Object-Oriented Programming in Smalltalk

Gabriela Arévalo, University of Berne
Alexandre Bergel, University of Berne
Prof. Dr. Stéphane Ducasse, Université de Savoie
Dr. Roel Wuyts, Université Libre de Bruxelles

November 21, 2006

1

Smalltalk Environment Basics

Main Author(s): Ducasse and Wuyts

1.1 Videos

You can find videos showing some important points of Smalltalk manipulation at http://www.iam.unibe.ch/ ducasse/Videos/SqueakO

1.2 Starting up

Similarly to Java, Smalltalk the source code is translated to byte-codes, which are then interpreted and executed by the Smalltalk Virtual Machine. (Note that this is an approximation because Smalltalk dialects were also the first languages to develop Just in Time compilation, i.e., a method is compiled into byte-codes but also into native code that is directly called instead of executing the byte codes.)

There are three important files:

visual.im (Binary): contains byte code of all the object of the system, the libraries and the modifications you made.

visual.cha (ASCII): contains all the modifications made in the image-file since this was created. It contains also all the code of the actions you will perform.

visual.sou (ASCII): contains the textual code of the initial classes of the system.

To open an image with drag-drop (on Macintosh, Windows):

- Drag the file 'visual.im' on the virtual machine to start the image.
- If you want to start your own image, just double click on it or drag it over the virtual machine.

On Unixes (Linux, Solaris, HPUX, AIX, ...): you should invoke the virtual machine passing it an image as parameter. For the first opening, execute the first script that per default uses the original image the script is installation dependent but should look like path/bin/visualworks path/image/visual.im Then after you can specify your own image.

1.3 First Impressions

After opening the image, and thus starting a Smalltalk session, you see two windows : the launcher (with menu, buttons and a transcript), and a Workspace window (the one containing text). You can minimize or close this last one, since we do not need it for the moment.

The launcher is the starting point for working with your environment and for the opening of all the programming tools that you might need. To begin, we will first create a fresh image.

Creating a fresh image. We are going to create an image for this lesson.

- select Save As... in the menu
- when the system prompts you for the name for the new image, you type *lesson*, followed by your username.
- the image is saved in the image directory.
- Have a look at the Transcript, and note what it says.

The Transcript is the lower part of the Launcher, and gives you system messages, like the one you see right now. We will see later on how you can put your own messages there.

About the mouse. Smalltalk was the first application to use multiple overlapping windows and a mouse. It extensively uses three mouse buttons, that are context sensitive and can be used everywhere throughout Smalltalk:

- the left mouse button is the select button
- the middle button is the operate button
- the right button is the window button

On a Macintosh, where only one button is available, you have to use some keyboard keys together with pressing your mouse button:

- the select button is the one button itself
- for the operate button, press the button while holding the alt-key pressed
- for the window button, press the button while holding the apple-key pressed

1.4 Adding Goodies and setting Preferences

Out of the box, there is already quite some code in a Smalltalk image (about 1000 classes containing the basic system: the complete compiler, parser classes, GUI framework, development environment, debugger, collection libraries, etc.) [1]. But for developing, it is convenient to load some extra tools.

You may have to load a package named *ImageConfiguration* or parcels. To load parcels we can use the *Parcel Manager* application (open it using the System menu in the Launcher). There is lots of optional applications you might load. Use it to load:

- *RBSUnitExtentions* (in *Environment Enhancements*)
- *MagicKeys* (in *Environment Enhancements*)
- *ColorEditing* (in *Environment Enhancements*)
- *RB_Tabs* (in *Environment Enhancements*)

You can also edit systemwide preferences. To do so, open the *Settings* application, again in the System menu in the Launcher. Use it to set a setting in Tools/Browser: select *Show all methods when no protocols are selected.*

[1] To answer a common question: yes, there are ways to strip this so that you can deploy smaller images to clients that do not contain all of these tools.

1.5 Selecting text, and doing basic text manipulations

One of the basic manipulations you do when programming is working with text. Therefore, this section introduces you to the different ways you can select text, and manipulate these selections.

The basic way of selecting text is by clicking in front of the first character you want to select, and dragging your mouse to the last character you want in the selection while keeping the button pressed down. Selected text will be highlighted.

Exercise 0 Select some parts of text in the Transcript. You can also select a single word by double clicking on it. When the text is delimited by ' (single quotes), "" (double quotes), () (parentheses), [] (brackets), or (braces), you can select anything in between by double clicking just after the first delimiter.

Exercise 1 Try these new selection techniques.

Now have a look at the text operations. Select a piece of text in the Transcript, and bring on the operate menu. Note that you have to keep your mouse button pressed to keep seeing the window.

Exercise 2 Copy this piece of text, and paste it after your selection. Afterwards cut the newly inserted piece of text.

Exercise 3 See if there is an occurrence of the word visual in the Transcript. Note that to find things in a text window, there is no need to select text. Just bring up the operate menu .

Exercise 4 Replace the word visual with C++ using the replace operation (if it does not contain Smalltalk, add this word or replace something else). Take your time and explore the different options of the replace operation.

Exercise 5 Bring up the operate menu, but don't select anything yet. Press and hold the shift button, and select paste in the operation menu. What happens ?

1.6 Opening a WorkSpace Window

We will now open a workspace window, a text window much like the Transcript, you use to type text and expressions and evaluate them.

To open a workspace:

- select the tools menu in the Launcher
- from the tools menu, select Workspace
- You will see a framing rectangle (with your mouse in the upper left corner), that indicates the position where the Workspace will open. Before you click, you can move your mouse around to change this position. Click one time once you have found a good spot for your Workspace.
- Now your mouse is in the bottom right corner, and you can adjust the size. If you click once more, once you have given it the size you like, the Workspace window appears.

This is the basic way of opening many kinds of applications. Experiment with it until you feel comfortable with it.

1.7 Evaluating Expressions

In the Workspace, type : `3`. Select it, and bring up the operate menu. In the operate you will see the next three different options for evaluating text and getting the result:

do it: do it evaluates the current selection, and does not show any result of the execution result.

print it: prints the result of the execution after your selection. The result is automatically highlighted, so you can easily delete it if you want to.

inspect it: opens an inspector on the result of the execution.

The distinction before these three operations is essential, so check that you REALLY understand their differences **Exercise 6** Select 3, bring up the operate menu, and select print it.

Exercise 7 Print the result of 3+4

Exercise 8 Type Date today and print it. Afterwards, select it again and inspect it.

After exercise 9, you will have an inspector on the result of the evaluation of the expression Date today (this tells Smalltalk to create an object containing the current date). This Inspector Window consists of two parts: the left one is a list view containing self (a pseudo variable containing the object you are inspecting) and the instance variables of the object. Right is a text field.

Exercise 9 Click on self in the inspector. What do you get? Does it resemble the result shown by printstring?

Exercise 10 Select day. What do you get? Now change this value, bring up the operate menu, and select accept it. Click again on self. Any difference?

Exercise 11 In the inspector edit field, type the following: self weekday, select it and print it. This causes the message weekday to be sent to self (i.e., the date object), and the result is printed. Experiment with other expressions like:

```
self daysInMonth
self monthName
```

Close the inspector when you are finished.

Exercise 12 Type in the Workspace the following expression: Time now, and inspect it. Have a look at self and the instance variables.

Exercise 13 Type in the Workspace the following expression: Time dateAndTimeNow. This tells the system to create an object representing both today's date and the current time, and open an inspector on it. Select the item self in the inspector. [Note that self is an object called an Array. It holds on to two other objects (elements 1 and 2). You can inspect each element to get either the time or the date object.

Using the System Transcript. We have already seen that the Transcript is a text window where the system informs you important information. You can also use the Transcript yourself as a very cheap user interface.

If you have a Workspace open, place it so that it does not cover the System Transcript. Otherwise, open one and take care of where you put it. Now, in the Workspace, type:

```
Transcript cr.
Transcript show: 'This is a test'.
Trancript cr.
```

Select these 3 lines and evaluate (do It) them with do it. This will cause the string This is a test to be printed in the Transcript, preceeded and followed by a carriage return. Note that the argument of the show: message was a literal string (you see this because it is contained in single quotes). It is important to know, because the argument of the show: method always has to be a string. This means that if you want any non-string object to be printed (like a Number for example), you first have to convert it to a string by sending the message printString to it. For example, type in the workspace the following expression and evaluate it:

Transcript show: 42 printString, 'is the answer to the Universe'. Note here that the comma is used to concatenate the two strings that are passed to the show: message 42 printString and 'is the answer to the Universe'.

Exercise 14 Experiment on your own with different expressions. Transcript cr ; show: This is a test ; cr Explain why this expression gives the same result that before. What is the semantics of ; ?

2

Objects and expressions

This lesson is about reading and understanding Smalltalk expressions, and differentiating between different types of messages and receivers. Note that in the expressions you will be asked to read and evaluate, you can assume that the implementation of methods generally corresponds to what their message names imply (i.e., 2 + 2 = 4).

Exercise 15 For each of the Smalltalk expressions below, fill in the answers:

---

3 + 4

- What is the receiver object?
- What is the message selector?
- What is/are the argument (s)?
- What is the message?
- What is the result returned by evaluating this expression?

Date today

- What is the receiver object?
- What is the message selector?
- What is/are the argument (s)?
- What is the message?
- What is the result returned by evaluating this expression?

---

anArray at: 1 put: 'hello'

- What is the receiver object?
- What is the message selector?
- What is/are the argument (s)?
- What is the message?
- What is the result returned by evaluating this expression?

Exercise 16 What kind of object does the literal expression 'Hello, Dave' describe?

Exercise 17 What kind of object does the literal expression #Node1 describe?

Exercise 18 What kind of object does the literal expression #(1 2 3) describe?

Exercise 19 What can one assume about a variable named Transcript?

Exercise 20 What can one assume about a variable named rectangle?

Exercise 21 Examine the following expression:

```
| anArray |
anArray := #('first' 'second' 'third' 'fourth').
anArray at: 2
```

What is the resulting value when it is evaluated (^ means return)? What happens if you remove the ^. Explain

Exercise 22 Which sets of parentheses are redundant with regard to evaluation of the following expressions:

```
((3 + 4) + (2 * 2) + (2 * 3))

(x isZero)
  ifTrue: [....]
(x includes: y)
  ifTrue: [....]
```

Exercise 23 Guess what are the results of the following expressions

```
6 + 4 / 2
1 + 3 negated
1 + (3 negated)
2 raisedTo: 3 + 2
2 negated raisedTo: 3 + 2
```

Exercise 24 Examine the following expression:

```
25@50
```

- What is the receiver object?
- What is the message selector?
- What is/are the argument (s)?
- What is the message?
- What is the result returned by evaluating this expression?

Exercise 25 Examine the following expression and write down the sequence of steps that the Smalltalk system would take to execute the following expression:

```
Date today daysInMonth
```

Exercise 26 Examine the following expression and write down the sequence of steps that the Smalltalk system would take to execute the following expression:

```
Transcript show: (45 + 9) printString
```

Exercise 27 Examine the following expression and write down the sequence of steps that the Smalltalk system would take to execute the following expression:

```
5@5 extent: 6.0 truncated @ 7
```

Exercise 28 During lecture, we saw how to write strings to the Transcript, and how the message printString could be sent to any non-string object to obtain a string representation. Now write a Smalltalk expression to print the result of 34 + 89 on the Transcript. Test your code !

Exercise 29 Examine the block expression:

```
| anArray sum |
sum := 0.
anArray := #(21 23 53 66 87).
anArray do: [:item | sum := sum + item].
sum
```

What is the final result of sum ? How could this piece of code be rewritten to use explicit array indexing (with the method at:) to access the array elements[1]? Test your version. Rewrite this code using inject:into:

[1] Note this is how you would proceed with Java or C++

3

Counter Example

Main Author(s): Bergel, Ducasse, Wuyts

3.1 A Simple Counter

We want you to implement a simple counter that follows the small example given below. Please note that we will ask you to define a test for this example.

```
| counter |
counter := SimpleCounter new.
counter increment; increment.
counter decrement.
counter value = 1
```

3.2 Creating your own class

In this part you will create your first class. In traditional Smalltalk environments a class is associated with a category (a folder containing the classes of your project).

When we are using Store, categories are replaced by packages. Therefore in VisualWorks with Store you define a package and define your class within this package. The steps we will do are the same ones every time you create a class, so memorize them well. We are going to create a class SimpleCounter in a package called DemoCounter. Figure **??** shows the result of creating such a package. Note that you all will be versioning your code in the Store database -with the rest of the students of the lecture-, so every package (each one belonging to different group of students) must have a different name. Therefore you should prefix them with your initials or group name.

3.2.1 Creating a Package

In the System Browser, click on the line Local Image located in the left-most upper pane (left button of the mouse) and select New ...Package. The system will ask you a name. You should write DemoCounter, postfixed with your initials or group name. This new package will be created and added to the list (see Figure **??**).

With the package selected, as shown in Figure **??**, you can edit its properties by clicking the properties tab of the editor. Properties you will likely have to set one day are the dependencies on other packages (for example, when you subclass a class from another package), post-load actions (an expression that is executed after loading that package from Store, for example to initialize something) and pre-unload actions (an expression that is executed just before unloading a package from your image, for example to close any windows from an application). In the context of this exercise we do not need any of this, so leave the properties alone for now.

3.2.2 Creating a Class

Creating a class requires five steps. They consist basically of editing the class definition template to specify the class you want to create. *Before you begin, make sure that only the package* DemoCounter *is selected.* (See Figure **??**)

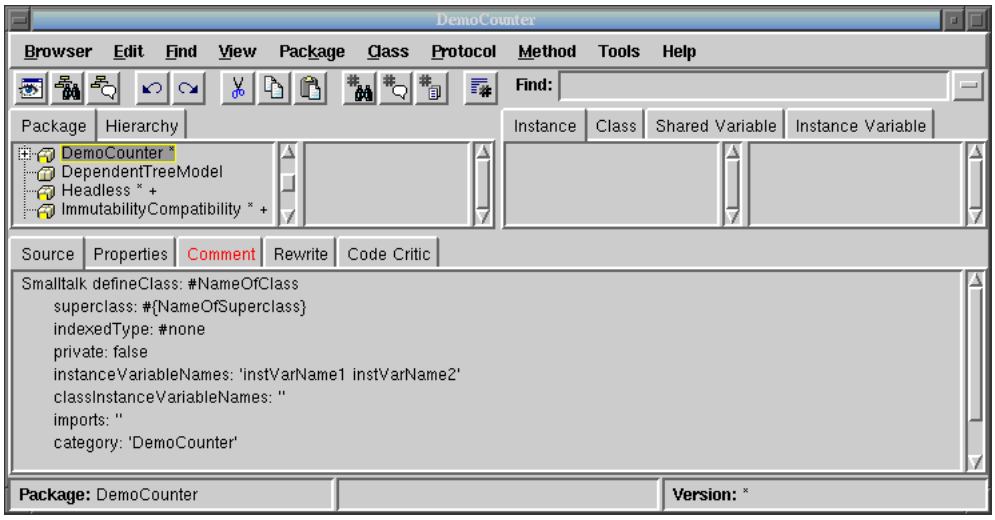

Figure 3.1: Your package is created.

1. **Superclass Specification**. First, you should replace the word NameOfSuperclass with the word Core.Object. Thus, you specify the superclass of the class you are creating. Note that this is not always the case that Object is the superclass, since you may to inherit behavior from a class specializing already Object.

2. **Class Name**. Next, you should fill in the name of your class by replacing the word NameOfClass with the word SimpleCounter. Take care that the name of the class starts with a capital letter and that you do not remove the # sign in front of NameOfClass.

3. **Instance Variable Specification**. Then, you should fill in the names of the instance variables of this class. We need one instance variable called value. You add it by replacing the words *instVarName1* and *instVarName2* with the word value. Take care that you leave the string quotes!

4. **Class Variable Specification**. As we do not need any class variable make sure that the argument for the class instance variables is an empty string (classInstanceVariableNames: '').

5. **Compilation**. That's it! We now have a filled-in class definition for the class SimpleCounter. To define it, we still have to **compile** it. Therefore, select the **accept** option from the operate menu (right-click button of the mouse). The class SimpleCounter is now compiled and immediately added to the system.

As we are disciplined developers, we provide a comment to SimpleCounter class by clicking **Comment** tab of the class definition (in the figure **??** the **Comment** is highlighted). You can write the following comment:

SimpleCounter is a concrete class which supports incrementing
and decrementing a counter.

Instance Variables:

value <Integer>

Select **accept** to store this class comment in the class.

3.3 Defining protocols and methods

In this part you will use the System Browser to learn how to add protocols and methods.

3.3.1 Creating and Testing Methods

The class we have defined has one instance variable value. You should remember that in Smalltalk, everything is an object, that instance variables are private to the object and that the only way to interact with an object is by sending messages to it.

Therefore, there is no other mechanism to access the instance variables from outside an object than sending a message to the object. What you can do is to define messages that return the value of the instance variable of a class. Such methods are called **accessors**, and it is a common practice to always define and use them. We start to create an accessor method for our instance variable value.

Remember that every method belongs to a protocol. These protocols are just a group of methods without any language semantics, but convey important navigation information for the reader of your class. Although protocols can have any name, Smalltalk programmers follow certain conventions for naming these protocols. If you define a method and are not sure what protocol it should be in, first go through existing code and try to find a fitting name.

An important remark: *Accessors* can be defined in protocols accessing or private. Use the accessing protocol when a client object (like an interface) really needs to access your data. Use private to clearly state that no client should use the accessor. This is purely a convention. There is no way in Smalltalk to enforce access rights like *private* in C++ or Java. To emphasize that objects are not just data structure but provide services that are more elaborated than just accessing data, put your accessors in a private protocol. As a good practice, if you are not sure then define your accessors in a private protocol and once some clients really need access, create a protocol accessing and move your methods there. Note that this discussion does not seem to be very important in the context of this specific simple example. However, this question is central to the notion of object and encapsulation of the data. An important side effect of this discussion is that you should always ask yourself when you, as a client of an object, are using an accessor if the object is really well defined and if it does not need extra functionality.

Exercise 30 Decide in which protocol you are going to put the accessor for value. We now create the accessor method for the instance variable value. Start by selecting the class DemoCounter in a browser, and make sure the **Instance** tab is selected (in the figure **??**, the **Instance** tab is in the middle of the window). Create a new protocol clicking the right-button of the mouse on the pane of methods categories, and choosing New, and give a name. Select the newly created protocol. Then in the bottom pane, the edit field displays a method template laying out the default structure of a method. Replace the template with the following method definition:

```
value
    "return the current value of the value instance variable"

    ^value
```

This defines a method called value, taking no arguments, having a method comment and returning the instance variable value. Then choose **accept** in the operate menu (right button of the mouse) to compile the method. You can now test your new method by typing and evaluating the next expression in a Workspace, in the Transcript, or any text editor SimpleCounter new value.

To use a workspace, click on the 'noteblock' icon of the launcher (last icon shown in Figure **??**).

This expression first creates a new instance of SimpleCounter, and then sends the message value to it and retrieves the current value of value. This should return nil (the default value for noninitialised instance variables; afterwards we will create instances where value has a reasonable default initialisation value).

Exercise 31 Another method that is normally used besides the *accessor* method is a so-called *mutator* method. Such a method is used to *change* the value of an instance variable from a client. For example, the next expression first creates a new SimpleCounter instance and then sets the value of value to 7:

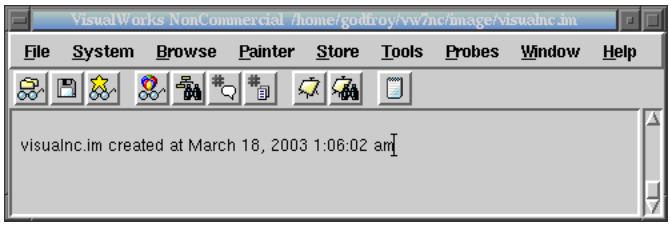

Figure 3.2: The Launcher of VisualWorks.

```
SimpleCounter new value: 7
```

This mutator method does not currently exist, so as an exercise write the method value: such that, when invoked on an instance of SimpleCouter, the value instance variable is set to the argument given to the message. Test your method by typing and evaluating the expression above.

Exercise 32 Implement the following methods in the protocol operations.

```
increment
   self value: self value + 1
decrement
   self value: self value - 1
```

Exercise 33 Implement the following methods in the protocol printing

```
printOn: aStream
   super printOn: aStream.
   aStream nextPutAll: ' with value: ',
   self  value printString.
   aStream cr.
```

Now test the methods increment and decrement but pay attention that the counter value is not initialized. Try:

```
SimpleCounter new value: 0; increment ; value.
```

Note that the method printOn: is used when you print an object or click on self in an inspector.

3.3.2 Adding an instance creation method

When we create a new instance of the class SimpleCounter using the message new, we would like to obtain a well initialized instance. To do so, we need to override the method new to add a call to an initialization method (invoking an initialize method is a very common practice! Ask for the senders of initialize). Notice that new is always sent to a class. This means that we have to define the new method on the *class side*, not on the *instance side*. To define an instance creation method like the method new you should be on the class side, so you click on the **Class** tab (See in the figure **??**, the **Class** is situated in the same level as the **Instance** tab).

Exercise 34 Define a new protocol called instance creation, and implement the method new as follows:

```
new "Create and return an initialized instance of SimpleCounter" |newInstance
|newInstance := super new. newInstance initialize. ^newInstance
```

This code returns a new and well initialized instance. We first create a new instance by calling the normal creation method (super new), then we assign this new created instance into the temporary variable called newInstance. Then we invoke the initialize method on this new created instance via the temporary variable and finally we return it.

Note that the previous method body is strictly equivalent to the following one. Try to understand why they are equivalent.

```
new "Create and return an initialized instance of SimpleCounter"
^super new initialize
```

3.3.3 Adding an instance initialization method

Now we have to write an initialization method that sets a default value to the value instance variable. However, as we mentioned the initialize message is sent to the newly created instance. This means that the initialize method should be defined at the instance side as any method that is sent to an instance of SimpleCounter like increment and decrement. The initialize method is responsible to set up the instance variable default values.

Therefore at the instance side, you should create a protocol initialize-release, and create the following method (the body of this method is left blank. Fill it in!).

```
initialize
  "set the initial value of the value to 0"
```

Remark. As we already mentioned, the initialize method is not automatically invoked by the method new. We had to override the method new to call the initialize method. This is a weakness of the Smalltalk libraries, so you should always check if the class that you are creating inherits from a new method that implements the call to the initialize method. It is a good practice to add such a calling structure (new calling initialize) in the root of the your class hierarchy. This way you share the calling structure and are sure that the initialize method is always called for all your classes.

Now create a new instance of class SimpleCounter. Is it initialized by default? The following code should now work without problem:

```
SimpleCounter new increment
```

3.3.4 Another instance creation method

If you want to be sure that you have really understood the distinction between instance and class methods, you should now define a different instance creation method named withValue:. This method receives an integer as argument and returns an instance of SimpleCounter with the specified value. The following expression should return 20.

```
(SimpleCounter withValue: 19) increment ; value
```

A Difficult Point Let us just think a bit! To create a new instance we said that we should send messages (like new and basicNew) to a class. For example to create an instance of SimpleCounter we sent new to SimpleCounter. As the classes are also objects in Smalltalk, they are instances of other classes that define the structure and the behavior of classes. One of the classes that represents classes as objects is Behavior. Browse the class Behavior. In particular, Behavior defines the methods new and basicNew that are responsible of creating new instances. If you did not redefine the new message locally to the class of SimpleCounter, when you send the message new to the class SimpleCounter, the new method executed is the one defined in Behavior. Try to understand why the methods *new* and *basicNew* are on the instance side on class Behavior while they are on the class side of your class.

3.4 Saving your Work

Several ways to save your work exist: You can

- Save the class by clicking on it and selecting the fileout menu item.
- Use the Monticello browser to save a package

To save our work, simply publish your package. This will open a dialog where you can give a comment, version numbers and blessing. After this is set, you can press Publish and your package will be stored in the database of Store. From then on, other people can load it from there, in the same way that you would use cvs or other multi-user versioning systems. Saving the image is also a way to save your working environment, but publishing it saves the code in the database. You can of course both publish your package (so that other people can load it, and that you can compare it with other versions, etc.) *and* save your image (so that next time that you start your image you are in the same working environment).