Advanced Object Oriented Design with Smalltalk

Gabriela Arévalo
Alexandre Bergel
Prof. Dr. Stéphane Ducasse
Michele Lanza
Dr. Roel Wuyts

Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland
{arevalo, bergel, ducasse, lanza, wuyts}@iam.unibe.ch
http://www.iam.unibe.ch/∼scg/Teaching/Smalltalk03/

June 19, 2015

Exercise 1

Basics of the VisualWorks Smalltalk Environment

1.1 Starting up

Smalltalk is an interpreted language: the source code is translated to Smalltalk-byte codes, which is 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 interpreting the byte codes.)

When looking at VisualWorks Smalltalk, there are three important files:

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

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.

On MacIntosh, to open an image:

- 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 Solaris: 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.

After opening the image, and thus starting a Smalltalk session, you see two windows : the VisualWorks 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 a new image for this lesson.

- select Save As... in the file-menu
- when the system prompts you for the name for the new image, you type lesson.
- 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. VisualWorks (or rather the older precedent) 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

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

Exercise: 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: 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: 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 ?

1.3 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 any kind of VisualWorks application windows. Experiment with it until you feel comfortable with it.

The Window menu. To resize a window on the Macintosh, click in the lower right corner while holding the alt-button. On a PC or Sun, you resize VisualWorks windows the same way as any other window.

Once you have opened your Workspace window, bring up the window menu, and experiment with it. Note that this menu is the same for each window, and contains the very basic window manipulations.

1.4 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 evaluation result.

print it: prints the result of the evaluation 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 evaluation.

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

Exercise: Print the result of 3+4 **Exercise:** 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 VisualWorks 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: Click on self in the inspector. What do you get ? Does it resemble the result shown by printstring ?

Exercise: 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: 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:** Type in the Workspace the following expression:

Time now, and inspect it. Have a look at self and the instance variables.

Exercise: Type in the Workspace the following expression: Time dateAndTimeNow. This tells Visual-Works 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 at the bottom of the Launcher 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: 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 ; ?

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

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

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

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

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

Exercise: 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: 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: 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: 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: 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: 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: 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: 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: 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++

Exercise 3

Counter Example

This document provides the initial exercise you should do to be familiar with Smalltalk syntax and VisualWorks 7.

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`. 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`, prefixed with your initials or group name. This new package will be created and added to the list (see Figure 3.1).

With the package selected, as shown in Figure 3.1, 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.

With Namespace

In VisualWorks you can also define your own namespace. If you do so you will have to define your namespace (in the *class* menu of the browser) before you can define the class in this new namespace. For now we suggest you to use the default namespace called `Smalltalk`. Note that classes (or methods or namespaces) can be moved to other namespaces or packages anytime.

Creating a Class

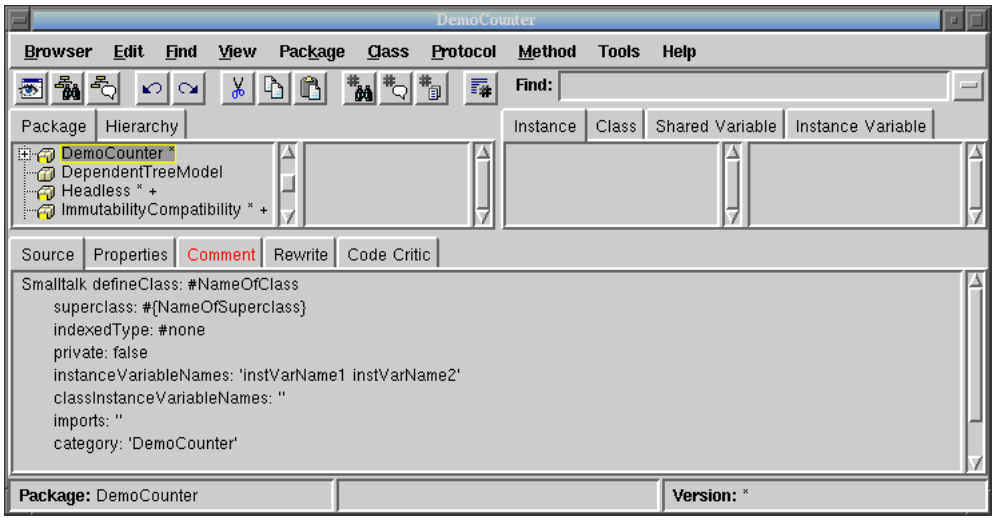

Figure 3.1: Your package is created.

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 3.1)

1. **Superclass Specification**. First, you should replace the word `NameOfSuperclass` with the word `Core.Object`[1]. Thus, you specify the superclass of the class you are creating.

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 3.1 the **Comment** is highlighted). You can write the following comment:

[1] We will see when we will be building a user interface for this object that we will change its superclass

```
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: 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 3.1, 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 last icon of the launcher as shown in Figure 3.2.

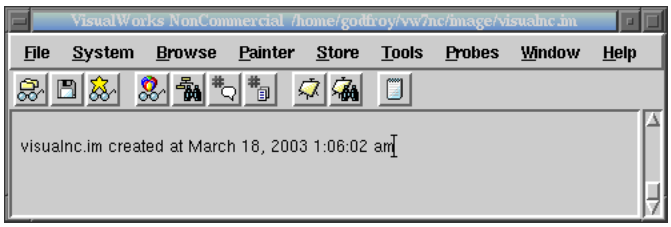

Figure 3.2: The Launcher of VisualWorks.

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: 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:

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

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

Exercise: 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 3.1, the **Class** is situated in the same level as the **Instance** tab).

Exercise: 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.

SUnit

Download the tutorial SUnit explained from http://www.iam.unibe.ch/∼ducasse/WebPages/Books.html and define a `TestCase` with several tests for the `SimpleCounter` class. To open the test runner execute

```
TestRunner open
```

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).

Exercise 4

A Simple Application: A LAN simulation

Basic LAN Application

The purpose of this exercise is to create a basis for writing future OO programs. We use the knowledge of the previous exercise to create classes and methods. We work on an application that simulates a simple **Local Area Network (LAN)**. We will create several classes: `Packet, Node, Workstation`, and `PrintServer`. We start with the simplest version of a LAN, then we will add new requirements and modify the proposed implementation to take them into account.

Creating the Class `Node`

The class `Node` will be the root of all the entities that form a `LAN`. This class contains the common behavior for all nodes. As a network is defined as a linked list of nodes, a Node should always know its next node. A node should be uniquely identifiable with a name. We represent the name of a node using a symbol (because symbols are unique in Smalltalk) and the next node using a node object. It is the node responsibility to send and receive packets of information.

```
Node inherits from Object
Collaborators: Node and Packet
Responsibility:
name (aSymbol) - returns the name of the node.
hasNextNode - tells if a node has a next node.
accept: aPacket - receives a packet and process it.
By default it is sent to the next node.
send: aPacket - sends a packet to the next node.
```

Exercise: Create a new package `LAN`, and create a subclass of `Object` called `Node`, with two instance variables: `name` and `nextNode`.

Exercise: Create accessors and mutators for the two instance variables. Document the mutators to inform users that the argument passed to `name:` should be a Symbol, and the arguments passed to `nextNode` should be a Node. Define them in a `private` protocol. Note that a node is identifiable via its name. Its name is part of its public interface, so you should move the method name from the `private` protocol to the `accessing` protocol (by drag'n'drop).

Exercise: Define a method called `hasNextNode` that returns whether the node has a next node or not.

Exercise: Create an instance method `printOn:` that puts the class name and name variable on the argument `aStream`. Include my next node's name ONLY if there is a next node (Hint: look at the method `printOn:` from previous exercises or other classes in the system, and consider that the instance variable `name` is a symbol and `nextNode` is a node). The expected `printOn:` method behavior is described by the following code:

```
(Node new
   name: #Node1 ;
   nextNode: (Node new name: #PC1)) printString

Node named: Node1 connected to: PC1
```

Exercise: Create a **class** method `new` and an **instance** method `initialize`. Make sure that a new instance of `Node` created with the new method uses `initialize` (see previous exercise). Leave `initialize` empty for now (it is difficult to give meaningful default values for the `name` and `nextNode` of `Node`. However, subclasses may want to override this method to do something meaningful).

Exercise: A node has two basic messages to send and receive packets. When a packet is sent to a node, the node has to accept the packet, and send it on. Note that with this simple behavior the packet can loop infinitely in the LAN. We will propose some solutions to this issue later. To implement this behavior, you should add a protocol `send-receive`, and implement the following two methods -in this case, we provide some partial code that you should complete in your implementation:

```
accept: thePacket
 "Having received the packet, send it on. This is the default
behavior My subclasses will probably override me to do
something special"

    ...

send: aPacket
"Precondition: self have a nextNode"

"Display debug information in the Transcript, then
send a packet to my following node"

 Transcript show:
   self name printString,
      ' sends a packet to ',
      self nextNode name printString; cr.
...
```

Creating the Class `Packet`

A packet is an object that represents a piece of information that is sent from node to node. So the responsibilities of this object are to allow us to define the originator of the sending, the address of the receiver and the contents.

```
Packet inherits from Object
Collaborators: Node
Responsibility:
addressee returns the addressee of the node to which
the packet is sent.
contents - describes the contents of the message sent.
originator - references the node that sent the packet.
```

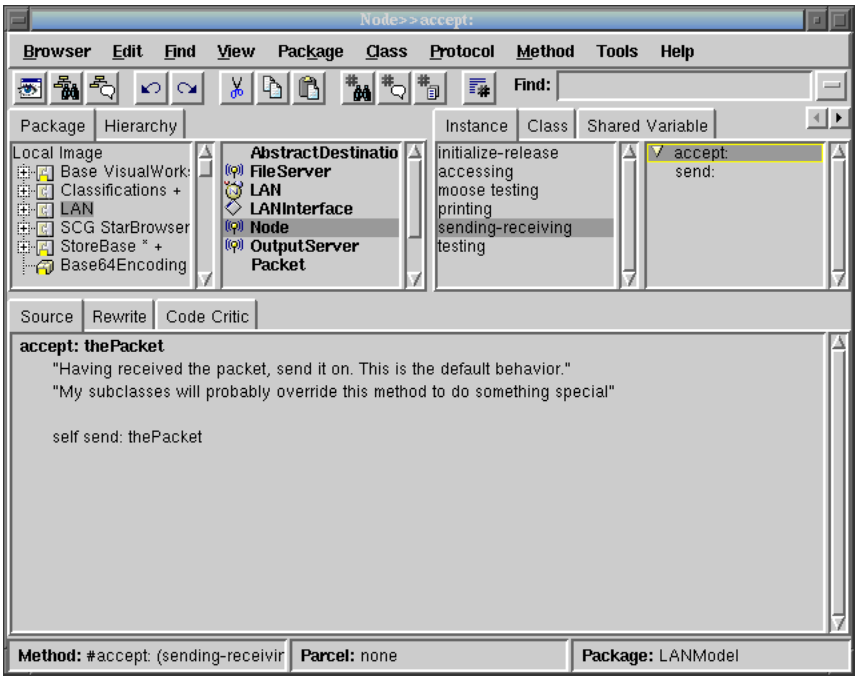

Figure 4.1: Definition of `accept:` method

Exercise: In the `LAN`, create a subclass of `Object` called `Packet`, with three instance variables: `contents`, `addressee`, and `originator`. Create accessors and mutators for each of them in the `accessing` protocol (in that particular case the accessors represents the public interface of the object). The addressee is represented as a symbol, the contents as a string and the originator has a reference to a node.

Exercise: Define the method `printOn: aStream` that puts a textual representation of a packet on its argument `aStream`.

Creating the Class `Workstation`

A workstation is the entry point for new packets onto the LAN network. It can originate packet to other workstations, printers or file servers. Since it is kind of network node, but provides additional behavior, we will make it a subclass of `Node`. Thus, it inherits the instance variables and methods defined in `Node`. Moreover, a workstation has to process packets that are addressed to it.

```
Workstation inherits from Node
Collaborators: Node, Workstation
and Packet
Responsibility: (the ones of node)
originate: aPacket - sends a packet.
accept: aPacket - perform an action on packets sent to the
workstation (printing in the transcript). For the other
packets just send them to the following nodes.
```

Exercise: In the package `LAN` create a subclass of `Node` called `Workstation` without instance variables.

Exercise: Define the method `accept: aPacket` so that if the workstation is the destination of the packet, the following message is written into the Transcript. Note that if the packets are not addressed to the workstation they are sent to the next node of the current one.

```
(Workstation new
    name: #Mac ;
    nextNode: (Printer new name: #PC1))
          accept: (Packet new addressee: #Mac)

A packet is accepted by the Workstation Mac
```

Hints: To implement the acceptance of a packet not addressed to the workstation, you could copy and paste the code of the `Node` class. However this is a bad practice, decreasing the reuse of code and the "Say it only once" rules. It is better to invoke the default code that is currently overriden by using `super`.

Exercise: Write the body for the method `originate:` that is responsible for inserting packets in the network in the method protocol `send-receive`. In particular a packet should be marked with its originator and then sent.

```
originate: aPacket
 "This is how packets get inserted into the network.
  This is a likely method to be rewritten to permit
  packets to be entered in various ways. Currently,
  I assume that someone else creates the packet and
  passes it to me as an argument."
 ...
```

Creating the class `LANPrinter`

Exercise: With nodes and workstations, we provide only limited functionality of a real LAN. Of course, we would like to do something with the packets that are travelling around the LAN. Therefore, you will now create a class `LanPrinter`, a special node that receives packets addressed to it and prints them (on the Transcript). Note that we use the name LanPrinter to avoid confusion with the existing class `Printer` in the namespace Smalltalk.Graphics (so you could use the name Printer in your namespace or the Smalltalk namespace if you really wanted to). Implement the class LanPrinter.

```
LanPrinter inherits from Node
Collaborators: Node and Packet
Responsibility:
accept: aPacket - if the packet is addressed to the
printer, prints the packet contents else sends the packet
to the following node.
print: aPacket - prints the contents of the packet
(into the Transcript for example).
```

Simulating the LAN

Implement the following two methods on the class side of the class `Node`, in a protocol called `examples`. But take care: the code presented below has **some bugs** that you should find and fix!.

```
simpleLan
  "Create a simple lan"
  "self simpleLan"

 | mac pc node1 node2 igPrinter |
```

```
"create the nodes, workstations, printers and fileserver"
mac := Workstation new name: #mac.
pc := Workstation new name: #pc.
node1 := Node new name: #node1.
node2 := Node new name: #node2.
node3 := Node new name: #node3.
igPrinter := Printer new name: #IGPrinter.

"connect the different nodes."
"I make following connections:
               mac -> node1 -> node2 ->
               igPrinter -> node3 -> pc -> mac"
mac nextNode: node1.
node1 nextNode: node2.
node2 nextNode: igPrinter.
igPrinter nextNode: node3.
node3 nextNode: pc.
pc nextNode: mac.

"create a packet and start simulation"
packet := Packet new
            addressee: #IGPrinter;
            contents: 'This packet travelled around
to the printer IGPrinter.

mac originate: packet.
```

---

anotherSimpleLan

```
"create the nodes, workstations and printers"

|mac pc node1 node2 igPrinter node3 packet |
mac:= Workstation new name: #mac.

pc := Workstation new name:#pc.

node1 := Node new name: #node1.

node2 := Node new name: #node2.

node3 := Node new name: #node3.

igPrinter := LanPrinter new name: #IGPrinter.

"connect the different nodes." "I make the following connections:
               mac -> node1 -> node2 ->
        igPrinter -> node3 -> pc -> mac"
mac nextNode: node1.

node1 nextNode: node2.

node2 nextNode:igPrinter.
```

```
igPrinter nextNode: node3.

node3 nextNode: pc.

pc nextNode: mac.

"create a packet and start simulation''
packet := Packet new
            addressee: #anotherPrinter;
            contents: 'This packet travels around
            to the printer IGPrinter'.
pc originate: packet.
```

As you will notice the system does not handle loops, so we will propose a solution to this problem in the future. To break the loop, use either **Ctrl-Y** or **Ctrl-C**, depending on your VisualWorks version.

Creating the Class `FileServer`

Create the class `FileServer`, which is a special node that saves packets that are addressed to it (You should just display a message on the Transcript).

```
FileServer inherits from Node
Collaborators: Node and Packet
Responsibility:
accept: aPacket - if the packet is addressed to the
file server save it (Transcript trace) else send the
packet to the following node.
save: aPacket - save a packet.
```

Exercise 5

Fundamentals on the Semantics of Self and Super

This lesson wants you to give a better understanding of self and super.

5.1 self

When the following message is evaluated:

```
aWorkstation originate: aPacket
```

The system starts to look up the method originate: starts in the class of the message receiver: Workstation. Since this class defines a method originate:, the method lookup stops and this method is executed.

Following is the code for this method:

```
Workstation>>originate: aPacket

   aPacket originator: self.
   self send: aPacket
```

1. It first sends the message originator: to an instance of Packet with as argument self which is a pseudo-variable that represents the receiver of originate: method. The same process occurs. Originator: is looked up into the class Packet. As Packet defines a method named originator:, the method lookup stops and the method is executed. As shown below the body of this method is to assign the value of the first argument (aNode) to the instance variable originator. Assignment is one of the few constructs of Smalltalk. It is not realized by a message sent but handle by the compiler. So no more message sends are performed for this part of originator:.

```
   Packet>>originator: aNode

         originator := aNode
```

2. In the second line of the method originate:, the message send: thePacket is sent to self. self represents the instance that receives the originate: message. **The semantics of self specifies that the method lookup should start in the class of the message receiver.** Here Workstation. Since there is no method send: defined on the class Workstation, the method lookup continues in the superclass of Workstation: Node. Node implements send:, so the method lookup stops and send: is invoked

```
Node>>send: thePacket

     self nextNode accept: thePacket
```

The same process occurs for the expressions contained into the body of the method send:.

5.2 super

Now we present the difference between the use of self and super. Self and super are both pseudo-variables that are managed by the system (compiler). They both represents the receiver of the message being executed. However, there is no use to pass super as method argument, self is enough for this.

The main difference between self and super is their semantics regarding method lookup.

- The semantics of self is to start the method lookup **into the class of the message receiver and to continue in its superclasses.**

- The semantics of super is to start the method look into **the superclass of class in which the method being executed was defined and to continue in its superclasses.**. Take care the semantics is **NOT** to start the method lookup into the superclass of the receiver class, the system would loop with such a definition (see exercise 1 to be convinced). Using super to invoke a method allows one to invoke overridden method.

Let us illustrate with the following expression: the message accept: is sent to an instance of Workstation.

```
aWorkstation accept: (Packet new addressee: #Mac)
```

As explained before the method is looked up into the class of the receiver, here Workstation. The method being defined into this class, the method lookup stops and the method is executed.

```
Workstation>>accept: aPacket

    (aPacket addressee = self name)
        ifTrue:[Transcript show: 'Packet accepted', self name asString]
        ifFalse: [super accept: aPacket]
```

Imagine that the test evaluates to false. The following expression is then evaluated.

```
super accept: aPacket
```

The method accept: is looked up in the superclass of the class in which the containing method accept: is defined. Here the containing method is defined into Workstation so the lookup starts in the superclass of Workstation: Node. The following code is executed following the rule explained before.

```
Node>>accept: aPacket

    self hasNextNode          ifTrue:[ self send: aPacket]
```

Remark. The previous example does not show well the vicious point in the super semantics: the method look into **the superclass of class in which the method being executed was defined and not in the superclass of the receiver class.**

You have to do the following exercise to prove yourself that you understand well the nuance.

Exercise: 1: Imagine now that we define a subclass of Workstation called AnotherWorkstation and that this class does NOT defined a method accept:. Evaluate the following expression with both semantics:

```
anAnotherWorkstation accept: (Packet new addressee: #Mac)
```

You should be convinced that the semantics of super change the lookup of the method so that the lookup (for the method via super) does NOT start in the superclass of the receiver class but in the superclass of the class in which the method containing the super. With the wrong semantics the system should loop.

Exercise 6

Object Responsibility and Better Encapsulation

6.1 Reducing the coupling between classes

To be a good citizen you as an object should follow as much as possible the following rules:

- Be private. Never let somebody else play with your data.
- Be lazy. Let do other objects your job.
- Be focused. Do only one main task.

While these guidelines are not really formal, one of the main consequences is that this is the responsibility of an object to provide a well defined interface protecting itself from its clients. The other consequence is that by delegating to other objects an object concentrates on a single task and responsibility. We now look how such guidelines can help us to provide better objects in our example.

6.1.1 Current situation

The interface of the packet class is really weak. It just provides free access to its data. The main impact of this weakness is the fact that the clients of the class Packet like Workstation relies on the internal coding of the Packet as shown in the first line of the following method.

```
 Workstation>>accept: aPacket

    aPacket addressee = self name
       ifTrue:[ Transcript show: 'A packet is accepted by the
Workstation ', self name asString]
       ifFalse: [super accept: aPacket]
```

As a consequence, if the structure of the class Packet would change, the code of its clients would have to change too. Generalizing such a bad practice would lead to system that are badly coupled and being really difficult to change to meet new requirements.

6.1.2 Solution.

This is the responsibility of a packet to say if the packet is addressed to a particular node or if it was sent by a particular node.

- Define a method named isAddressedTo: aNode in 'testing' protocol that answers if a given packet is addressed to the specified node.

- Define a method named isOriginatedFrom: aNode in 'testing' protocol that answers if a given packet is originated from the specified node.

Once these methods are defined, change the code of all the clients of the class Packet to call them.

6.2 A Question of Creation Responsibility

One of the problem with the previous approach for creating the nodes and the packets is the following:

it is the responsibility of the client of the objects to create them well-formed. For example, it is possible to create a node without specifying a name! This is a disaster for our LAN system (create an example method 3, and try it out). The same problem occurs with the packet: it is possible to create a packet without address nor contents.

We will find a solution to these problems.

Exercise: Define a class method named withName: in the class Node (protocol 'instance creation') that creates a new node and assign its name.

```
withName: aSymbol
....
```

Define a class method named withName:nextNode: in the class Node (protocol 'instance creation') that creates a new node and assign its name and the next node in the LAN

```
withName: aSymbol nextNode: aNode
....
```

Note that the first method can simply invoke the second one.

Define a class method named send:to*:* in the class Packet (protocol 'instance creation') that creates a new Packet with a contents and an address.

```
send: aString to: aSymbol

....
```

Now the problem is that we want to forbid the creation of non-well formed instances of these classes. To do so, we will simply redefine the creation method *new* so that it will raise an error.

Exercise: Rewrite the new method of the class Node and Packet as the following:

```
new

    self error: 'you should invoke the method... to create a...'
```

However, you have just introduced a problem: the instance creation methods you just wrote in exercise 11 will not work anymore, because they call *new*, and that calling results in an error ! The solution is to rewrite them such as

```
Node class>>withName: aSymbol nextNode: aNode
      ^ self basicNew initialize name: aSymbol ; nextNode: aNode
```

Do the same for the instance creation methods in class Packet.

Exercise: Update and rerun your examples to make sure that your changes were correct.

Note that the previous code may break if a subclass specialize the nextNode: method does not return the instance. To protect ourslef from possible unexpected extension we add yourself that returns the receiver a the first cascaded message (here name:), here the newly created instance.

```
Node class>>withName: aSymbol nextNode: aNode
      ^ self basicNew initialize name: aSymbol ; nextNode: aNode ; yourself
```

6.3 Reducing the coupling between classes

To be a good citizen you as an object should follow as much as possible the following rules:

- Be private. Never let somebody else play with your private data.
- Be lazy. Let do other objects your job.
- Be focused. Do only one main task.

While these guidelines are not really formal, one of the main consequences is that this is the responsibility of an object to provide a well defined interface protecting itself from its clients. The other consequence is that by delegating to other objects an object concentrates on a single task and responsibility. We now look how such guidelines can help us to provide better objects in our example.

6.3.1 Current situation

The interface of the packet class is really weak. It just provides free access to its data. The main impact of this weakness is the fact that the clients of the class Packet like Workstation relies on the internal coding of the Packet as shown in the first line of the following method.

```
Workstation>>accept: aPacket

    aPacket addressee = self name

       ifTrue:[ Transcript show: 'A packet is accepted by the Workstation ', self name a
       ifFalse: [super accept: aPacket]
```

As a consequence, if the structure of the class Packet would change, the code of its clients would have to change too. Generalizing such a bad practice would lead to system that are badly coupled and being really difficult to change to meet new requirements.

6.3.2 Solution.

This is the responsibility of a packet to say if the packet is addressed to a particular node or if it was sent by a particular node.

- Define a method named isAddressedTo: aNode in 'testing' protocol that answers if a given packet is addressed to the specified node.
- Define a method named isOriginatedFrom: aNode in 'testing' protocol that answers if a given packet is originated from the specified node.

Once these methods are defined, change the code of all the clients of the class Packet to call them. You should note that a better interface encapsulates better the private data and the way they are represented. This allows one to locate the change in case of evolution.

6.4 A Question of Creation Responsibility

One of the problems with the first approach for creating the nodes and the packets is the following: it is the responsibility of the client of the objects to create them well-formed. For example, it is possible to create a node without specifying a name! This is a disaster for our LAN system, the node would never reachable, and worse the system would breaks because the assumptions that the name of a node is specified would not hold anymore (insert an anonymous node in Lan and try it out). The same problem occurs with the packet: it is possible to create a packet without address nor contents.

The solution to these problems is to give the responsibility to the objects to create well-formed instances. Several variations are possible:

- When possible, providing default values for instance variable is a good way to provide well-defined instances.
- It is also a good solution to propose a consistent and well-defined creation interface. For example one can only provide an instance creation method that requires the mandatory value for the instance and forbid the creation of other instances.

The class Packet. We investigate the two solutions for the Packet class. For the first solution, the principle is that the creation method (new) should invoke an initialize method. Implement this solution. Just remember that new is sent to classes (a class method) and that initialize is sent to instances (instance method). Implement the method new in a 'instance creation' protocol and initialize in a 'initialize-release' protocol.

```
Packet class>>new

...

Packet>>initialize
   ...
```

The only default value that can have a default value is contents, choose

```
contents = 'no contents'
```

Ideally if each LAN would contain a default trash node, the default address and originator would point to it. We will implement this functionality in a future lesson. Implement first your own solution.

Remarks and Analysis. Note that with this solution it would be convenient to know if a packet contents is the default one or not. For this purpose you could provide the method hasDefaultContents that tests that. You can implement it in a clever way as shown below:

Instead of writing:

```
Packet>>hasDefaultContents

 ^ contents = 'no contents'

Packet>>initialize
 ...

contents := 'no contents'
...
```

You should apply the rule: 'Say only once' and define a new method that returns the default content and use it as shown below:

```
Packet>>defaultContents

    ^ 'no contents'

Packet>>initialize
    ...

    contents := self defaultContent
    ...

Packet>>hasDefaultContent
    ^contents = self defaultContents
```

With this solution, we limit the knowledge to the internal coding of the default contents value to only one method. This way changing it does not affect the clients nor the other part of the class.

6.5 Proposing a creational interface

Packet. We now apply the second approach by providing a better interface for creating packet. For this purpose we define a new creation method that requires a contents and an address.

Define a **class** methods named send:to*:* and to: in the class Packet (protocol 'instance creation') that creates a new Packet with a contents and an address.

```
Packet class>>send: aString to: aSymbol

....

Packet class>>to: aSymbol

....
```

The class Node. Now apply the same techniques to the class Node. Note that you already implemented a similar schema that the default value in the previous lessons. Indeed by default instance variable value is nil and you already implemented the method hasNextNode that to provide a good interface.
Define a **class** method named withName: in the class Node (protocol 'instance creation') that creates a new node and assign its name.

```
Node class>>withName: aSymbol

....
```

Define a **class** method named withName:connectedTo: in the class Node (protocol 'instance creation') that creates a new node and assign its name and the next node in the LAN.

```
 Node class>>withName: aSymbol connectedTo: aNode

....
```

Note that if to avoid to duplicate information, the first method can simply invoke the second one.

6.6 Forbidding the Basic Instance Creation

One the last question that should be discussed is the following one: should we or not let a client create an instance without using the constrained interface? There is no general answer, it really depends on what we want to express. Sometimes it could be convenient to create an uncompleted instance for debugging or user interface interaction purpose.

Let us imagine that we want to ensure that no instance can be created without calling the methods we specified. We simply redefine the creation method new so that it will raise an error.

Rewrite the new method of the class Node and Packet as the following:

```
Node class>>new

                self error: 'you should invoke the method... to create a...'
```

However, you have just introduced a problem: the instance creation methods you just wrote in the previous exercise will not work anymore, because they call new, and that calling results in an error! Propose a solution to this problem.

6.6.1 Remarks and Analysis.

A first solution could be the following code:

```
 Node class>>withName: aSymbol connectedTo: aNode

      ^ super new initialize name: aSymbol ; nextNode: aNode
```

However, even if the semantics permits such a call using super with a different method selector than the containing method one, it is a bad practice. In fact it implies an implicit dependency between two different methods in different classes, whereas the super normal use links two methods with the same name in two different classes. It is always a good practice to invoke the own methods of an object by using self. This conceptually avoids to link the class and its superclass and we can continue to consider the class as self contained.

The solution is to rewrite the method such as:

```
Node class>>withName: aSymbol connectedTo: aNode

 ^  self basicNew initialize name: aSymbol ; nextNode: aNode
```

In Smalltalk there is a convention that all the methods starting with 'basic' should not be overridden. basicNew is the method responsible for always providing an newly created instance. You can for example browse all the methods starting with 'basic*' and limit yourself to Object and Behavior.

You can do the same for the instance creation methods in class Packet.

6.7 Protecting yourself from your children

The following code is a possible way to define an instance creation method for the class Node.

```
Node class>>withName: aSymbol

 ^ self new name: aSymbol
```

We create a new instance by invoking new, we assign the name of the node and then we return it. One possible problem with such a code is that a subclass of the class Node may redefine the method name: (for example to have a persistent object) and return another value than the receiver (here the newly created instance). In such a case invoking the method withName: on such a class would not return the new instance. One way to solve this problem is the following:

```
Node class>>withName: aSymbol

 |newInstance|
 newInstance := self new.
 NewInstance name: aSymbol.
 ^ newInstance
```

This is a good solution but it is a bit too much verbose. It introduces extra complexity by the the extra temporary variable definition and assignment. A good Smalltalk solution for this problem is illustrated by the following code and relies on the use of the yourself message.

```
Node class>>withName: aSymbol

   ^ self new name: aSymbol ; yourself
```

yourself specifies that the receiver of the first message involved into the cascade (name: here and not new) is return. Guess what is the code of the yourself method is and check by looking in the library if your guess is right.

Exercise 7

Hook and Template Methods

7.1 Providing Hook Methods

Current situation. The solution proposed for printing a Node displays the following information:

```
(Node withName: #Node1 connectedTo: (Node new name: #PC1)) printString

      Node named: Node1 connected to: PC1
```

A straightforward way to implement the printOn: method on the class Node is the following code:

```
Node>>printOn: aStream

 aStream nextPutAll: 'Node named: ', self name asString.
 self hasNextNode
  ifTrue:[ aStream nextPutAll: ' connected to: ', self nextNode name]
```

However, with such implementation the printing of all kinds of nodes is the same.

New Requirements. To help in the understanding of the LAN we would like that depending on the specific class of node we obtain a specific printing like the following ones:

```
 (Workstation withName: #Mac connectedTo: (LanPrinter withName:
#PC1) printString

   Workstation Mac connected to Printer PC1

(LanPrinter withName: #Pr1 connectedTo: (Node withName: #N1)
printString

 Printer Pr1 connected to Node N1
```

Define the method typeName that returns a string representing the name of the type of node in the 'printing' protocol. This method should be defined in Node and all its subclasses.

```
(LanPrinter withName: #PC1) typeName

      'Printer'

(Node withName: #N1) typeName
     'Node'
```

Define the method simplePrintString on the class Node to provide more information about a node as show below:

```
(Workstation withName: #Mac connectedTo: (LanPrinter withName:
#PC1)) simplePrint

'Workstation Mac'

(LanPrinter withName: #PC1) simplePrint

'Printer PC1'
```

Then modify the printOn: method of the class Node to produce the following output:

```
(self withName: #Mac connectedTo: (LanPrinter new name:
#PC1))

'Node Mac connected to Printer PC1'
```

Remark: The method typeName is called a *hook* method. This reflects the fact that it allows the subclasses to specialize the behavior of the superclass, here the printing of a all the different kinds of nodes. The method simplePrintString, even if in our case is rather simple, is called a template method. This name reflects the fact that the method specifies the context in which hook methods will be called and how they will fit into the template method to produce the expected result.

Note that for abstract classes hook methods can be abstract too, one other case the hook method can propose a default behavior.

The Smalltalk class library contains a lot of such hooks that allows an easy customization of the proposed behavior. The proposed requirement already exists in the system. Study the method printOn: on the class Object.

Exercise 8

Extending the LAN Application

This lesson uses the basic LAN-example and adds new classes and behaviour. Doing so, the design is extended to be more general and adaptive.

8.1 Handling Loops

When a packet is sent to an unknown node, it loops endlessly around the LAN. You will implement two solutions for this problem.

Solution 1. The first obvious solution is to avoid that a node resends a packet if it was the originator of the packet that it is sent. Modify the accept: method of the class Node to implement such a functionality.

Solution 2. The first solution is fragile because it relies on the fact that a packet is marked by its originator and that this node belongs to the LAN. A 'bad' node could pollute the network by originate packets with a anonymous name. Think about different solutions.

Among the possible solutions, two are worth to be further analyzed:

1. Each node keeps track of the packets it already received. When a packet already received is asked to be accepted again by the node, the packet is not sent again in the LAN. This solution implies that packet can be uniquely identified. Their current representation does not allow that. We could imagine to tag the packet with a unique generated identifier. Moreover, each node would have to remember the identity of all the packets and there is no simple way to know when the identity of treated node can be removed from the nodes.

2. Each packet keeps track of the node it visited. Every time a packet aarrived at a node, it is asked if it has already been here. This solution implies a modification of the communication between the nodes and the packet: the node must ask the status of the packet. This solution allows the construction of different packet semantics (one could imagine that packets are broadcasted to all the nodes, or have to be accepted twice). Moreover once a packet is accepted, the references to the visited nodes are simply destroyed with the packet so there is no need to propagate this information among the nodes.

We propose you to implement the second solution so that the class Packet provides the following interface (the new responsibilities are in bold).

```
Packet inherits from Object

Collaborators: Node

Responsibility:
```

```
addressee returns the addressee of the node to which the packet
is sent.

contents describes the contents of the message sent.

originator references the node that sent the packet.

isAddressedTo: aNode answers if a given packet is addressed to
the specified node. isOriginatedFrom: aNode answers if a given
packet is originated from the specified node.

isAcceptableBy: aNode answers if a packet is acceptable by
a node

hasBeenAcceptedBy: aNode tells a packet that it has been
accepted by a given node.
```

- New instance variable. A packet needs to keep track of the nodes it visited. Add a new instance variable called visitedNodes in the class Packet. We want to collect the visited nodes in a set. Browse the class Set and its superclass to find the function you need.

- Initialize the new instance variable. Modify the initialize methods of the class Packet so that the visitedNodes instance variable is initialized with an empty set.

- Node Acceptation Methods. In a protocol named 'node acceptation', define the method isAcceptableBy: and hasBeenAcceptedBy:.

- Test if your implementation works by sending a 'bad' node with a bad originator into the LAN.

8.2 Introducing a Shared Initialization Process

As you noticed, each time a new class is created that is not a subclass of Node we have to implement a new method whose the only purpose was to call the initialize method. We want to habe such a behavior specified only once and shared by all our Lan classes.

Define a class LanObject that inherits form Object, implements an instance method initialize and a class method new that automatically calls the initialize method on the newly created object and retrun it.

Then make all the classes that previously inherited from Object inherit from LanObject and check and remove if necessary if the unnecessary new methods.

8.3 Broadcasting and Multiple Addresses

Up to now, when a packet reaches a node it is addressed to, the packet is handled by the node and the transmission of the packet is terminated (because is not sent to the next node in the network). In this exercise, we want you to provide facilities for broadcasting. If a node handles a packet that is broadcasted, the packet must be sent to the next node in the LAN instead of terminating the connection. For example, broadcasting makes it possible to save the contents of the same packet on different fileservers of the LAN. First try to solve this problem, and implement it afterwards.
In the current LAN, a packet only has one addressee. This exercise wants to add packets that have multiple addressees. Propose a solution for this problem, and implement it afterwards.

8.4 Different Documents

Suppose we have several kinds of documents (ASCII and Postscript) and two kinds of LANPrinter in the LAN (LANASCIIPrinter and LANPostscriptPrinter). We then want to make sure that every printer prints the right kind of document. Propose a solution for this problem.

8.5 Logging Node

We want to add a logging facility: this means each time a packet is sent from a node, we want to identify the node and the packet. Propose and implement a solution. Hint: introduce a new subclass of Node between Node and its subclasses and specialize the send: method.

8.6 Automatic Naming

The name of a node have to be specified by its creator. We would like to have an automatic naming process that occurs when no name are specified. Note that the names should be unique. As a solution we propose you to use a counter, as this counter have to last over instance creations but still does not have any meaning for a particular node we use an instance variable of the class node.

Note that the NetworkManager could also be the perfect object to implement such a fonctionality. We also would like that all the printer names start with Pr. Propose a solution.

Workstation Mac connected to Printer PC

ApplicationModel already:

- defines basic application behavior (opening, running, closing, minimizing,)
- can open an application interface.

Our application subclass will have to implement

- the actual interface to be opened,
- behavior specific for your application,
- glue code, to glue together the models and View/Controllers.

Basically, our application class will thus implement application specific code, thereby linking the views/controllers used in the interface with the domain model. As explained in the lecture, models and view/controllers do not know each other directly, but will each talk to the applicationModel that actually glues everything together.

Building an application (i.e. constructing a subclass of ApplicationModel) thus boils down to two steps:

- building the interface
- programming the applicationModel

8.7 Building the interface

Now we will need to build the interface as pictured above. An interface contains several widgets (user interface elements), in this case an input field and two buttons. There are several kinds of widgets:

- data widgets (gather/display input): let the user enter information, or display information
- action widgets (invoke operations): buttons or menus, e.g. to increment or decrement the counter
- static widgets (organise/structure the interface): labels identifying other widgets for the user.

You build an interface by creating a visual specification of the contents and the layout. To do so, there are several steps to be taken:

1. opening a blank canvas,
2. painting the canvas with widgets chosen from a Palette,
3. setting properties for each widget and applying them to the canvas,
4. installing the canvas in an application model.

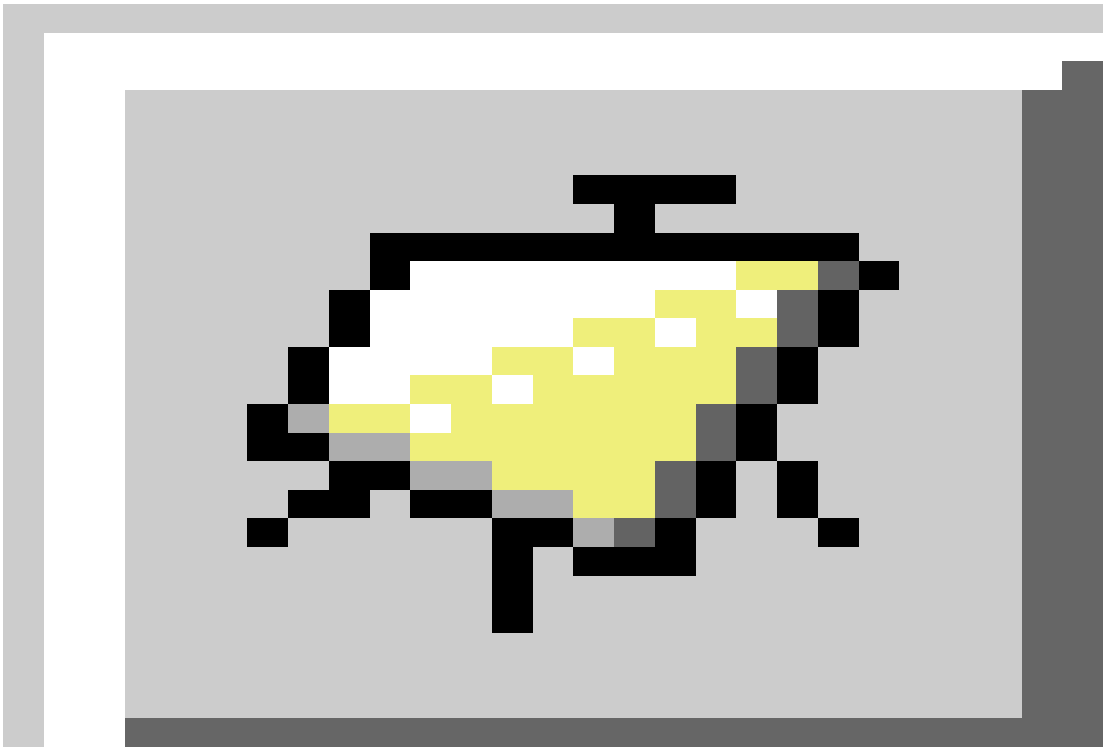

Figure 8.1: to do

Step 1: opening a blank canvas A canvas is the place where you visually edit the interface of the application. To open a blank canvas, use the canvas button (as shown above) on the VisualWorks Launcher, or select New Canvas in the Tools menu of the VisualWorks Launcher. VisualWorks will open a window containing an unlabeled canvas, a Canvas Tool, and a palette:

- the canvas tool provides you with the basic operations to build/install/define and open your application.
- the palette contains predefined widgets to use on the canvas.
- the unlabeled canvas is a visual representation for the window we are going to build.

Step 2: painting the canvas We will now paint the widgets such that our interface looks l ike the one pictured above. Basically this comes down on selecting widgets on the palette (by clicking them once), and putting them on the canvas (by clicking once again).

First, we will put an input field on our canvas. To do so, follow these steps:

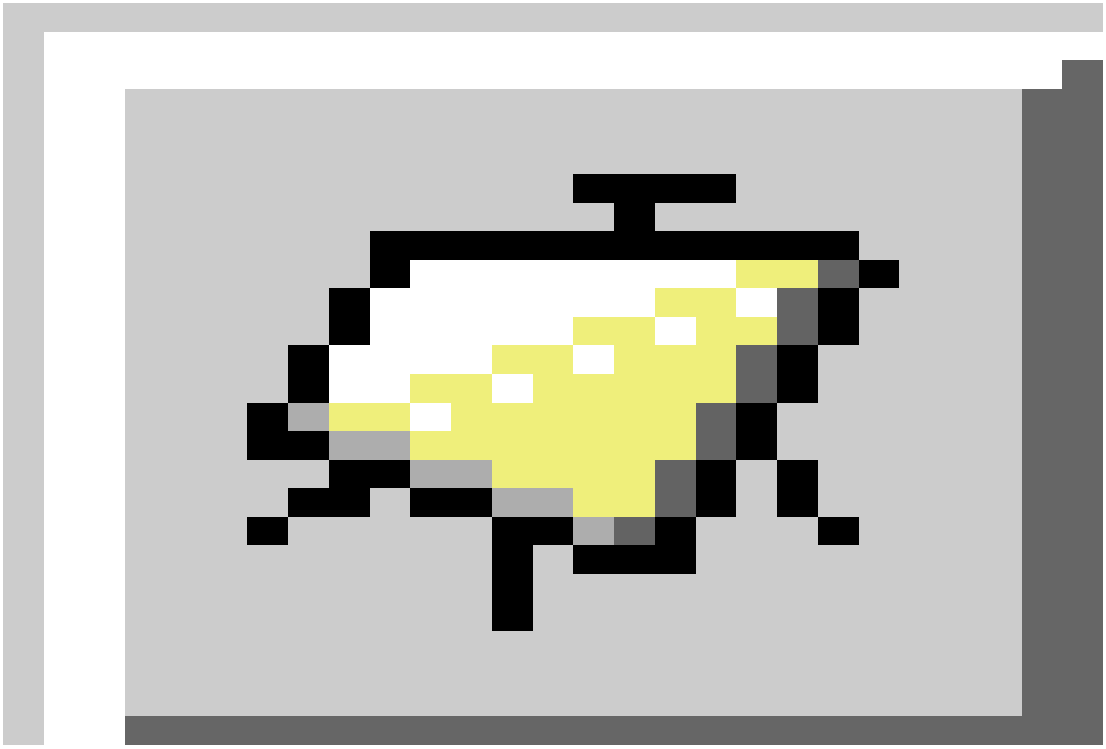

Figure 8.2: to do

1. verify that the single-selection button on the palette is active (it should look like the picture above). This enables you to paint a single copy of a widget on the canvas.
2. note that, when you select a widget on the palette, the name of the selected widget is shown in the indicator field at the bottom of the palette.
3. select the Input Field widget by clicking it once (if you select the wrong widget, select other widgets until the indicator field displays Input Field).
4. paint the input field by moving the mouse pointer to the canvas and clicking the select button once, positioning the widget, and clicking the select button a second time to place it on the canvas.

Once widgets are painted on the canvas, there are several editing operations that can be performed:

1. to select a widget: click it once
2. to deselect a widget: hold down the shift button while clicking on the selected widget, or click somewhere outside the widget
3. to resize a widget: select the widget, click on one of the handles of the widget (the black squares at the outside of the widget) and resize it.
4. to move a widget: select it, press the select button between the handles of the widget and move it
5. to cut/copy a widget: select the widget, bring up the operate menu and select cut/copy in the edit menu
6. to paste a widget (once you have cut/copied it): bring up the operate menu anywhere on a canvas, and select paste from the edit menu. The pasted widget is automatically placed at the same position as the widget that is cut/copied, and is automatically selected. You can now move it to another position.

Exercise: copy the one button widget that is currently on the canvas to make a second one, and position the two buttons according to the picture of the application.

Step 3: setting and applying properties of widgets We now have painted widgets, and are ready to set their properties. Properties define a variety of visual attributes, the nature of the data they use or display, and how that data is referenced by the application. We will now specify the different properties for our input field and buttons.

To display a widgets properties, we use the so-called Properties Tool. To open this tool, select the input field and click the Properties button on the Canvas Tool. The properties tool opens, and we are now ready to examine and change the properties that are available for an input field.

The properties are always arranged in a notebook, containing several pages. By clicking a tab of such a page, you select that page. Note that a Properties Tool does not belong to a particular canvas, or a particular widget. For example, if you now select one of the two buttons, the Properties Tool will change to allow you to view/change the properties for that widget.

We will now fill in the properties for the input field. Select the input field widget on the canvas. Go to the Basics page. Type in the aspect field: counterValue (always start aspect names with a small letter), and select Number in the type box. Apply these changes to the widget by pressing the apply button. You can now select the Details page. On this page, mark the check box Read-Only. Also apply these changes too.

Just to be a little bit less blind. The symbol that you typed in the aspect field corresponds to the selector of a method that we will create after. This method will return the model corresponding to the input field. Here as you will see the model will be value holder on the Number. This means that the valueHolder on a number will be the model (of the MVC pattern) for the inputField widget. The model of the InputField will be a ValueHolder, a basic object that send the message update to its dependent when it receives the message value:.

Exercise:: Set and apply the following properties for the left button:

Page Property Setting Basics Label increment Action increment Be Default checked The symbol associated with the Action button is the selector of a method of the application model that will be invoked when the button is pressed. Exercise: Set and apply these properties for the right button: Page Property Setting Basics Label decrement Action decrement Be Default unchecked Size as Default unchecked

Step 4 : Installing the canvas on an application model At any time in the painting process, you can save the canvas by installing it in an application model. Installing a canvas creates an interface specification, which serves as the applications blueprint for building an operational window. An interface specification is a description of an interface. Each installed interface specification is stored in (and returned by) a unique class method in the application model by default named windowSpec. Note that a same interface specification can be save with different names, more interesting a same set of widget can be saved in different positions under different method name.

You can think of a canvas as the VisualWorks graphical user interface for creating and editing an interface specification. Whereas a canvas is a graphical depiction of the windows contents and layout, an interface specification is a symbolic representation that an application model can interpret.

To install a canvas:

- click Install... in the canvas tool
- a dialog box comes up where you have to provide the name of the application model and the class method in which to install the canvas. Provide SimpleCounterApp as class name. Leave windowSpec (the default name of the class method where the interface specification is stored) as name of the selector. Press OK when finished.
- since your application model does not exist yet, you get another dialog box where you have to provide some information concerning your application model. Leave the name of the class, but provide DemoCounter as name for the category. Since we are creating a normal application (and not a dialog box or so), choose the application check box. Note that VisualWorks then fills in ApplicationModel as superclass. Leave this and select OK. Select a second time OK to close the first dialog box.

The canvas is now installed on the class SimpleCounterApp. Open a browser, go to the category DemoCounter, select the class switch to see the class methods, and note that there is a method windowSpec in a protocol called interface specs.

8.8 Programming the application model

As said in previous section, we now have to program our application model to: specify the interfaces appearance and basic behavior, supplement the applications basic behavior with application-specific behavior.

As said before there are several kinds of widgets: static widgets, action widgets and data widgets. Each of these kinds of widgets needs special programming care.

Static Widgets These are widgets like labels and separators that have no controller since they are just used to display something, and do not accept any kind of user input. No programming is required in the application model for this kind of widgets.

Action Widgets An action widget delegates an action to the application model from which it was built. Thus, when a user activates an action widget (for example, clicking the increment button), a message is sent by the widget to our application model (an instance of the class SimpleCounterApp). What message is sent is defined in the properties of the widget, in the Action field on the Basics page. Since we have defined the action property of the left button to be increment, this means that a message increment is sent to the application model when the user presses the increment button.

Data Widgets A data widget is designed to use an auxiliary object called a value model to manage the data it presents. (The value model play the M of the MVC pattern. This means that it propagates an update message to its dependent, the widget.) Thus, instead of holding on to the data directly it delegates this task to a value model:

- when a data widget accepts input from a user, it sends this input to its value model for storage,
- when a data widget needs to update its display, it asks its value model for the data to be displayed.

The basic way to set up this interaction between a widget and its value model is by:

- telling a widget the name of its value model (in our input field we filled in the aspect field on the basics page with counterValue, telling the widget to use a message with this name to access its value model in the application model.
- programming the application model such that it is able to create and return this value model. For example, since we have provided counterValue as name for of the message that will be used by the input field widget to access its valueModel, we will have to provide this message in the class SimpleCounterApp.

Defining stub methods, and opening the application As was said in the beginning, the application model is the glue for the models and the views/controllers. This means we have to implement:

- methods for every data widget to let the widget access its value model,
- methods that perform a certain action and that are triggered by action widget.

Luckily, VisualWorks helps us with this step by generating stub-methods, methods with a default implementation that can then be changed to provide the desired behavior. To create such methods, we have to fill in the properties for every widget on our canvas (which we have done in previous steps), and then we use the define property.

To define properties: deselect every widget on the canvas, and select the define button on the canvas tool. A list will come up with all the models where the system will create stub methods for. Leave all the models selected and press OK. The system will now generate the stub methods.

Note that often it is better to write by yourself the code generated, because you can have the control of the way the value model are created and accessed.

We now have a basic application that we can open. To do so, select the Open button on the canvas tool.. You now can click on the buttons, but since we have not yet provided any actions, the default action happens (which is to do nothing).

Go to your browser again, and deselect the class SimpleCounterApp, and select it again. Set the switch to instance, and you will notice that the generation process added some methods:

- two methods in the action protocol: increment and decrement,
- a method counterValue in a protocol aspects.

8.9 About value models

In previous section we explained that a data widget holds on to a value model, and that this value model actually holds the model. A data widget performs two basic operations with its value model:

- ask the contents of the value model using the value message,
- set contents of the value model using the value: message.

VisualWorks provides a whole hierarchy of different value models in the class ValueModel and its subclasses. The simplest is ValueHolder: it wraps any kind of object, and allows to access it using value (to get the stored object) and value: (to set the object). Sending the message asValue to that object creates a valueholder on an object. Moreover using a valueHolder ensure that its dependents receive the message update:, each time the value model receives value:.

In our application, we have an input field that should display a number. The input field is a data widget, so it has to hold on to a value model. This value model will actually store a number. Note that the Model-View-Controller principle tells us that the data widget (a view-controller pair) should not know its model directly. Therefore, the input field only knows that it has to send counterValue to the application model, and the model knows nothing (since it is wrapped in a value model). This means that we have to program our application model so that it provides the correct mapping.

If you look at the implementation of the method counterValue (a stub method generated by the define command), you will see the following piece of code:

```
counterValue
        "This method was generated by UIDefiner.  Any edits made here
        may be lost whenever methods are automatically defined.  The
        initialization provided below may have been preempted by an
        initialize method."

        ^counterValue isNil
                ifTrue:[counterValue :- 0 asValue]
                ifFalse:[counterValue]
```

This code implement a lazzy initialization of the value model. This means that if the valueModel (counterValue) is defined, it is created, stored and return. If the valueModel is already defined, it is just simply return. Note that this is the method that is sent by the input field to access its value model.

Note such kind of lazzy initialization can be replaced by the following methods:

```
SimpleCounter>>initialize
        super initialize.
        counterValue := 0 asValue.
```

```
SimplieCounter>>counterValue
        ^ counterValue
```

The following code only works that the initialize method is automatically invoke when the application model is created. This is the case because the class ApplicationModel class defines a class method new as follows.

```
ApplicationModel>>new
        ^super new initialize
```

Exercise: Provide the implementation for increment and decrement, and test it.