1

Counter Example

Main Author(s): Bergel, Ducasse, Wuyts

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
```

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.

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.

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 **??**)

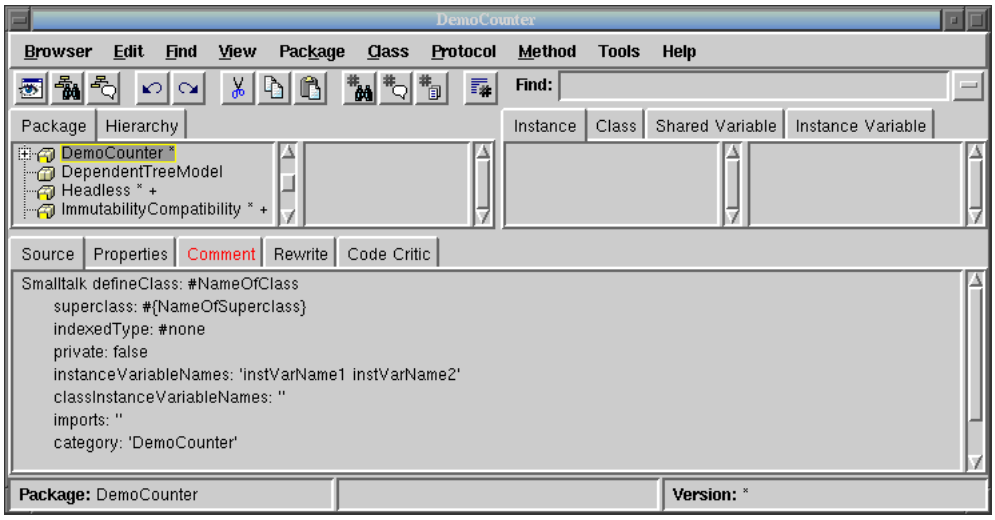

Figure 1.1: Your package is created.

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.

Defining protocols and methods

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

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 0 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 1 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:

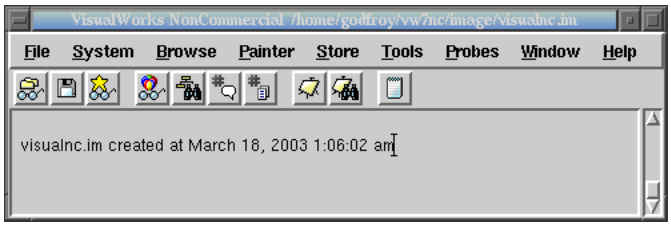

Figure 1.2: The Launcher of VisualWorks.

```
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 2 Implement the following methods in the protocol operations.

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

Exercise 3 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.

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 4 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
```

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 does not have specific and predefined semantics; it is just a convention to name the method that 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
```

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.

Saving your Work

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).