OO Design with Smalltalk a Pure Object Oriented Language and Environment

About the exercises

The exercises developped in the following lessons have been originally written by Roel Wuyts and Koen De Hondt from University of Brussels.I heavily extended them and thanks them for sharing them with me. You can find all the information relative to the lecture at University of Bern at http://www.iam.unibe.ch/@@

About me

Dr. Stéphane Ducasse
Room 101
Email: ducasse@iam.unibe.ch
WWW: http://www.iam.unibe.ch/~ducasse/

Feel free to come and ask questions about the lectures and possible student projects or DiplomatArbeit. You can send email, I will reply but it is always better to have an answer in front of a browser. The last versions of theses files will be available at http://www.iam.unibe.ch/~ducasse/

Learning Objectives

At the end of this lecture you should be able to

- Read Smalltalk code and understand it
- Interact with the environment and find information
- Understand some basic library element
- Define your own classes
- Apply some elementary OO design guidelines

How do you will get evaluated?

The most important question is how will you get evaluated for this lecture. As this lecture is not obligatory and is a special lecture, there will be no final exam. However, you will have to prove that you made your exercises and that you know how to interact with the system. This means that at the end of the lecture I will personaly take some time with each of the attendee and ask him to find some information with VisualWorks or explain some pieces of code. The questions that I will ask will be exactly at the same level that the first exercises. So for those that will effectively open VisualWorks and do the exercises, there will be no problems. For the other ones, I think that they will not obtain the lecture.

How do we will proceed?

You will have to do the following exercises at home or during your free time and during the exercises sessions. Do the exercises of the first 4 chapters alone and ask me questions during the first exercises session. Pay attention that the exercises at the end are more exciting that the first stupid ones. I will start the exercises considering that the first 4 chapters have been worked. So you can ask me questions on the previous chapters but you should at least have tried before.

Outline of the exercises

1. Basics of the VisualWorks Smalltalk Environment
2. Objects and expressions
3. Viewing, creating and editing classes
4. Defining protocols and methods
5. Basic LAN Application
6. Understanding Self and Super Better
7. Extending the LAN Application
8. VisualWorks Application Building
9. More about Applications
10. Building an Interface for the LAN Application
11. Building a Dialog and originating packets

Where to get VisualWorks or Squeak?

You can get a full but non-commercial version of VisualWorks 3.0 at http://www.object-share.com/vwnc/. Several versions are available: PC, Linux and Mac. The only difference between a non-commercial version and a commercial one is that you should not develop software that you sell with the non-commercial version. Moreover, you cannot load commercial parcels (byte-code) into a non-commercial virtual machine. The inverse is possible i.e. you can distribute code developed with the non-commercial to person using a commercial version.

For Squeak go to http://www.squeak.org/
Register, download and install the version. If you have any kind of problem for installing VisualWorks, just ask and do not wait not the end of the lecture for that.

Some Conventions

Throughout the exercises I will use italic text (like this) for text you have to type and courier (`like that`) to refer all the Smalltalk entities (class names, instance variables, methods).

Chapter 1

Interacting with the VisualWorks Smalltalk Environment

1.1 Meaning of the files

In Smalltalk, the source code of classes and methods is translated to Smalltalk-byte code, 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 executed directly on the processor.)

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

- visual.sou (ASCII): contains the textual code of the initial classes of the system. This file is never changed. It is only read, never written.
- visual.im (Binary): contains byte code of all the objects of the system, the libraries and the modifications you made. This file is your personal file representing the state of your system after every action you make.
- visual.cha (ASCII): contains all the modifications made in the image-file and all the new source code since this was created (cha is for changes). This file should be in sync with the .im file. This file is useful to restore the state after an image crash.

There are several implications to consider: You need to have write access to the .im and .cha files. The .sou file is shared by all the users to save space. The VM being a byte-code stack machine does not need to have the source code to work, only the byte-code (the image) suffices. However you as a programmer need to read the code. In case of problem like “source code not found”, the system automatically disassembles the byte-code and display it instead of the source code. That’s why if the browser shows you code like the one displayed below containing t1 t2 … instead of normal variable names, do not save your image and check if the source code is reachable by the system (look into the settings if the variable VISUALWORKS points to the right directory).

```
accept: t1
      "     ***This is decompiled code.***
      This may reflect a problem with the configuration of your image
      and its sources and changes files.
```

```
        Please refer to the documentation and the settings tool for help
in
        setting up the proper source code files."
        self send: t1.
        ^self
```

1.2 Starting up

On Macintosh, to open an image:

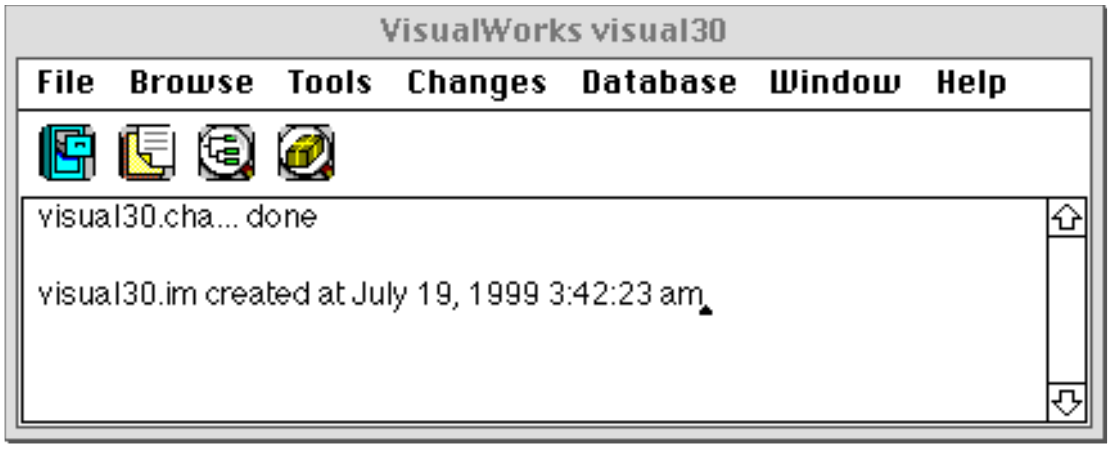
- Drag the file 'visual.im' on the virtual machine to start the image if this is the first time, or double click on it.
- If you want to start your own image, just double click on it or drag it over the virtual machine.

On Solaris: you should invoke the virtual machine passing it an image as parameter. Let us imagine that VisualWorks has been installed in the directory visualworks and that the virtual machine is called vw you will have to type

visualworks/bin/vw visualworks/image/visual.im

Then after you can specify your own image visualworks/bin/vw myvisual.im

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 the copyright message). You can iconize or close this last one, since we do not need it for the moment. Associated with the Launcher is the Transcript. The Transcript is the widget that displays the information of the system below the Launcher toolbar. In the displayed Launcher the Transcript displays the feedback we get when we save an image. We will see later on how you can put your own messages there.



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.

1.3 Creating your own image

The first time you started with the default image. This image is used by all the users (and you if you want to restart with a clean image in the future), so you should not modify it. Note that you may not have the right to do write access es with this image. We need to create your own image. We do so by saving the default 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, give the name you want for your image.
- Have a look at the Transcript and note what it says.

1.4 About the mouse

The Smalltalk Development Environment of which VisualWorks is a descendant was the first application to use multiple overlapping windows and a mouse, back in 72 when it was created. 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: you can select text, select the window or part of the window where you want to type text.
- The middle button is the operate button: once you have selected a piece of text or a window, this button allows you to interact, to operate on it. Depending of the context, you can for example, copy, paste or undo some text, or compile, format methods.
- The right button is the window button: this button has the responsibility to interact with the windowing system. You can iconify, resize, relable, close windows.

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 mouse button pressed alone.
- For the operate button, press the button while holding the alt-key pressed.
- For the window button, press the button while holding the apple-key pressed.

1.4.1 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 : 1

Select some parts of text in the `Transcript`.
Hints: There is a faster way to select a piece of text. You can also select a single word by double clicking on it. When the text is delimited by ' (single quotes), "" (double quotes), () (parentheses), [] (brackets), or {} (braces), you can select anything in between by double clicking just after the first delimiter.

Exercise : 1

Try these new selection techniques.

Now we have a look at the text operations offered by the mouse menu. 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 : 1

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

Exercise : 1

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 : 1

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

1

Exercise : 1

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?

1.5 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
- Or you can click on the workspace icon: 

You will see a framing rectangle (with your mouse in the upper left corner), that indicates the position where the Workspace will open. You can move your mouse around to change the 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 of the Workspace. 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 window. Experiment with it until you feel comfortable with it.

1.5.1 The Window menu

To resize a window on the Macintosh, click in the lower right corner while holding the alt-button (option). 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.

1.6 Evaluating Expressions

In the operate menu you will see the next three different options do it, print it and inspect for evaluating an expression and getting the result:

- do it: evaluates the current selection, and does not show any result of the evaluation result. For example, if you evaluate the following expression `Browser open` using

do it the system will open a class browser but it will not show you a result. We do it when we are interested only by the side effects of an expression and not its result.

- print it: prints the result of the evaluation of your selection just after the selected text. The result is automatically highlighted, so you can easily delete it if you want to. For example if you select and evaluate the expression `Browser open`, the system will open a class browser and will print the result of the expression in our example you will obtain the string `an UIBuilder`. It is important to notice that the displayed result is a string. The system obtains such a string by sending the message `printString` to the result. The same effect can be obtained by sending the message printString explicitly to the result of the expression evaluation. "Printing" the expression `Browser open` is equivalent to do it the expression `Browser open printString`.
- inspect it: opens an inspector on the result of the evaluation. Inspecting works in a similar way that printing an expression. The main difference is that the result is not printed in the current window but open an inspector. The system obtains an inspector by sending the message inspect to the result. The same effect can be obtained by sending the message inspect explicitly. Inspecting the expression `Browser open` is equivalent to "do it" the expression `Browser open inspect`.

The distinction between these three operations is essential, so check that you REALLY understand their differences.

Exercise: **1**

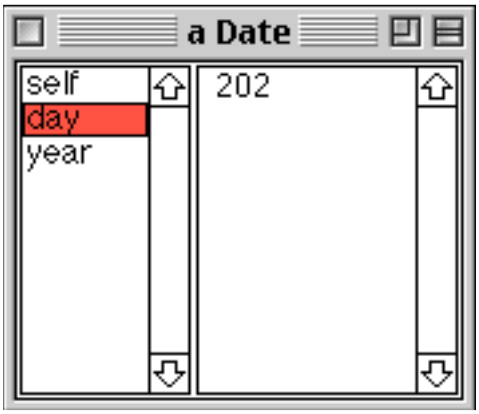
select 3, bring up the operate menu, and select print it.

Exercise: **1**

print the result of 3+4

Exercise 9: **1**

Type `Date today` and print it. Afterwards, select it again and inspect it. You should have a window similar as the one below. Such a window is an inspector on the result of the evaluation of the expression `Date today` (the expression tells to 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, in this case `day` and `year`. On the right is an edit field showing the value of the currently selected variable.



- Click on `self` in the inspector. What do you get? Does it resemble the result shown by `Date today printString`?
- Select `day`. What do you get? Now change this value, bring up the operate menu, and select and accept it (menu operation). Click again on `self`. Any difference?
- In the inspector edit field (right 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 that we are currently inspecting) and the result to be printed. Experiment with other expressions like:
  ```
  self daysInMonth
  self monthName
  ```
  Close the inspector when you are finished.

Note. It is really important to be clear about the fact that an inspector is a debugging tool. In this sense it directly accesses and shows the internals of an object, thus it violates the encapsulation principle. An inspector does not use the public interface of an object but uses the reflective capabilities of Smalltalk to access object internals. That's why you should be aware that some of the information that an inspector shows is strictly dependent on the private internal representation of the inspected object.

- What hypothesis can you elaborate on the way dates are internally represented?
- Check the interface of the `Date` class in particular the instance creation protocol on the class side. Do you agree with us that there is a difference between the way objects are represented and their public interface defined in terms of behavior?

Exercise: 1

Type in the Workspace the following expression: `Time now`, and inspect it. Have a look at `self` and the instance variables. Browse the interface of the class `Time`.

Exercise: 1

Type in the Workspace the following expression: `Time dateAndTimeNow`. This tells VisualWorks 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.

1.7 Using the System Transcript

We have already seen that the `Transcript` is a text window at the bottom of the Launcher where the system shows you important information. You can also use the Transcript yourself as an inexpensive user interface.

If you have a Workspace open, place it so that it does not cover the `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 "do it" them.

This will cause the string This is a test to be printed in the `Transcript`, preceded 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). This 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 `printStrin`g 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 printStrin`g and `'is the answer to the Universe'`.

Exercise: **1**

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 ';'?

Chapter 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: 1

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 result returned by evaluating the expression, what is the string representation of the result?

Date today

What is the receiver object?
What is the message selector?
What is/are the argument(s)?
What is the result returned by evaluating the expression, what is the string representation of the result?

#(calvin hates suzie) at: 2 put: ‘loves’

What is the receiver object?
What is the message selector?
What is/are the argument(s)?
What is the result returned by evaluating the expression, what is the string representation of the result?

Exercise: 1

What kind of object does the literal expression **`'Hello, Dave'`** describe?

Exercise: 1

What kind of object does the literal expression **`#Node1`** describe?

Exercise: 1

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

Exercise: 1

What can one assume about a variable named **`Transcript`**?

Exercise: **1**

What can one assume about a variable named **`rectangle`**?

Exercise: **1**

Examine the following expression :
| anArray |
anArray := #('first' 'second' 'third' 'fourth').
^anArray at: 2

What is the resulting value when it is evaluated (^ means return)?

Exercise: **1**

Remember that the precedence rules are the following, the greater precedence is evaluated prior to the lower.

Unary > binary > keywords
() > Unary

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: **1**

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: **1**

Examine the following expression:
25@50
What type of message is being sent?
What is the message selector?
What is the receiver object?
What is the resulting value (use VisualWorks for this)?

Exercise: **1**

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: **1**

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: **1**

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: **1**

In lesson 1 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: **1**

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? Test your version. Rewrite this code using `inject:into:`

Exercise: **1**

Evaluate the following expressions and elaborate a hypothesis regarding the difference between identity and equality between strings and symbols.

```
'lulu' = 'lulu'
'lulu' == 'lulu'
#lulu = #lulu
#lulu == #lulu
```

Exercise: 1

Understanding the common protocol shared by all the collections is a key point in learning of Smalltalk and will help to write fast, simple and elegant code. You should be able to identify and use the following methods: do:, collect:, detect:, reject:, select:, includes:, isEmpty, size and inject:into:. Guess the results of the following expressions and check them by evaluating them.

Knowing that `ColorValue constantNames returns the following value` #(#black #blue #brown #chartreuse #cyan #darkCyan #darkGray #darkGreen #darkMagenta #darkRed #olive #gray #green #lightYellow #lightGray #magenta #navy #orange #orchid #paleGreen #pink #purple #red #royalBlue #salmon #lightCyan #springGreen #veryDarkGray #veryLightGray #white #yellow)

Guess the result of the following expressions:

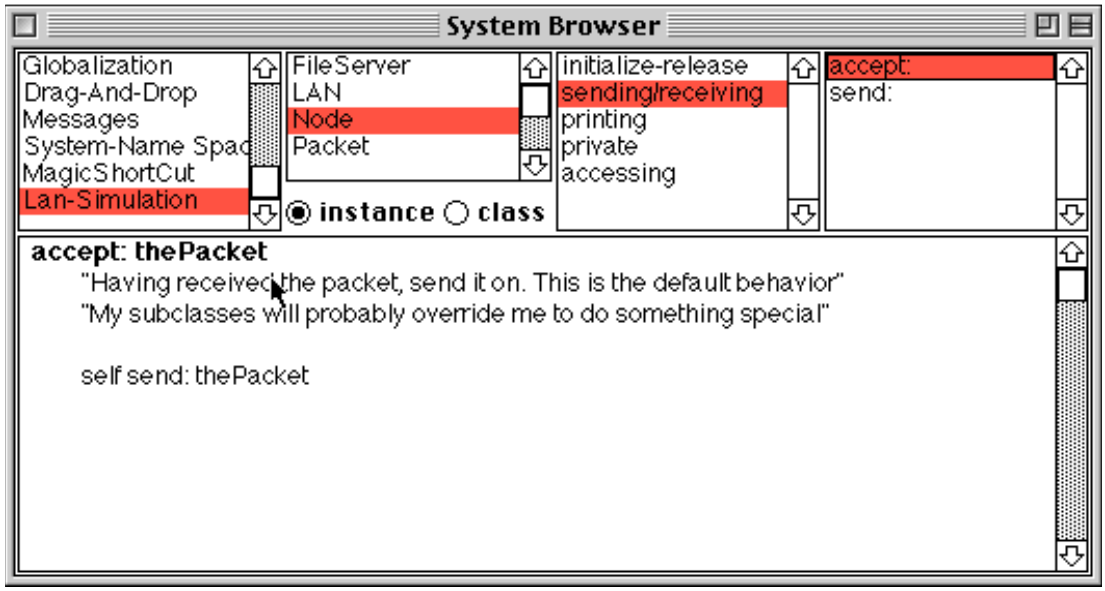
```
ColorValue constantNames size

ColorValue constantNames includes: #turqoise

ColorValue constantNames select: [:each| each size = 4]

ColorValue constantNames detect: [:each| each size = 4]

ColorValue constantNames reject: [:each| each size = 4]

ColorValue constantNames collect: [:each| each asString]

ColorValue constantNames do: [:each | Transcript show: each ;cr]

ColorValue constantNames
inject: ''
into: [:final :current| final , current asString]
```

Chapter 3

Viewing, creating and editing classes

This lesson will show you the use of the System Browser to browse through the class system, to define a class, and to save this class to file. In the first part, we will browse through some classes. In the second part, we will create our first class. In the third part, we will save it. For this lesson you need to work on your own image, so first start a new image (use visual.im), and save it under a different name (for example, lesson).

From the Browse menu in the Launcher, open a Class Browser. This is the basic tool you use to find classes, browse their code, and implement your own classes and methods (see the screen dump below). A class browser is composed by four list views and one edit field below.



3.1 Looking at existing classes

A Smalltalk environment like VisualWorks contains more than 1500 classes and 8000 methods. To ease navigation in this huge amount of information, the system proposes some means to organise and navigate through it. The idea is to categorize the classes and the methods into groups or folders named class categories for the classes and method pro-

tocols for the methods. Please note that these elements do not possess any language semantics and are just a way to organize the information.

The Class Browser consists of four lists on top and of an edit space. The four lists are (from left to right):

- The category list displays groups of classes (= categories). In the picture, the selected class category is 'Lan-Simulation'.
- The class list shows the classes in a selected category. In the picture, FileServer, Lan, Node and Packet are the classes classified into the category 'Lan-Simulation'.
- The protocol list shows groups of methods (= protocols) in a class. In the displayed browser the protocol named sending/receiving groups all the methods related to the packet acceptation and sending.
- The method list shows the methods of the selected protocol.

The contents of the edit field that spans the whole lower part of the window changes depending on the current selection you make in the lists. Initially, with nothing selected, it is empty. In the previous screenshot it displays the current selected method.

How to Browse?

One of the predominant impressions when one starts to program with Smalltalk is that there is too much information available. The truth behind this is that all the information is potentially available. However:

- You do not have to know all the classes and all the methods before starting to program.
- The environment is there to help you to find the information you are looking for. Use the method senders and implementors (menu operate on the method list pane, or Implementors of the Browse Menu) to quickly identify which classes implement the selected method and which methods callthe selected method.

A good way to read this information is to consider the class browser as a book: The four panes representing the sections, subsections and subsubsections, the method senders and implementors function as the cross-references and index, the explain function (menu operate on the edit field) as a first aid. Moreover you do not have to read the body of the method to use it, normally reading the first line containing the method name and argument and the comments explaining what the purpose of the method is should suffice. You should only read a method body as your last chance to understand the purpose of the method or to understand how the programmer implemented the functionality.

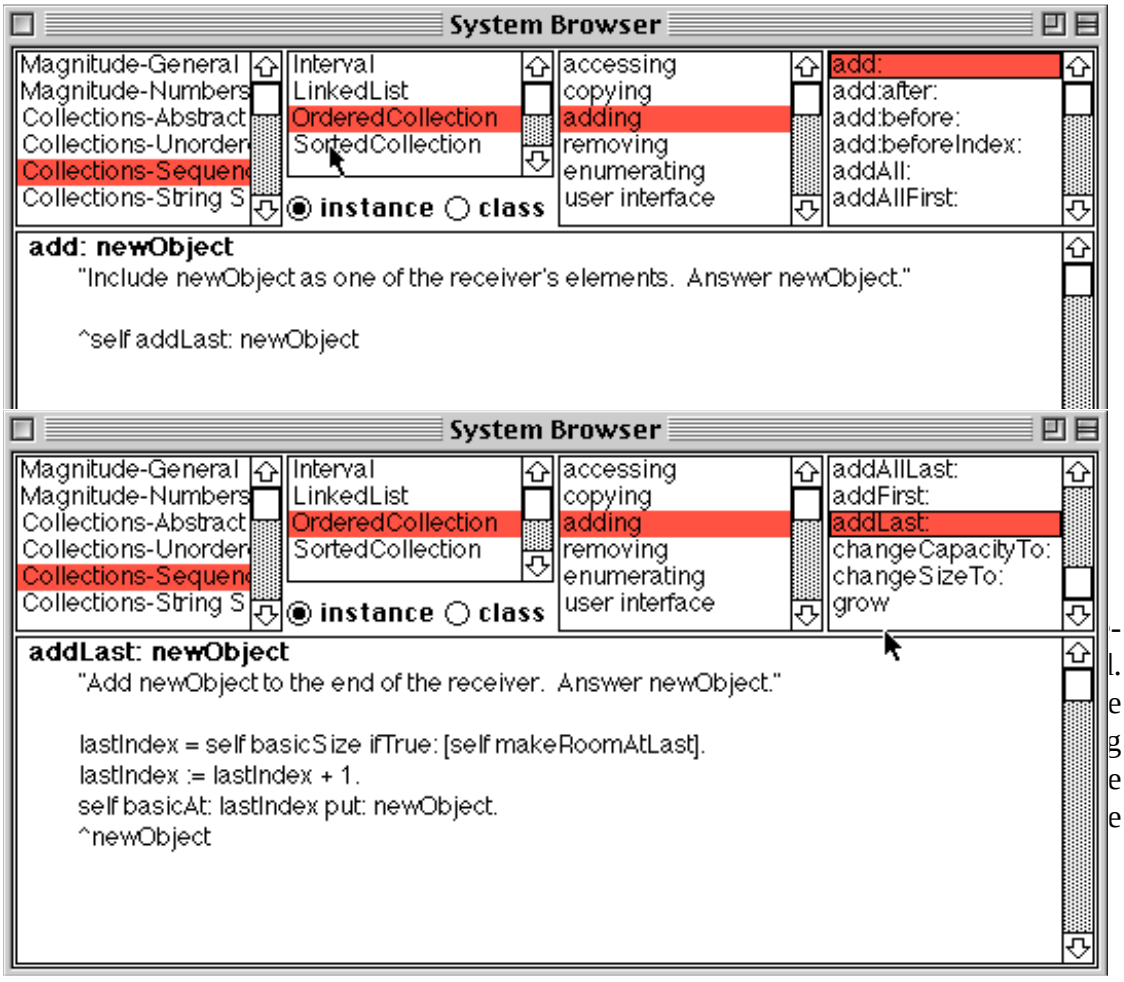
Illustration of Browsing

Let us consider the following example that first creates an empty ordered collection, then adds 35 to it.

If you inspect the result you will get 35 and not the collection. This is one of the famous Smalltalk library legacy. One way to solve this behaviour is to add a new line containing `ordColl`. However imagine that we want to understand why we have such a behaviour.

```
|ordColl|
ordColl := OrderedCollection new: 5.
ordColl add: 35.
```

After browsing the class `OrderedCollection` you should be able to obtain the following situation.



This is just a simple illustration of the power of the ability to read the code. Since ALL the environment and ALL the entities of the system are just objects, their code is available on line. This means that you can for example read the code of all the objects you want to understand or extend.

An important clarification. A lot of programmers are afraid by the fact that all the code is available. Especially for their product they think that this is dangerous to openly deliver the code they are producing. They are right. The main point here is that the source code is only present in the development environment and not in the running executable. When you sell a product developed in Smalltalk, first lot of the code implementing the development environment is removed (debugger, compiler, parser, editor…) and second you normally only delivers an executable consisting of a VM and the byte code of your application, not its source code.

3.1.1 Exercises

We will begin with selecting a category called `Graphics-Geometry`. When you do that the content of the edit field will be updated to present you a class-template. This template is used when you want to create classes (we will do so later on). The class list shows the classes in the selected category.

Exercise: **1**

Now select the class `Point`. The edit field shows the definition of the class `Point`. Note how the template is filled in. Try to understand the structure of this class.

- Ask for all the references to this class
- Ask for all references to the instance variable `x`
- Ask for all the senders of the method `x`
- Ask for the comment of the class `Point`

You can now select the protocol called `accessing`. The edit field updates again, to enable you to add a method to the class `Point` in the selected protocol. Have a look at the different protocols and their methods. Select methods in the method list, and look at their code. Begin in the `accessing` protocol, and try to understand what is going on.

Exercise: **1**

Every list in the upper half of the class browser has its own operation menu that is displayed if you press the operation button on a list. Do this for the four lists, and try out any commands that you do not fully understand (do not remove anything).

Exercise: **1**

What are the superclasses of `Array`?

Exercise: **1**

Who references (by name) the class `ByteArray`?

Exercise: **1**

How many instance variables does an instance of class `LuminanceBasedColorPolicy` have?

Exercise: **1**

Using the method sender and implementor functionality describe step by step how you would rename a method such that other methods that once used it can still do? A freely available tool, the Refactoring Browser that you can load (menu Tools item Load Parcel Named), does all these steps automatically for you. It contains a lot more refactorings and is a really good browser. Try it.

Exercise: **1**

Find at least 3 classes that implement the method `at:`

Exercise: **1**

Can I compare instances of the class `Date` with the `>` and `<` operators? Give reasons for your answer.

Exercise: **1**

Go to the class `FixedPoint`, and locate the protocol double dispatching. This protocol contains the methods for a technique called double dispatching. Try to figure out what this technique is all about and what it solves. To do so, look at where the methods of this protocol are used.

Exercise: **1**

Find all the classes implementing `ifTrue:ifFalse:`

Exercise: **1**

Evaluate and explain the differences between the two following expressions:

```
0 to: 10 by: 2 do:
    [:i | Transcript show: I printString ; cr]
(0 to: 10 by: 2) do:
    [:i | Transcript show: I printString ; cr]
```

Hint: Find the method `to:by:do:` and the method `to:by:` defined in `Integer.`

Exercise: **1**

Print the following expressions and explain why the result is different.

```
Array with: 1 with: 2 with: 3
Array with: 1 ; with: 2 ; with: 3
```

Exercise: **1**

Find the method `factorial` and reimplement it using `inject:into:`.

Exercise: **1**

Check using a hierarchy browser which subclasses of `Magnitude` are abstract classes. Check especially the comments. You can also check all the sender of `subclassResponsibility`

Exercise: **1**

Search the implementors of `->`
Inspect the expressions

```
#lulu->23
23->34
```

Exercise: **1**

- Write an expression that returns the subclasses of `Collection` class.
- Using only public methods of the class `Behavior,` find the expression that returns the number of methods defined in one class.

Exercise: **1**

(for the wild and foolish) Find the method `browseAllSelect`:
Using this method write an expression that opens a browser showing all the unary methods.

Hint: look in the `CompiledMethod` class or its superclasses how we can know the number of arguments of a method.
Browser browseAllSelect: [:method | method .....]

Exercise: **1**

Inspect the following expression: #(calvin hates suzie) at: 1 + 1 put: #loves
The result is not what we expected, how do you explain that? Find the method responsible for such an unexpected behaviour.
Propose (yourself) a solution, so that the array (the receiver) is returned instead of the result of the message.

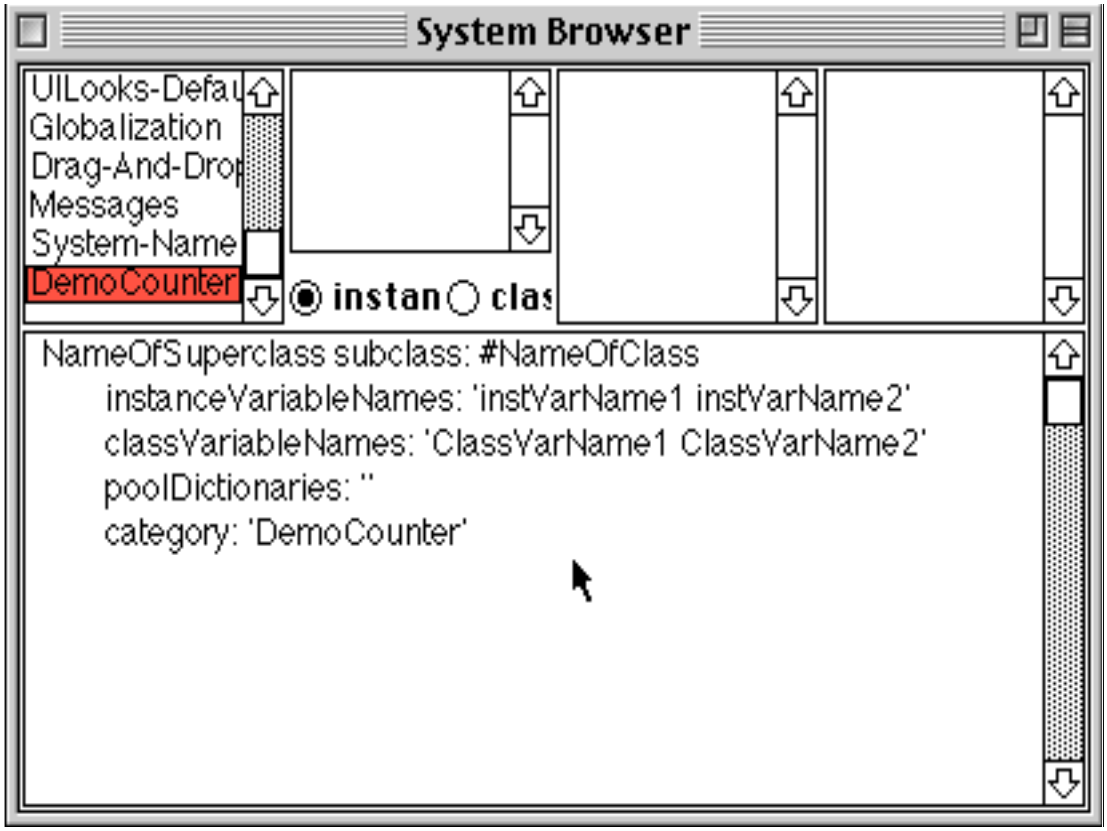
3.2 Creating your own class

In this part we will create our first class in a category of our own. The steps we are going to take are the same every time you create a class, so memorize them well. We are going to create a class `SimpleCounter` in a category called `DemoCounter`.

Step 1: Creating a category

Step 2: Creating a class

Creating a class requires the following five steps that consist basically of editing the class definition template to specify the class you want to create. Before you begin, make sure that only the category `DemoCounter` is selected.



- Superclass Specification. First, replace the word `NameOfSuperclass` with the word `Model`. This is to specify the superclass of the class you are creating. (Note that `Model` is the superclass used for objects that will play a model role in a MVC triad, see future lessons. So, for your other classes you should type the superclass of the class that you are creating).
- Class Name. Next, 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 and that you do not remove the # sign in front of `NameOfClass`.
- Instance Variable Specification. Then, fill in the names of the instance variables of this class. We need one instance variable called `counterValue`. You add it by replacing the words instVarName1 and instVarName2 with the word *`counterValue`*.. Take care that you leave the string quotes!
- Class Variable Specification. Now you can fill in any class variables you may use. Since we need none, remove the words `ClassVarName1` and `ClassVarName2`, leaving an empty string (i.e. 2 single quotes ’’).
- Compilation. That’s it! We now have a filled-in class definition for the class `SimpleCounter`. To add it to the system, we still have to compile it. Therefore, select the accept option from the operate menu. The class `SimpleCounter` is now compiled and added to the system.

As we are good citizens, we give `SimpleCounter` a class comment by selecting comment from the operate menu of the class list. Give this comment:

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

Instance Variables:
counterValue                                <Integer>
```

Select accept to store this class comment in the class.

Filing the category out on disk

To be able to load your class next week, we now create a so-called file-out. A file-out is a text file that contains method and/or class definitions, and that you can use to load your classes and/or methods in an image.

To create the file-out: Select the category `DemoCounter` and select file-out from the operate menu. Give a filename (for example 'democounter.st'). The system now writes the textual representation of all the classes in the selected category to this file. If you want to just save one class, one protocol or even a single method, you can use the appropriate operate menu item on the element you want to save.

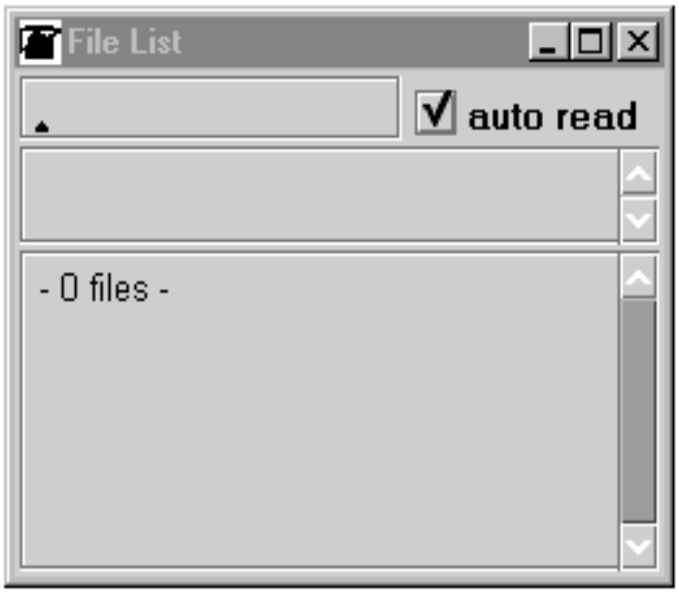
Chapter 4

Defining protocols and methods

This lesson will show how to use the System Browser to add protocols and methods. Therefore, we will use the class `SimpleCounter` created before, and add some behaviour. We will also test this class. If you saved your image at the end of the last lesson and you can restart it, just read the following to learn how to fill in code into your environment.

4.1 Filing in Smalltalk code

At the end of the previous lesson we created a file-out containing the classes in the category `DemoCounter`. To import these classes in the environment we are working, we need to perform a file-in operation. To do so, we will use a tool called the File List. To start with, open this tool (it's under Tools, in the VisualWorks Launcher). You can click on the file in icon. You get something like this:



There are three important parts in this window:

- On the top you have an edit field that allow you to specify in which directory you want to look for your files and which files you want to filter. For example if you want to see all the files with the extension .st in the directory called Macintosh HD:User:Stef: just type: Macintosh HD:User:Stef:*.st. Note that you can also open an operate menu in the edit field on top. The last menu item is volumes. Select this to get a list with the file volumes of your computer (partitions on a PC, hard disks on a Mac,…). Select the volume your files are on. When you have done that, the file list will contain the files and directories of that volume. To change to a specific directory, select the directory in the list, bring up the operate menu, and select new pattern.
- The middle part of the window display the list of the files that match the expression contained into the top edit field. You can select a file then using the operate menu file in it.
- The bottom subwindow is a text viewer, displaying either the contents or some statistics of the selected file.

Exercise: 1

If you do not have the image containing the definition of the class `DemoCounter`, file-in the file you have created in the previous lesson.

4.2 Creating and testing methods

The class we have defined has one instance variable, `counterValue`. Remember that in Smalltalk, everything is an object and that the only way to interact with an object is by sending it messages. Therefore, there is no mechanism to access the instance variables from outside. What you can do is define messages that return the value of the instance variable of a class. Such methods are called accessors, and it is common practice to always define and use them. We will start to create an accessor method for our instance variable `counterValue`.

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.

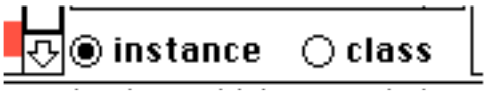
An important remark: Accessors can be defined in protocols `'accessing'` or `'private'`. Use the `'accessing'` protocol when a client object (like an interface) really needs to access your data. Use `'private'` to clearly state that no client should use the accessor. Again this is purely a convention. There is no way in Smalltalk to enforce access

rights like private in C++. 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 first define your accessors in a `'private'` protocol and once some clients really need access to some specific data, create a new protocol `'accessing'` and move your method 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.

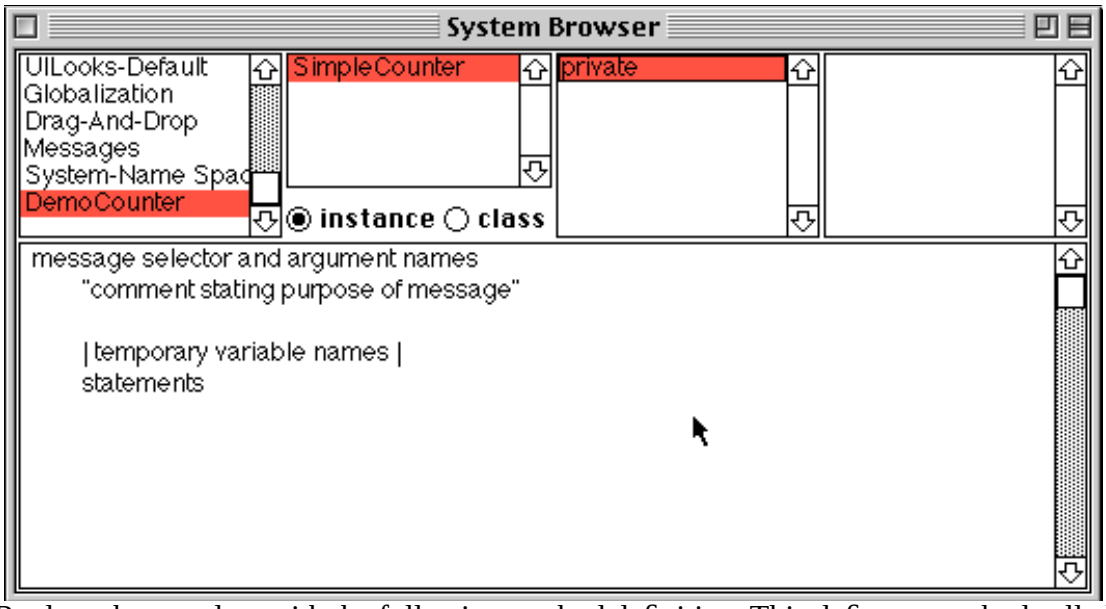
Exercise: **1**

Decide in which protocol you are going to put the accessor for `counterValue`

We now create the accessor method for the instance variable counterValue. Start by selecting the class *`DemoCounter`* in a browser, and make sure the class/instance switch is set to instance.



Create a new protocol. Select the newly created protocol. Then the edit field displays a method template laying out the default structure of a method.



Replace the template with the following method definition. This defines a method called counterValue, taking no arguments, having a method comment and returning the instance variable counterValue. Then choose accept in the operate menu to compile the method.:

```
counterValue
    ^counterValue
```

After having written the text, you can now test your new method by typing and evaluating the next expression in a workspace:

```
SimpleCounter new counterValue
```

This expression first creates a new instance of SimpleCounter, and then sends the message counterValue to it to retrieve the current value of counterValue. This should return nil (the default value for noninitialised instance variables; at the end of this lesson we will create instances where counterValue 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 create a new SimpleCounter instance and then sets the value of `counterValue` to 7:

```
SimpleCounter new counterValue: 7
```

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

Exercise 1

Implement the following methods in the given protocols:

protocol	methods
operations	increment self counterValue: self counterValue + 1
operations	decrement self counterValue: self counterValue – 1
printing	printOn: aStream super printOn: aStream. aStream nextPutAll: ' with value: ', self counterValue printString. aStream cr.

Now test the methods increment and decrement. Note that the method printOn: is used when you do print it or click on self in an inspector.

4.2.1 Adding an instance creation method

When we create a new instance of the class `SimpleCounter` using the message `new`, we would like to obtain an instance well initialized. 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 we have to define the `new` method on the class side. To define an instance creation method like the method `new` you should be on the class side, so set the class/instance switch on class.

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
```

4.2.2 Adding an instance initialization method

Now we have to write an initialization method that sets a default value to the `counterValue` 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 create a protocol `initialize-release`, and create following method (the body of this method is left blank. Fill it in!

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

Remark. As we already mentionned, the `initialize` method is not automatically invoked by the method `new`. We had to override the method `new` to call the `initialize` method. This 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

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

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

4.2.3 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 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, Be-

havior 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`.

Chapter 5

A Basic LAN Application

In the following lesson we will work on an application that simulates a simple LAN network. The purpose of this lesson is to create a basis for further lessons on writing OO programs. It will use the knowledge of previous lessons for creating classes and methods. We will create several classes for simulating the LAN: `Packet`, `Node`, `Workstation`, and `PrintServer`. We start with the simplest version of a LAN then during the following exercises we will add new requirements and modify the proposed implementation to take them into account.

5.1 Creating the Class Node

The class `Node` will be the root of all the entities that form a LAN. This class contains the basic behavior common for all nodes. The responsibility of a node is to be inserted into a network, which is basically a linked list of nodes, so a Node should know its next node. A node should be uniquely identifiable with a name. This is its responsibility to send and receive packets of information. We chose to represent the name of a node by a symbol because symbols are unique in Smalltalk and the next node by a node object.

Node inherits from Model
Collaborates with Node and Packet
Responsibility:
name (aSymbol) returns the name of the node
hasNextNode tells is a node has a next node
send: aPacket sends a packet to the following node
accept: aPacket receives a packet and treat it. Per default send it to the following node

Exercise **1**

Create a new category `LAN`, and create a subclass of `Model` called `Node`, with two instance variables: `name` and `nextNode`. (We ask you to create Node as a subclass of Model because in the future lessons you will create a user interface for a LAN and a node will play the role Model of a Model-View-Controller triad).

Exercise **1**

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.

Exercise **1**

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

Exercise **1**

Create an instance method `printOn:` that puts my class name and my 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 lesson, and consider that `name` instance variable is a symbol and `nextNode` a node).

Printing a `Node` should result in the following:

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

Node named: Node1 connected to: PC1

Exercise **1**

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 lesson). Leave `initialize` empty for the moment (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).

Exercise 1

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, add a protocol '`send-receive`', and implement the following two methods, for which we only give the selector and the comment (and some partial code):

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

Node>>send: aPacket
    "Precondition: self have a nextNode"
    "send a packet to my following node"
    Transcript show:
        self name printString,
        ' sends a packet to ',
        self nextNode name printString;cr.
        …
```

5.2 Creating the Class Packet

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

Packet inherits from Object
Collaborations: 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.

Exercise **1**

In the category `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 **1**

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

5.3 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 of file servers. Since it is kind of a network node, but provides additional behavior, we will make it a subclass of Node. That way, it inherits the instance variables and methods defined in Node. Moreover, a workstation have to treat packets that are destinated to it in a special way.

`Workstation inherits from Node`
`Collaborations: Node, Workstation and Packet`
`Responsibility: (the ones of node)`
`originate: aPacket sends a packet.`
`accept: aPacket does some actions on packets send to the workstation (printing in the transcript). For the other packets just send them to the following nodes.`

Exercise **1**

In the category **`LAN`**, create a subclass of **`Node`** called **`Workstation`** without instance variables.

Exercise **1**

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 acceptation of packet addressed to other node, 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 **1**

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.

```
Workstation>>originate: aPacket
    "This is how packets are 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."
    ...
```

5.4 Creating the class LANPrinter

Exercise **1**

Having only nodes and workstations 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 create a class `LanPrinter` here, a special node that receive packets addressed to it and print them (on the `Transcript`). Note that we named it this way because `Printer` already exists in the system. Write this class.

`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)

5.5 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 has some bugs that you should find and fix! As you will notice creating a LAN is boring. We will fix that in the future by proposing a NetworkManager class.

```
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 IG-
Printer.
        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 IGPrint-
er’.
        pc originate: packet.
```

As you will notice the system does not handle loops, we will propose a solution to this problem in future lessons. To break the loop, try Ctrl-C or Ctrl-Y depending of the VisualWorks versions.

5.6 Creating of the Class FileServer

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

Table 5.1

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

Chapter 6

Fundamentals on the Semantics of Self and Super

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

6.1 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
```

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
```

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:`.

6.2 `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.

Chapter 7

Object Responsibility and Better Encapsulation

7.1 Reducing the coupling between classes

To be a good object you have to 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.

7.1.1 Law of Demeter

@@ Stef @@

7.1.2 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 is badly coupled and being really difficult to change to meet new requirements.

7.1.3 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.

Exercise 1

- 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.

Chapter 8

The Question of Class Responsibility

8.1 Class 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 be 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.

8.1.1 Applying to 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 (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.

8.1.2 Say Something Only Once

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 apply the rule ‘Say only once’ and avoid to duplicate the information. We 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.

8.2 Defining a Creation Interface.

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 two 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
    ....
```

Class methods for sharing default values.

For the method to:, the contents of the packet is not defined.There are two ways to provide a default value: (a) you let the initialize method defining the default value as shown in the previous section or (b) you can invoke send:to: with the default value (Note that the initialize method should not be called). Moreover, you should consider that the method defaultContents as you implemented it in the previous section is an instance method and that while you are implementing the method to: you do not have already created an instance.

The solution to this problem is to define two methods defaultContents one at the instance level and the other at the class level as follow:

```
Packet class>>defaultContents

    ^ 'no contents'

Packet>>defaultContents

    ^self class defaultContents

Packet class>>to: aSymbol
        ^self send: self defaultContents to: aSymbol
```

Exercise **1**

Implement this solution

8.2.1 Applying to 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.

Exercise **1**

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.

8.3 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.

8.3.1 Forcing client to use the right method

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.

8.3.2 Avoid to call super on a different selector

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.

basicNew to the rescue!

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 that always returns a newly created instance.

Exercise **1**

Browse all the methods starting with 'basic*' and limit yourself to `Object` and `Behavior`.

Exercise **1**

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

8.4 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 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.

Chapter 9

Hook and Template Methods

Hook and Template methods are soem of the basic tool that a designer has to design extensible system. Template method set up the context in which hook methods will be called. In this context hook methods represent customizable entry points that future subclasses can specialize while being sure to be invoked in a coherent context. Here we present a small scenario to show you how template and hook methods talk to each other.

9.1 Studying a famoos couple

The Smalltalk class library contains lot of hooks that allows an easy customization of the proposed behavior. For example every object knows how to respond to the message printString by returning a string. Such a behavior is implemented in the following way: the method printString is a template method that creates a stream which is passed as argument of the printOn: hook method. printString is the method called when you do a print it.

```
Object>>printString
    "Answer a String whose characters are a description of the receiver."

    | aStream |
    aStream := WriteStream on: (String new: 16).
    self printOn: aStream.
    ^aStream contents
```

Per default the printOn: method defined on the class Object writes in the stream argument the concatenation of ‘an’ or ‘a’ and the class name of the receiver.

```
Object>>printOn: aStream
    "Append to the argument aStream a sequence of characters
    that describes the receiver."

    | title |
    title := self class name.
    aStream nextPutAll:
          ((title at: 1) isVowel ifTrue: ['an '] ifFalse: ['a ']).
    aStream print: self class
```

As Object is the root of the inheritance hierarchy, any class can simply sepcify a new printOn: without having worry about the template method.

Example of hook methods

For example the class Array specializes the printOn: method is the following way:

```
Array>>printOn: aStream
    "Append to the argument, aStream, the elements of the Array
    enclosed by parentheses."

    | tooMany |
    tooMany := aStream position + self maxPrint.
    aStream nextPutAll: '#('.
    self do: [:element |
            aStream position > tooMany
                ifTrue:
                    [aStream nextPutAll: '...(more)...)'.
                    ^self].
            element printOn: aStream]
        separatedBy: [aStream space].
    aStream nextPut: $)
```

The class False has only one instance false so the specialization is rather simple.

```
False>>printOn: aStream
    "Print false."

    aStream nextPutAll: 'false'
```

The class Behavior that represents a class extends the default hook but still invokes the default one.

```
Behavior>>printOn: aStream
    "Append to the argument aStream a statement of which
    superclass the receiver descends from."

    aStream nextPutAll: 'a descendent of '.
    superclass printOn: aStream
```

Exercise 1

Ask all the implementers of the hook method printOn: and browse some of them. Verify that there is only one printString method defined in the system.

9.2 Designing our own hook/Template couple

Current Situation

A possible way to print a `Node` is the following one (the first line is the call and the second line the resulting string).

```
(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 an 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
```

However the do not want to duplicate the same code in all the subclasses of Node.

Exercise 1

- 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)) simplePrintString
    'Workstation Mac'

(LanPrinter withName: #PC1) simplePrintString
    'Printer PC1'
```

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

```
 (Node withName: #Mac connectedTo: (LanPrinter new name: #PC1))
'Node  Mac connected to Printer PC1'
```

The method `typeName` is a hook method. 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 a template method that 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.

Chapter 10

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.

10.1 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.

Solution1.

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:

- 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.
- 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.
`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.
`hasBeenAcceptedBy: aNode` tells a packet that it has been accepted by a given node.
`isAcceptableBy: aNode` answers if a packet is acceptable by a 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.

10.2 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.

10.3 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.

10.4 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.

10.5 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.

10.6 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

10.7 Introducing a Lan Manager

@@

Chapter 11

VisualWorks Application Building

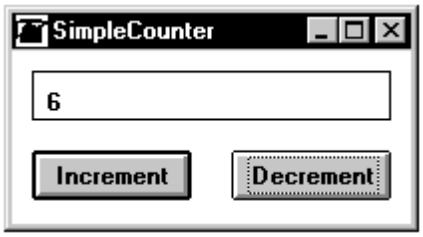
So far, you have been introduced to the basic OOP concepts and basic VisualWorks tools. This lesson will give an introduction to building applications in VisualWorks. Therefore, we will create a simple user-interface for our SimpleCounter application.

11.1 Model-View-Controller

As explained in the lecture, VisualWorks extensively uses the MVC paradigm, mixed with a dependency mechanism. In this lesson we will present the basic use of MVC, while the following lessons will further explore this, and the dependency mechanism.

Specs for the Application

Now we create a very simple application that introduces the basic tools and procedures to follow when developing a VisualWorks application. This is the interface for the application we are going to develop:



When the application opens, the input field will display 0. When the
the increment button, this value should be incremented by 1. Clickin
rement button will decrement the value with one.

11.2 ApplicationModel

As told before, everything in Smalltalk is an object. So are applicat
ate: they are instances that know how to open, view, update and chang

selves, accept input, ... So, creating an application consists of cr
that knows how to do all these things. Luckily, there is already a
defined that you can use to build applications. The basic class you ar
build will therefore be a subclass of a class called ApplicationMod
tionModel 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 sp
thereby linking the views/controllers used in the interface with
model. As explained in the lecture, models and view/controllers do
each other directly, but will each talk to the applicationModel that
everything together.

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

- building the interface
- programming the applicationModel

11.3 Building the interface

Now we will need to build the interface as pictured above. An interfa
several widgets (user interface elements), in this case an input fie
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 conte
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.

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 like 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:

- 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.
- 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.
- 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).
- 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:

- to select a widget: click it once
- to deselect a widget: hold down the shift button while clicking on the selected widget, or click somewhere outside the widget
- 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.
- to move a widget: select it, press the select button between the handles of the widget and move it
- to cut/copy a widget: select the widget, bring up the operate menu and select cut/copy in the edit menu
- 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 1

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 widget's 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: 1

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.

Page	Property	Setting
Basics	Label	decrement
	Action	decrement
	Be Default	unchecked

Exercise: 1

Set and apply these properties for the right button:

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 application's 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 window's 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'.

11.4 Programming the application model

As said in previous section, we now have to program our application model to:

- specify the interface's appearance and basic behavior,
- supplement the application's 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.

11.4.1 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:

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

SimpleCounter>>counterValue
    ^ counterValue
```

The following code only works if 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 2: 1

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

Chapter 12

Lesson 10: More about Applications

This lesson uses lesson 8 as basis, and explains some extras about application building.

12.1 Outlining

On the canvas tool you see a line of buttons as below, that is used to line out components. The first 6 are used to align them with other widgets, the middle 4 are used to equal spacing between widgets, and the last two are concerned with equalling heights and widths.

Exercise **1**

Use these alignment tools to properly align your application

12.2 Making the widget's positions relative

A handy feature is to set up the size and position of the widgets relative to window bounds. You make widgets relative using the Position page in their properties. The proportion sets the percentage (between 0 and 1) for the relative position; the offset uses this as start. Note that 0 means left or top and 1 means right or bottom. For example, to say to our input field that it should at all times keep 10 pixels from the left and right border, we would set the first (L) and the third (R) positions to:

L 0 10
R 1 -10

To make sure that our left button keeps ten from the left side, and keeps ten from the middle of the window, we use:

L 0 10
R 0.5 -10

Likewise, for the other button:

L 0.5 10
R 1 -10

Exercise **1**

Make the components relative, and resize your application…

12.3 Changing the input field's model

Currently, the model of our input field is a simple number. This means that we have to put more logic in our application, including behavior that one would expect in the model. In other words, there is too much logic in the application, which gives problems when updating/reusing this application and model. This section will therefore use our implementation of SimpleCounter (see lessons 3 and 4) as model instead of number.

There are two major issues we have to deal with:

- 1.use a SimpleCounter instance as model instead of a number,
- 2.take care of the dependency-mechanism

12.4 Make a SimpleCounter instance the model

As explained before, the inputfield has a valueholder as model, and uses the messages value and value: to get/put the data from it. However, SimpleCounter has no messages value and value: but messages counterValue and counterValue:. This means we cannot use a simple value holder that holds the SimpleCounter instance, but have to use a more sophisticated one that translates these messages. This is done using the class AspectAdaptor. So, our input field will hold an AspectAdaptor, which will actually hold the instance of SimpleCounter. To use this aspectAdaptor, we have to initialize counterValue like this:

```
counterValue := (AspectAdaptor forAspect: #counterValue)
                    subject: SimpleCounter new;
                    subjectSendsUpdates: true
```

In the application model, the method counterValue is used to return the actual model to be used by the input field. Since this method uses lazy initialization, it actually performs two functions (see lesson 8):

- 1.initialize and return the value model (the ifTrue:-branch),
- 2.return the value model if it has been initialized

Exercise 3: Adapt the implementation to use the implementation given above. Take care because, when a user uses define… in the Canvas Tool, the counterValue method is regenerated !

12.5 Dependency mechanism

As explained in the lectures, in the Model-View-Controller the model does not know its dependents and does not invoked directly their update when it changes. Instead, it sends to itself a changed message, this has as a consequence that its dependents know their model has changed and that their update method is invoked. Our model, the SimpleCounter instance does not send change messages… yet. Since the aspect we are interested in is counterValue, we have to send a change message in the counterValue mutator (the accessor is just used to get the value, so there's no need to send a change message there). Change the counterValue mutator code so it resembles the following:

```
counterValue: aNumber
        counterValue := aNumber.
        self changed: #counterValue
```

Test your application now. Does it work correctly ?

Exercise 1

Adapt the implementation of increment and decrement method in SimpleCounterAppl to use the increment and decrement methods that are already defined on SimpleCounter. Hint: take a look at the AspectAdaptor class if necessary.

12.6 Using the builder at run-time

The builder (class UIBuilder) is the part of the User Interface Builder (UIB) that is responsible for constructing user interfaces from the resources (interface specification, menu specification,...). It is also responsible for helping create the user interface in the canvas editing process, and it provides access to the interface after the user interface is built. We will explore this last functionality by adding extra behavior that disables the decrement button when the value displayed is 0.

Assigning an ID to the button

To disable the button, we will need to talk to the button at run-time. The button is kept in the builder (that we can access at runtime by sending a message builder to self), but we need to give it an identifier to be able to identify it. To give it an identifier, open a canvas tool on the simpleCounter canvas used in previous exercises (use the resources editor or if you use the refactoring browser just look at the class method windowSpec and click on edit), open a properties tool on the decrement button and get the Basics page. Fill in #decrementButton in the ID-field. We will use this identifier later on. Also, on the details page, check the Initially Disabled check box. Install the new canvas and open it.

Enhancing our domain model

Exercise 1

Write a method named isZero that returns whether counterValue equals 0 or not on SimpleCounter in a protocol testing.

OnchangeSend: to:

We now want to be notified when the value of our domain-model changes by using the message onchangeSend: #aSymbol to: anObject that is defined on all value models. It expresses that we want the value model to send the message aSymbol to anObject when its value changes (note that in most of the cases we set anObject to be self). Usually this dependency is set up in the initialize method of the application.

Exercise 1

Set up a dependency to be notified with a message counterValueChanged whenever the value of counterValue changes.

```
SimpleCounterApp>>initialize
    ....
    counterValue onChangeSend: #counterValueChanged to: self
```

To enable or disable the button, you first have to ask the builder for it, and then send enable: aBoolean to it.

Exercise 1

Define the method counterValueChanged so that if the value of the counter is zero, the decrement button is enable. Your code should contain the following expressions:

```
(self builder componentAt: #decrementButton) isEnabled: true
```

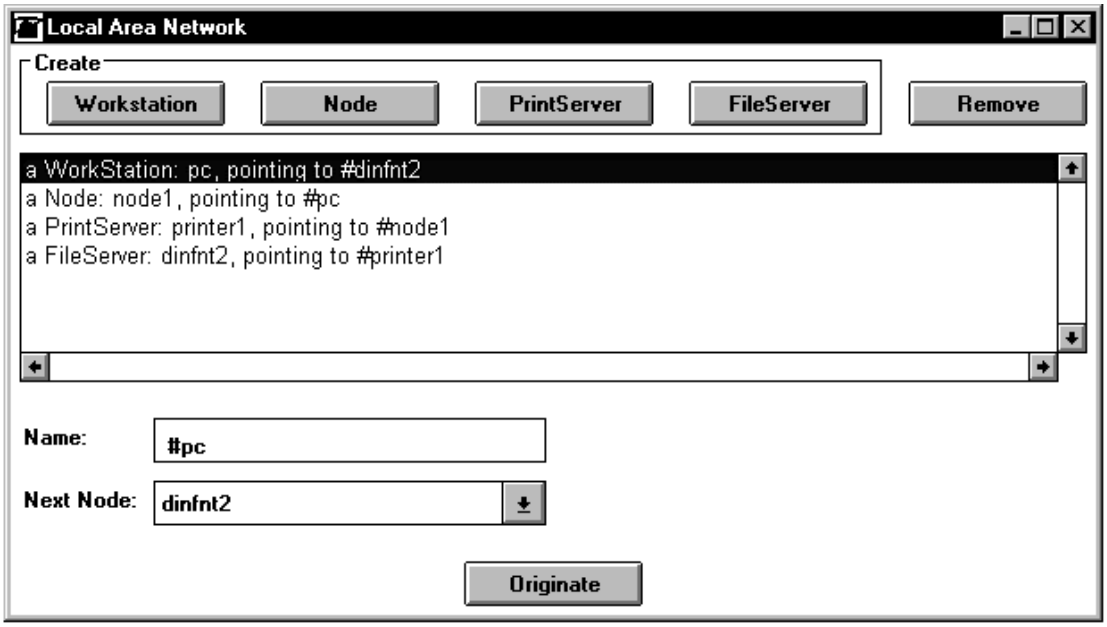
Chapter 13

Building an Interface for the LAN Application

In this exercise you will build a basic interface that allows us to more easily create and run LAN simulations. You will learn how to use some other widgets, more value models and a dialog interface. We will start with an interface for the basic LAN example (containing Node, Workstation, PrintServer and FileServer).

13.1 Overview

This is an example of the application running:



As said before, there are several stages when developing an applicati

- 1.developing the domain model,
- 2.building an interface, and
- 3.programming the application model

13.2 Model

For this exercise, our domain model will be our LAN classes (Node a classes, Packet and subclasses,…). We already have this domain model, doesn't send changed messages.

Exercise **1**

Adapt the class Node and Packet to send change messages. When you consistently used your mutator, this boils down to adding self changed: #nameOfMessageWithoutColons in the body of the mutators.

13.3 Building the interface

We are then ready to build the interface as displayed above and using erties given here:

Action Button		
Basics	Label:	Workstation
	Action:	newWorkstation

Action Button		
Basics	Label:	Node
	Action:	newNode

Action Button		
Basics	Label:	PrintServer
	Action:	newPrintServer

Action Button		
Basics	Label:	FileServer
	Action:	newFileServer

Action Button		
Basics	Label:	Remove
	Action:	remove
	ID:	removeButton

List View		
Basics	Aspect:	nodeList
Details	Scroll Bars	Horizontal, Vertical
	Bordered	On
	Can Tab	On

Label		
Basics	Label:	Name

Input Field		
Basics	Aspect:	nodeName
	Type:	Symbol
Notification	Change:	changedNode

The last raw means that when the value of the inputField is changed, cation get a notification: its method changedNode is invoked. NameMen

Label		
Basics	Label:	Next Node

Menu Button		
Basics	Label:	<none>
	Aspect:	nextNode
	Menu:	deviceNameMenu
Details	Bordered	On
Notifcation	Change:	changedNode

Action Button		
Basics	Label:	Originate
	Action:	originate
	ID:	originateButton

Group Box		
Basics	Label:	Create

Install and define the action and aspect methods of this application.

13.4 Opening the Application, and manually adding some methods

Try to open the application. You will get an exception saying that the ing #deviceNameMenu is not found. The reason is that the define proces not generate menus, and that we have to do it manually. We will there to manually create a method deviceNameMenu (in the aspect protocol) method should return a valueHolder containing a menu. In the beginni

menu will be empty.

Exercise **1**

Using the inspiration of other generated aspect methods, add another instance variable and write the method deviceNameMenu.

Open your application again. Try the different buttons. Afterwards, ty
the input field, and the press tab, return, or select accept in the
operate menu. You should get an exception, because we asked to be no
when the input field changes with a message changedNode, but this is n
erated.

Exercise **1**

Add a method changedNode in a protocol 'private'. For the moment let it return self, just as the other action methods do.

13.5 Programming the application model

The basic action methods

We now have to connect our interface to our domain model. We start
list widget, because it is the most interesting one. A list widget us
InList value model. SelectionInList is a value model with three ins
ables:

- dependents: the dependents of the SelectionInList include at least the list widget. Users might want to become dependent to.
- listHolder: this is a ValueHolder on the list to be displayed in the list view.
- selectionIndex: this is a ValueHolder that contains the index of the currently selected element.

This is not really important to know the instance variable. The impo
sages of SelectionInList are list (returns the list) , list: (to s
tion (returns the current selected element), selection: (to set
selected element of the list), selectionIndex (returns the index of
selected element) , selectionIndex: (to set the index of the selecte
and selectionHolder (returns the selectionHolder).

Exercise **1**

Read the class comment and browse the messages listed above of SelectionInList.

We now implement the action methods to add different kinds of nodes. with the method newNode. In the method body:

- get the list object from your nodeList,
- ask this list to add a new Node,test it

Afterwards

- proceed and implement the messages newFileServer, newPrintServer and newWorkstation.
- implement the method remove (nothing should happen if there is no selection, otherwise the current selection should be removed)

Connecting the name field

In the previous lesson we used an AspectAdaptor to connect our input an instance of SimpleCounter. The AspectAdaptor did the translation the input field (that uses value and value:) and its model (which us Value and counterValue:). We now use AspectAdaptors to let several share a single model.

The model of the name and nextNode widget should be the currently se node in the nodeList. Therefore, if this selection changes, we woul widgets to get updated, and when we fill in and accept a value, th affect the current selection. Therefore, we should create and assig Adaptors for the name and nextNode aspects that both have the same jectChannel. Note that here we will use subjectChannel instead of because the model will be a valueHolder. With a subject, this is the some domain specific element itself).

We will again need to write an initialize method in a protocol call release' to initialize the variables:

- get the selectionHolder object from your nodeList (store it in an temporary variable)
- create a new AspectAdaptor with as subjectChannel the selectionHolder, and a forAspect: of #name. Assign this to the variable nodeName.
- create a new AspectAdaptor with a subjectChannel the selectionHolder, and a forAspect: of #nextNode. Assign this to the variable nextNode.

Open your application, add some nodes, select a node. The input field update. Change the name and select accept in the operate menu (or do return). Deselect the node again and it should update.

Connecting the next node field

When the application is running, and you try to expand the menu butt
ing happens. this is because the menu that is supposed to be there,
should contain the nodes to point to, is still empty. So, we still
this menu.

Note that menus basically contain Association's (an Association is a
pair (look it up)), where the key is the name that is used to displa
and the value is the object you get when asking for the selection. I
the keys will be the names of the nodes, and the values will be the n
selves. Now, you should first check a class MenuBuilder that aids in
menus:

Exercise **1**

Browse the class MenuBuilder (especially the examples at the class side).

Then, go to the method changedNode in the ‘private’ protocol:

- create a new instance of MenuBuilder, and hold it in a temporary variable
- iterate through the nodeList's list adding an Association of "item name -> item" to the MenuBuilder for each item in the list.
- set the value of DeviceNameMenu to be the menuBuilder's menu (use setValue: to do this; using value: the menu button will flash each time you add or change a node)

Test the implementation by creating some nodes, filling in their names
next nodes.

Remove a device from the list. If you do this, you will notice that
contains the removed node ! Modify the remove method to send self ch
Node as the last action in the method. Test your application again.

Chapter 14

Building a Dialog and Originating Packets

In the previous exercise we build a graphical user interface to structure the nodes in the LAN application. We left one thing for this exercise: the originate button. When the user clicks the originate button, we want a dialog box to open that allows us to fill in the originator, addressee and contents of the packet we are going to send. Based on this information, we can then start simulating.

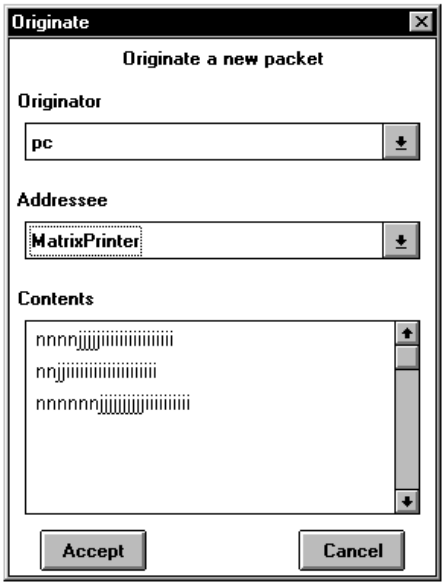
14.1 Dialogs

Custom Dialogs are the least simple VisualWorks applications. A custom dialog can get its resources and widgets from the main application model. Or you can create a separate application model for it, typically a subclass of SimpleDialog. Using the main application model provides tighter integration, since the main model does not need to query a second model for the values that it needs.

You can configure a SimpleDialog dynamically, as we will do in this exercise. This approach is typically used when the widget models needed by the dialog are not needed beyond the lifetime of the dialog. Simple Dialogs are self-contained applications that can be used to collect user input in a controlled way. VisualWorks helps you build the dialog interface, but you must supply the underlying ValueModels to hold the user input until the user selects the Accept button.

14.2 The canvas

Open a new canvas from the Launcher and paint the window shown to the right (there are two menu buttons and a text editor). Then fill in next properties:



Menu Button		
Basics	Label:	<none>
	Aspect:	originator
	Menu:	originatorMenu
Details	Bordered	On

Menu Button		
Basics	Label:	<none>
	Aspect:	addressee
	Menu:	addresseeMenu
Details	Bordered	On

Text Editor		
Basics	Aspect:	contents
Details	Scroll Bars	Vertical
	Bordered	On

Action Button		
Basics	Label:	Accept
	Actions:	accept

Action Button		
Basics	Label:	Cancel
	Actions:	cancel

Define the aspects of the canvas.

Take care when installing the canvas: we are going to install it in our application class (LANInterface), but under a different name than windowSpec (because the interface of our application is stored there, and we do not want to override it, right?). Instead, call the method originateDialogSpec.

Try to open the canvas. You will notice an exception, because the dialog is supposed to work with a SimpleDialog, not the LANInterface itself (a subclass of ApplicationModel). Close the exception and proceed to the next step.

Extending the domain models to support dynamic menus

The two menu buttons will have to show appropriate lists of workstations or outputservers. In fact, we would like to be able to select all nodes that can originate packets or that can do output.

Open a Browser and select the class Node

- Create a new protocol called ‘testing’.
- Add the method canOriginate that returns false.
- Add the method canOutput that returns false.

Select the class Workstation

- Create a new protocol called ‘testing’.
- Override the method canOriginate to return true.

Select the class OutputServer

- Create a new protocol called ‘testing’.
- Override the method canOutput to return true.

We can now ask every node these two questions, and they will answer what's appropriate in their case. These methods allow us to dynamically build menus of the appropriate devices for the user to select when originating a new packet.

Connecting the dialog to the LANINterface

We will start by filling in the originate method. Use following implementation:

originate

```
    | dialogModel dialogBuilder returnVal packet dialogOriginator dialogAddressee dialogContents |
    "the next three lines create ValueHolders to support the three dialog widgets"
    dialogOriginator := nil asValue.
    dialogAddressee := nil asValue.
    dialogContents := String new asValue.

    "next two lines create a new SimpleDialog  object and retrieves the builder"
    dialogModel := SimpleDialog  new.
    dialogBuilder := dialogModel builder.

    "the following  lines connect  the widgets of the interface with the ValueHolders  created"
    dialogBuilder aspectAt: #originator put: dialogOriginator.
    dialogBuilder aspectAt: #addressee put: dialogAddressee.
    dialogBuilder aspectAt: #contents put: dialogContents.

    "the following  lines ask the LANInterface for the originators  and outputters menus. We will write these next, so select proceed  when VisualWorks indicates that they are new messages."
    dialogBuilder aspectAt: #originators put: self originatorsMenu.
    dialogBuilder aspectAt: #addressees put: self addresseesMenu.

    "the following  lines open the dialog interface, originateDialog, and  accept user input"
```

```
returnVal := dialogModel openFor: self interface: #originateDialog.
"returnvalue will be true if the user selected Accept, otherwise it will be false"
returnVal ifTrue: ["create a new packet, fill it in and give it to the workstation"
        packet := Packet send: dialogContents  value to:dialogAddressee   value name.
        packet originator: dialogOriginator  value]
```

We still have to write two messages originatorsMenu and addresseesMenu that have to dynamically create and return a menu. Write these two messages, using canOriginate and canOutput and the hints provided in previous lesson when we wrote the method changedNode (in the section 'Connecting the next node field).

If you want, you can now experiment with other additions:

- disable the remove button, the name field or the next node field if no device is selected
- add a window menu that mimics the buttons on the interface
- catch the #closeRequest message and pop up a dialog asking the user if they really want to close
- …