and community. Among the contributors are Alan Kay and Dan Ingalls, the two main founders and developers of the language back in the sixties and seventies. VisualWorks is a proprietary version, sold by Cincom. Therefore it is mostly used in a commercial way. Both VisualWorks and Squak are direct descendants of the original Smalltalk-80, which makes them "pure" Smalltalk systems. For a more detailed list of versions the reader is referred to the official Smalltalk website [2], which also provides many other resources to learn more about the language.

3 The Main Characteristics

In this section the main concepts and features of Smalltalk are described.

As already stated in the first section, Smalltalk is object oriented, reflective, has dynamic strong typing and pervasively uses message passing between objects. What each property means will be described now.

3.1 Object Orientation

Nowadays object oriented languages such as Java, C++ and Python are widely used to program modern software systems. Smalltalk was one of the first languages to be considered object oriented, and had a big influence on these languages, but the object orientation of the modern object oriented languages is different from the one of Smalltalk. In Smalltalk everything is an object. This includes numbers, strings, characters, blocks of code, hence it is a pure object programming language. "Purely" object oriented means, that there are no primitive values like integers, characters etc. like in Java. Alan Kay, who was one of the main characters to "invent" object orientation, defined object oriented programming as:

"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things." [9]

This statement is not intended to be a clear definition of object oriented programming. The extreme late-binding (dynamic typing) demonstrates Alan Kays distaste for static types and may be considered a concept which is not crucial for a language to be object oriented. So it can be observed, that languages like Java or C++ would not completely satisfy this definition, because they do not use a dynamic type system, but if messaging is considered the same as method invocation, the other properties are fulfilled.
An object in Smalltalk is always an instance of a class. Like in modern languages the class describes the properties and behavior of the instances. Hence everything is an object, the classes themselves are objects too. They are instances of the so called *metaclasses*, more on that can be found in Section 5.1.

3.2 Message Passing

In Smalltalk every method call and computation is realized by sending messages between objects. Since every object holds a state and this state and the methods of the object are private and not visible to anyone, other objects have to send a message to the object in order to invoke a method. As a consequence of this process, programming in Smalltalk can be seen as objects talking to each other, asking others for services and answering to requests of other objects.

3.3 Strong Dynamic Typing

Smalltalk has a dynamic typing system, which means that type checking is not performed at compile time and that a variable can contain any object. Nevertheless an object does detect if a message having a parameter of the wrong type is received and does only handle messages for which a correct behavior is defined. So even though the language does offer dynamic typing which allows a high applicability and reusability the sending of invalid messages is prevented at runtime which makes the typing strong.

3.4 Reflection

Smalltalk is a reflective programming language which generally means that all the objects, classes and methods are also accessible at runtime. This means that they can

be queried and modified while the program is running by sending messages to them. And since everything is an object and Smalltalk is implemented in Smalltalk itself, the whole system is changeable and has the full power to change itself. Observing this, Smalltalk can be seen as a “living” system.
If for example a class is being redefined (e.g. adding new variables) all existing instances will be changed according to the new definition. This is done by an intern update mechanism that in fact creates a new class and fills instances with data from old instances, but the system continues to run all the time.
The described characteristic is known as structural reflection.

Smalltalk is not just structurally reflective, but also computationally reflective. That denotes the ability to access and observe the computational state of the program (using `thisContext`, see 4.2).

For more details about the concrete reflective aspects and mechanisms, the interested reader is referred to [11].

4 The Basics of the Syntax

Within this section the basic syntax, reserved words and valid expressions and statements of the language are presented.

The core syntax of Smalltalk is rather small compared to the amount of keywords that other languages use.
There are only six reserved words that can be seen as the keywords of the language, because they cannot be changed.

```
nil, true, false, self, super, thisContext
```

Listing 1: The six reserved words of Smalltalk

4.1 Constants

The three objects **`nil, true`** and **`false`** are called constants, because they are predefined and not changeable by the programmer. They are all singleton instances of a respective class.
The constant **`nil`** is used to indicate an undefined value or that there is no value, very similar to `NULL` in C or Java. Technically it is the singleton instance of the class `UndefinedObject`.
The constants `true` and `false` are used to represent the boolean values true and false and are singleton instances of the classes `True` and `False` respectively.

4.2 Pseudo-Variables

There are actually three pseudo-variables, namely `self`, `super` and `thisContext`. The pseudo-variables `self` and `super` are quite similar to `this` and `super` known from Java. So `self` does refer to the current object (receiver of a message) itself. If a message is sent to `super` then the lookup for the method starts in the superclass.
Using `thisContext` the current activation of a method is accessible and the object can send messages to itself, to find out things like who this message came from.

4.3 Variables

In Smalltalk variables point to an object and are used quite similar as in most programming languages. It is not necessary to declare a variable and a variable has no inherent type. To keep it safe, in Smalltalk the programmer is not able to explicitly take the address or use references and pointers. Variables do always point to objects and if used they are always dereferenced implicitly.
The assignment of a value to a variable is realized with the syntax shown in Listing 2.

```
var := expression
```

Listing 2: Assignment syntax

Hence in Smalltalk the variable has no type, every object can be assigned to it and hence everything is an object this is a very strong, profound and interesting concept. When describing constructs like *code blocks* in later sections, it will become clear how many possibilities the programmer has with variables in Smalltalk.

4.4 Comments

Like almost every language Smalltalk does offer the possibility to insert comments everywhere in the code to achieve better readability and understanding of the code. Unlike most languages Smalltalk uses the double quote for comments.

```
"This is a comment"
```

Listing 3: Smalltalk comment

4.5 Literals

Smalltalk allows a lot of objects to be written as literal values. To be precise there are integer literals, scaled decimal literals, floating point literals, character literals, string literals, symbol literals and array literals. So for example for the floating point numbers the precision can be defined explicitly. For string literals the single quote is used and for characters the $-sign is used.

```
1.234              "Floating point, single-precision"
1.234q             "Floating point, quad-precision"
'hello␣world'      "A string literal"
$#                 "The # - character"
```

Listing 4: Literal examples

A more detailed description of all the literals available can be found in [9].

5 The Key Concepts and Control Structures

This section is dedicated to describing the key concepts of Smalltalk such as classes, objects and messages. Furthermore standard constructs like conditional statements and loops are described and compared to the ones known from other languages.

5.1 Classes

As most object oriented languages Smalltalk uses the principle of classes. A class defines the behavior and structure of its instances. The behavior is defined with the specification and implementation of the *instance methods* offered and the structure is specified by the *instance variables*.

Smalltalk	**Java**
Instance Variables	Data Members
Instance Methods	Methods

Table 1: Comparison to Java

The *instance variables* correlate to the *data members* or *fields* in languages like Java or C++. The *instance methods* are similar to the *methods* known from Java.

A class itself is also an object, and as mentioned earlier every class is instance of a *metaclass*, actually the singleton instance. The metaclasses themselves are all instances of the class `Metaclass`, which is then instance of a metaclass again and this `Metaclass class` is then recursively an instance of `Metaclass`.
Let's look at an example. We assume that the class `Person` has been defined and then the "hierarchy" shown in Listing 5 arises.

```
Person          "instance of metaclass Person class"
Person class    "instance of class Metaclass"
Metaclass       "instance of metaclass Metaclass class"
Metaclass class    "instance of class Metaclass"
```

Listing 5: A class hierarchy

Since the class is an object it can have its own methods too, the so called *class methods*. These methods are invoked when sending messages directly to the class instead to instances of the class. It can easily be seen that these class methods are the instance methods of the metaclass defining the class.

5.2 Class Inheritance

The principle of class inheritance in Smalltalk is more or less the same as in Java or C++. When defining a class, the programmer has the possibility to specify a superclass. The created class is then a subclass of this superclass. That means that the subclass inherits all the instance variables and also all the instance methods of the superclass. Within the subclass it is also possible to create an instance method that has the same name as a method of the superclass. This concept is called *overriding* a method and is nowadays well known from other object oriented programming languages. With the help of the “keyword” `super` the subclass can directly send messages to the superclass in order to call methods that in the subclass have been overridden. As the root of the class hierarchy exists a class named `Object`. This is a concept that Java adopted from Smalltalk.
The inheritance of classes is usually also mirrored between the metaclasses of the subclass and the superclass. To get a deeper insight on metaclasses and metaclass inheritance the reader is referred to [7, Chapter 5].

5.3 Objects

As already pointed out before objects are the central concept in Smalltalk. Every object is an instance of a class, therefore it has its instance variables and instance methods, like described before in Section 5.1. In order to communicate with an object, which means invoking its methods, it is necessary to send it a message. Objects can also be passed as arguments in messages, they can be returned as a result of a message (method) and they can be assigned to a variable (see 4.3).

Encapsulation regarding objects is very important in Smalltalk. Actually an object encapsulates three things:

- the object’s behavior
- the object’s state

- the object's structure

The behavior (method) can only be invoked by sending a message to an object, "asking" the object to do something. Similarly the state and the structure can only be accessed and modified by sending a message to the object.

5.3.1 Creating Objects

The syntax to create objects in Smalltalk is very similar to the one known from modern object oriented languages. It is realized by sending the desired class the message `new`. The method `new` can be compared to constructors in Java or C++, but in Smalltalk it is only a normal class method. It is implicitly inherited from the root class and if desired it can be overridden by the developer, for the purpose of variable initialization or other. If nothing is explicitly initialized all instance variables are set to `nil` when creating a new instance.
As an example Listing 6 creates an instance of the `Person` class and assigns it to the variable `myPerson`.

```
myPerson := Person new     "Creating a Person instance"
```

Listing 6: Creating a new object

5.3.2 Garbage Collection

Smalltalk does have an automatic garbage collection. This means that every object that has no remaining references will be deleted automatically. It is not possible to delete an object manually, so the programmer has to remove all references in order to delete the object. This garbage collection system is very similar to the one Java has and results in a very safe and easy way of handling objects, because the programmer does not need to worry about deleting objects. As a consequence of this, memory leaks, that languages like C++ have to struggle with, are no problem in Smalltalk.

5.4 Messages & Methods

In Smalltalk computation is done by sending messages to objects. Sending a message to an object is like asking the object to do something (execute a specific method). The object which receives the message responds by invoking a method, if there is one corresponding to the received message in the objects method namespace. The method's or messages's name is called *selector*.
The messages are dispatched dynamically at runtime. As a central concept of object orientation different objects may of course respond to the same message in a completely different way.

A message in Smalltalk can have zero or more arguments. These correspond to the parameters of a method or function in Java or C++. Three types of messages can be distinguished.

- **Unary Messages**: messages with no argument
- **Binary Messages**: messages with exactly one argument (for $+, -$...)
- **Keyword Messages**: messages with one ore more arguments

5.4.1 Unary Messages

These messages do not have any arguments and can be compared to functions with no parameter.

The following example shows the comparison between Smalltalk and Java, assuming an object called `MyObject` which should be initialized. In Smalltalk the message with name `initialize` is being sent to the object and in Java the method `initialize()` is being called. Note that there is no predefined method called `initialize` in both of these languages, for the example it's just assumed that such a message has been defined.

Smalltalk	**Java**
`MyObject initialize`	`MyObject.initialize()`

Table 2: Comparison to Java

5.4.2 Binary Messages

Binary Messages are used to realize statements such as `5 + 3` or `5 > 3`. The big difference to other programming languages is that in Smalltalk `+` or `>` does not represent an operator.
To take the first example this just means that the message with name `+` is sent to the object `5` and as an argument `3` is given.

This kind of approach has a problem with correct precedences. So for example the statement `5 - 3 * 2` will not result in -1 but in 4. The reason is that messages are evaluated from left to right and so first it will subtract 3 from 5 and then multiply the result with 2. To solve this problem simply parentheses are needed.

5.4.3 Keyword Messages

To describe keyword messages it is essential to look at how a *keyword* in Smalltalk is defined: *A keyword is an identifier which is directly followed by a colon.*

Some simple examples are `key:` or `thisIsAKeyword:`.

Keyword messages are like method calls in Java where the method has one or more parameters, but they are one of the most different constructs regarding the syntax. A keyword message is a message which has one or more arguments and the name (selector) of the message is composed of one or more keywords. The keywords are basically in front of a parameter and therefore like the description of it. Some examples are shown in Listing 7.

```
aPerson name: 'Peter'

aPerson birthdayYear: 1975 month: 2 day: 12

aString copyFrom: 3 to: 7
```

Listing 7: Examples for keyword messages

Listing 8 shows how equivalent Java statements could look like (depending on the methods and messages implementation).

```
aPerson.setName("Peter");

aPerson.setBirthday(1975, 2, 12);

aString.subString(3, 7);
```

Listing 8: Examples for Keyword messages

The examples show an interesting characteristic of Smalltalk, which is that statements often are very similar to an English sentence. Also the keyword does describe the role and kind of the argument if programmed in a nice way.

5.5 Code Blocks

Code Blocks are a concept that cannot be found directly in languages like Java. They are objects and comparable to anonymous functions of other languages. Blocks contain a sequence of executable statements, may also have arguments and a block does also return something, which is the last statement executed in the block. Optionally it is also possible to write an explicit return statement by using the caret character in front of the statement (`^ result`).
Unlike a normal function in another language like C, the code block in Smalltalk does not have a name, but as it is an object it may be assigned to a normal variable. It is also possible to send messages to the block, so for example the message `value` is used

to request a computation with given arguments.
Blocks are especially important in Smalltalk to easily create loops and conditional statements as can be seen in Sections 5.6 and 5.7.

Code Blocks do also represent a real *lambda calculus* implementation with full closure semantics within Smalltalk. If one is familiar with lambda calculus it is easy to see that the lambda expression $\lambda x.(x * 2)$ would correspond to the block `[ :x | x * 2 ]`.

Listing 9 describes the general syntax of blocks and shows some examples.

```
[ :arg0 :arg1 ... | <expressions> ]    "Syntax"

[2+2+4]      "Block with no arguments"

testBlock := [ :a :b |          "Block with 2 arguments"
               b := a + 5.      "Set b to a + 5"
               b + a            "Return b + a"
             ]

testBlock value: 2 value: 3     "Sending message with
                                 arguments 2 and 3 to
                                 the block"
```

Listing 9: Code Blocks

Note that regarding the statement sequences Smalltalk uses the period as a separator. This gives the code a similarity to a natural language, which follows the concept that Smalltalk should be a language which is easy to understand.

5.6 Conditional Statements

Unlike other languages Smalltalk does not offer a predefined syntax for *if*-statements. But there are easy ways to realize these constructs using objects and sending messages. Namely there are the following messages that can be sent:

- `ifTrue:` corresponds to `IF...THEN...`
- `ifFalse:` corresponds to `IF NOT...THEN...`
- `ifTrue:ifFalse:` corresponds to `IF...THEN...ELSE...`

This means it is possible to test if something is true or false and then do something by using the first two messages and giving one argument to the conditional block. The third message realizes the `IF...THEN...ELSE...` construct and needs, as the selector shows, two arguments. Additionally there is also `ifFalse:ifTrue:` as the reverse construct. As arguments blocks must be used. Listing 10 shows an example for an `IF...THEN...ELSE...` construct.

```
(x > 0)
   ifTrue:[ y := 'positive' ]
   ifFalse:[ y := 'negative' ]
```

Listing 10: Example for conditional statement

5.7 Loops

The principle to create loops is similar to the one of the *if*-statements. In order to create a *while*-loop it is possible to send a `whileTrue:` message to a block giving another block as argument. An example is shown in Listing 11.

```
i := 0.
x := 1.
[ i < 10 ] whileTrue: [
   x := x * 2.
   i := i +1
]
```

Listing 11: A *while-loop*

The *for*-loop is also quite easy to write by using the pattern *startIndex* `to:` *endIndex* `do:` *block*. The block has to have one argument (for the index). Listing 12 shows an example for a *for*-loop.

```
x := 1.
1 to: 10 do: [ :i |
   x := x * 2
]
```

Listing 12: A *for-loop*

5.8 Indexed Instance Variables

Indexed instance variables are a powerful and flexible concept to realize objects which represent an ordered sequence of some objects. These objects are needed to represent things like arrays and strings. The indexed instance variables are different from the normal (called named) instance variables. The named instance variables bind a name to a value whereas indexed instance variables bind an index to a value. The index is an unsigned positive integer. An object with indexed variables may also contain named variables, but it can contain only one set of indexed instance variables.

As an example lets look at the class `Array` which is implemented using indexed variables. This allows the usage of arrays similarly to other language even though there is no special structure for arrays in Smalltalk. Listing 13 shows how arrays are used.

```
myArray := Array new: 5     "Create an array of size 5"
myArray at: 2 put: 42       "Putting number at index 2"
myArray at: 2               "Get number at index 2"
```

Listing 13: Arrays as an example for objects with indexed variables

Smalltalk does offer also data structures like lists or sets which are implemented using arrays and objects with indexed instance variables. More details about these topics can be found in [7] and [10].

6 The Smalltalk Environment

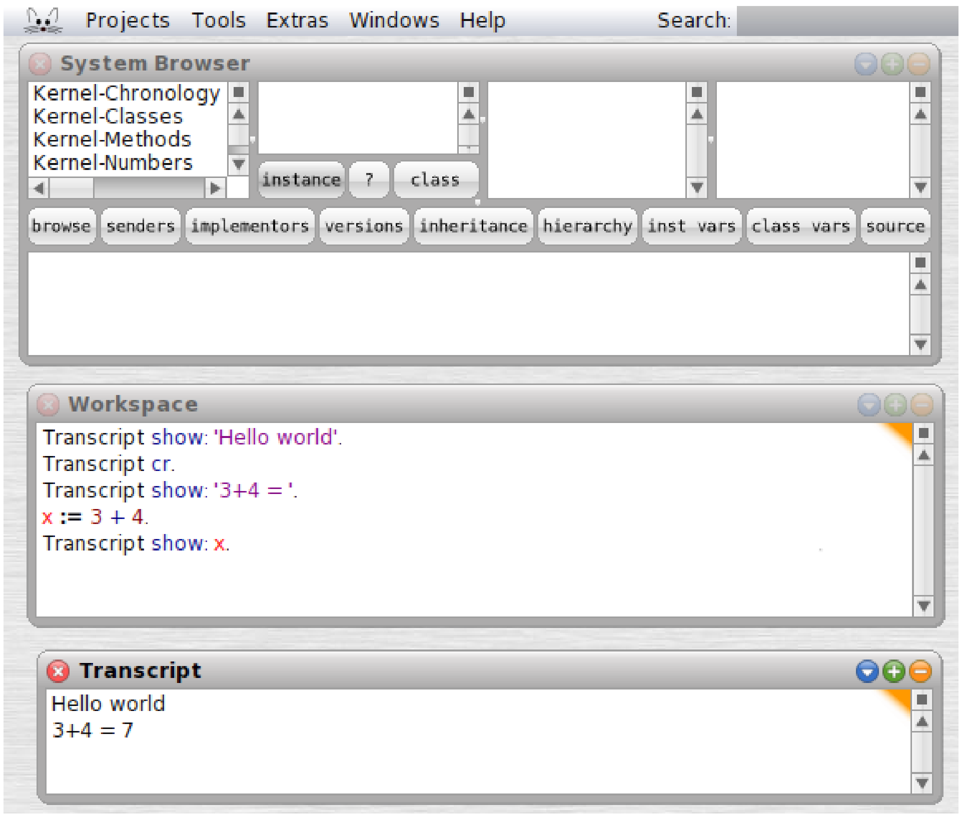
Programming with Smalltalk cannot be compared to programming in other common languages. Smalltalk is not just a language where code is written in a file and then compiled and executed. It is a whole environment itself, just like an operating system. In fact in the past it was designed to run on a dedicated machine as the main system and not on top of another OS.

The Smalltalk Environment basically just contains objects that are sending messages to each other. The system can be seen as a "living" system because all the objects are encapsulated within an image. This image has an address space, a persistent storage and runs in a special process. To develop applications running Smalltalk on top of another OS like Linux, the Virtual Machine of Smalltalk with an image file is used.

Hence all the objects within this image or system are running and "alive" it is not possible to distinguish runtime and compile time from each other. Even though the system is running all the time the programmer has all possibilities to change or add something. This includes adding methods to classes, adding new classes, modify (or also rename) classes or methods, insert new subclasses or superclasses and so on. The compilation takes place if a message or something else is changed and the programmer confirms these changes. This compilation is fast, has no link or load phase and is not visible for the developer. After adding methods, classes or making some other changes the image file can be saved. This is comparable to an operating system which is hibernating. Due to this principle, if using implementations that are derived directly from the original Smalltalk-80 (like *Squeak* or *VisualWorks*), the programmer actually works with the same objects that began to exist and run back in the seventies.

The typical Smalltalk Environment is written mainly in Smalltalk itself, there are only some routines within the virtual machine written in assembly language or C that allows the environment to access hardware with high-level Smalltalk code.

A Smalltalk Environment normally is a window-based interface. It offers a main window



```
Transcript show: 'Hello world'.
Transcript cr.
Transcript show: '3+4 = '.
x := 3 + 4.
Transcript show: x.
```

```
Hello world
3+4 = 7
```

Figure 1: Squeak Environment with open Browser, Transcript and Workspace

part called the *world*. The system is designed to be controlled with a pointing device (mouse) with three buttons. In documentations these buttons are often referred to with different colors in order to keep it general and to avoid using right/left button. There are some key tools within the environment a programmer will use a lot.

- **The System Browser:** used to view, delete, modify and create new classes and their methods
- **The Workspace:** an empty window where arbitrary code and text can be written to and expressions may be evaluated or printed.
- **The Debugger:** a handy tool to debug methods and look at current variables and methods.
- **The Inspector:** used to view existing objects in an easily readable way
- **The Transcript:** an object used to log system messages

To visualize what an environment looks like, the current open source implementation Squeak in the version 4.1 is used [3].
Figure 1 shows the Squeak environment with the standard image file. Within the world

the System Browser, a Workspace and a Transcript have been opened. In this example the Browser is not used, but in the Workspace some example statements have been written. To visualize them on the screen the Transcript is used by sending the message `show` with a value to it. The message `cr` tells the Transcript to insert a carriage return (new line).

6.1 Example: Creating a Simple Class

This short section shows how to create a simple class with Squeak Smalltalk.

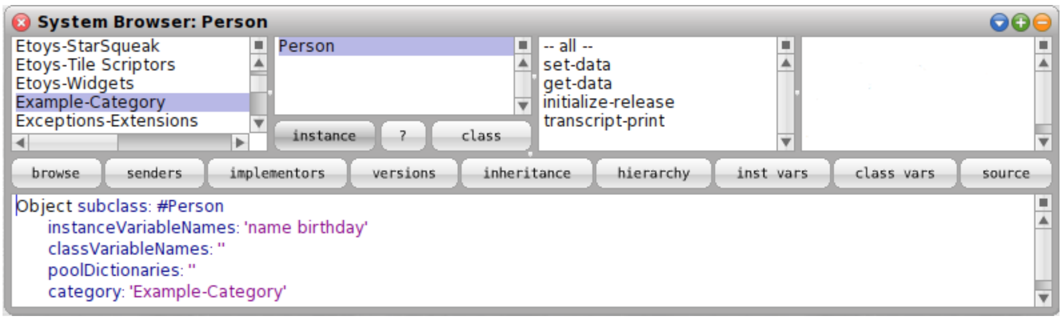
The goal is to create a class representing a `Person` which stores the name and the birthday. The first step is defining the class itself. This can be done using the System Browser with the code shown in Listing 14. It creates a class called `Person` as a subclass of `Object`. The class has the instance variables `name` and `birthday`.
Note that the #-sign is used for symbol literals, which specify instances of the class `Symbol`. More on symbols and their usage can be found in [9].

```
Object subclass: #Person
    instanceVariableNames: 'name␣birthday'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Example-Category'
```

Listing 14: Defining the class

The class does not have any class variable and does not use any pool dictionaries. A pool dictionary is a dictionary (a type of collection) whose keys define variables which can be shared by multiple classes. This means that the class can access any variable within a dictionary stated in `poolDictionaries:`. This concept may be compared to namespaces used in other languages. The category of a class is used to organize classes in a structured way, comparable to packages in Java. The available categories are listed in the leftmost pane of the System Browser. As category any available category can be selected, in the example a first defined *Example-Category* is chosen.
Notable is also that the class is being created by sending a message to the class `Object`. The class `Object` will understand this message and create the `Person` class as a subclass with the given specifications. This shows the characteristic that everything is done by sending messages.

Now it is possible to select the new class in the System Browser.
The next step is to create some methods for the class. The methods are all categorized within different protocols, visible and modifiable in the third pane of the System Browser. Therefore it is recommended to first create some protocols to keep the class well-arranged (e.g. *get-data, set-data, initialize-release, transcript-print*). Figure 2 shows new class including some defined protocols. For the `Person` class some simple methods to initialize the object, to get the name and birthday, to set them and to print the information about the person will be created.



```
Object subclass: #Person
	instanceVariableNames: 'name birthday'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Example-Category'
```

Figure 2: The System Browser showing the new class

Initializing

First a method to initialize an object with some default values for the instance variables is presented. The method is put in the protocol *initialize-release*.

```
initialize
    "Initialize Person with some default values"
    name := 'undefined'.
    birthday := 'unknown'.
```

Listing 15: Initialize method

Note that it is not mandatory to define such a method. Smalltalk will just initialize every variable with `nil` after the creation of the instance, but it is a good practice to initialize manually. Another solution for this problem would be to override the `new` method.

Setting data

To set the values of the instance variables the methods `name:` and `birthdayYear: month: day:` could be defined as shown in Listing 16 and 17.

```
name: aString
    "Sets the name of the person"
    name := aString.
```

Listing 16: Method to set the name

```
birthdayYear: year month: month day: day
    "Sets the birthday"
    birthday := year asString , '-' ,
                month asString , '-' , day asString.
```

Listing 17: Method to set the birthday

Note that in the example method to set the birthday, the parameters are all converted to strings using the predefined message `asString` and the strings are concatenated with the comma, resulting in a single string as value of the instance variable `birthday`. Here the class could of course be modeled such that there is a variable for year, month and day or the method to set the birthday could just take one argument, with the drawback that there is no fixed format for the date.
For the example this version is used mainly for the purpose to show how a method with more keyword parameters is realized in practice.

Getting data

The methods to get the value of an instance variable are really simple. It is only necessary to return the value using the caret sign (^). Listing 18 shows the method to return the name.

```
name
   "Gets the name of the person"
   ^name
```

Listing 18: Method to get the name

Note that there is no conflict having a method `name:` and `name` , because the selector is clearly distinguishable. The first method will be executed when receiving the keyword message with selector `name:`. When receiving the unary message `name` the second method will be executed.
Returning the birthday variable is the same and therefore omitted at this point.

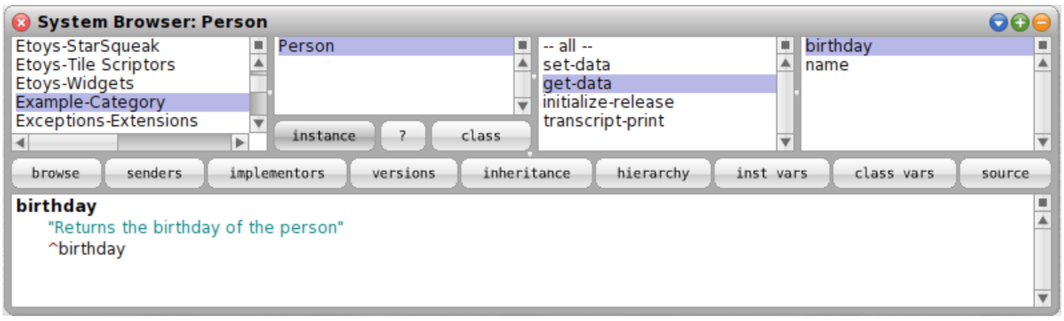
Printing information on the Transcript

To print out all the information on the Transcript the method shown in Listing 19 can be used. For order purposes it should be saved in a category like *print-transcript* or similar.

```
printInfo
   "Prints person info to Transcript"
   Transcript show: 'The␣name␣of␣the␣person␣is␣'.
   Transcript show: name.
   Transcript show: ',␣it‘s␣birthday␣is␣'.
   Transcript show: birthday.
   Transcript cr.
```

Listing 19: Method to print information

Now the `Person` class with all the methods is visible in the System Browser like any other class, as can be seen in Figure 3.


System Browser: Person

```
birthday
	"Returns the birthday of the person"
	^birthday
```

Figure 3: The System Browser showing the created example class

Its now possible to use the class. An example use case is presented in Listing 20.

```
myPerson := Person new.
myPerson printInfo.
myPerson name: 'Peter'.
myPerson birthdayYear: 1975 month: 2 day: 12.
myPerson printInfo.
```

Listing 20: Testing the new class

First a new instance is created and then the initialized instance is printed to the Transcript. Afterwards the values are changed and the data is printed again. The Transcript shows the following after executing the code above:

The name of the person is undefined, it's birthday is unknown
The name of the person is Peter, it's birthday is 1975-2-12

To learn more about Squeak Smalltalk the official Squeak website is recommended [3]. A nice tutorial with lots of similar examples and guidelines can be found in the open book *Squeak by Example* [6].

6.2 Using Reflection

This section shows how the reflective behavior of Smalltalk can be used by providing some simple examples based on the class presented in the previous section.

As explained earlier Smalltalk offers a lot of reflective features which allow the programmer to access the state of objects at runtime and to change the whole system while it is running.
To visualize these characteristics the `Person` class from above is used. Additionally there have been created two subclasses `Student` and `Professor`. Listing 21 shows some messages that can be sent to the class to retrieve information about it. The result is given right next to the statement as a comment.

```
"Get the class a variable is instance of"
myPerson class. "Person"

"Get the parent class"
Person superclass. "Object"

"Get all subclasses"
Person subclasses.  "{Student . Professor}"

"Test if the class responds to a specific message"
Person respondsTo: #name.   "true"
Person respondsTo: #foo.    "false"

"Get the instance variables of the class"
Person instVarNames.     "#('name' 'birthday')"

"Get all available message selectors"
Person selectors.   "#(#birthday #printInfo
                     #birthdayYear:month:day:
                     #initialize #name: #name)"
```

Listing 21: Examples for reflection

It is also possible to call a method using reflection with the `perform:` message providing the selector of the desired method as an argument: `perform:#selector`. This may be useful if the selector is not known until runtime. Listing 22 shows an example using `perform`.

```
"Sending the message regularly"
myPerson name.
"Equivalent: Sending the message using perform:"
myPerson perform: #name.
```

Listing 22: An example using reflection to call a method

The presented concepts are very useful, if the programmer does not know the structure, inheritance hierarchy or the methods of an object he is working with. With the provided reflective behavior the class of an object can be determined and all the properties of a class like the available methods can easily be retrieved during runtime.
This is a really powerful feature of Smalltalk and even though many other languages offer some sort of reflective concepts, they often only provide them using some extensions or special libraries. For example in Java the package `java.lang.reflect` offers some reflective features.
To get more informations about the reflective behavior of Smalltalk the reader is referred to [11].

7 Usage in the Past, Usage Now

Smalltalk was originally designed by Alan Kay to be a language which is easy to learn, has a simple syntax and motivates the developer to try new things and be creative. Therefore it was a goal that children and unexperienced users should be able to learn programming and understand computers with the help of Smalltalk.

Hence Smalltalk is not just a language but a whole environment, it was also intended to run as an operating system on a dedicated machine. In fact it actually includes tools to access hardware, schedulers for threads and basic programs an operating system needs. Nevertheless Smalltalk never became really famous. One reason that it did not become as famous as Windows, Unix or Mac OS is, that it was never really advertised by a strong company to the end users. Another problem was exactly this concept of a whole operating system. It was just a big problem that for Smalltalk-80 an entire computer was needed to use it, because of the high costs for hardware at that time. A very important factor in the past were also the hardware requirements of the system. At this time computers were just not ready for such high level things like the user interface with windows. The limitation was mainly the memory, because in these days computer memory was really small compared to now and it was seen as a huge waste that Smalltalk needed up to two megabytes. Therefore Smalltalk did not manage to establish a good image on the market and when systems such as Windows, Unix or Mac OS became popular it became even more difficult, because now users were accustomed to these operating systems.

Nowadays Smalltalk does not really try to be an operating system anymore. With the fast machines it is easily possible to run Smalltalk on top of an installed OS. In fact it is really fast and does not need much resources and it can be used even in systems considered as slow and cheap. As an example Squeak is used in a project where small and simple laptops are developed for children in poor regions all over the world[1]. It finds use also in other areas regarding the education of children, which inventor Alan Kay does support a lot even nowadays.
But there exist also Smalltalk-based OO-Databases (e.g. *Magma* [5]) or powerful tools for Web application development (e.g. *Seaside* [1]) and many other applications.

[1]`http://one.laptop.org/`

8 Conclusion

Smalltalk is a language with a long history which never became really a widely used language in business, but it was important for the development of our modern computers. It helped a lot to develop central features of computers known nowadays, like the pointing device, windowing system, "personal" computers and others. Smalltalk used these concepts long before others did and a lot has been adopted from Smalltalk to develop Apple's Mac OS or Microsoft's Windows. And as already mentioned before developer Alan Kay did not really aim at the big commercial success.
Further Smalltalk crucially contributed to the development of object oriented programming as we know it today. Actually it was a pioneer on this area and as described throughout this report modern languages like Java adopted many concepts from Smalltalk.

With the implementation of Squeak the language gained new popularity and a quite large community supporting Smalltalk. With Squeak there is now an easy and free way to get a pure and real Smalltalk implementation. Many programmers appreciate the easy concept of Smalltalk a lot and love the fast way of solving problems with Smalltalk. Alan Kay and other developers claim that Smalltalk even nowadays is a language where a programmer can create certain solutions faster than in most other languages and especially changing things later on is faster and easier, because of the reflective behavior.
Smalltalk is not dead, it probably won't be in the near future and maybe even more and more programmers learn to appreciate this easy and simple but also strong and powerful language.

References

[1] Seaside Website. `http://www.seaside.st/` [Last accessed: 16 February 2011].

[2] Smalltalk. `http://smalltalk.org/` [Last accessed: 16 February 2011].

[3] Squeak. `http://www.squeak.org/` [Last accessed: 16 February 2011].

[4] Visualworks. `http://www.cincomsmalltalk.com/main/products/visualworks/` [Last accessed: 16 February 2011].

[5] The Magma Object Database. `http://wiki.squeak.org/squeak/2665` [Last accessed: 16 February 2011], 2010.

[6] A. Black, S. Ducasse, O. Nierstrasz, and D. Pollet. *Squeak by Example*. 2007. Available from: `http://squeakbyexample.org/SBE.pdf` [Last accessed: 16 February 2011].

[7] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[8] A. Kay. The early history of Smalltalk, 1994. Available from: `http://smalltalk.org/smalltalk/TheEarlyHistoryOfSmalltalk_I.html` [Last accessed: 16 February 2011].

[9] A.L. Lovejoy. Smalltalk: Getting the message, 2007. Available from: `http://smalltalk.org/articles/article_20100320_a3_Getting_The_Message.html` [Last accessed: 16 February 2011].

[10] H. Porter. Smalltalk: A white paper overview. Technical report, Computer Science Department Portland State University, 2003. Available from: `http://web.cecs.pdx.edu/~harry/musings/SmalltalkOverview.html` [Last accessed: 16 February 2011].

[11] F. Rivard. Smalltalk: a Reflective Language. Available from: `http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/rivard/rivard.html` [Last accessed: 16 February 2011].

[12] I. Sutherland. Sketchpad: A man-machine graphical communication system. Technical report, University of Cambridge, Computer Laboratory, 2003. Available from: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf` [Last accessed: 16 February 2011].