

Abstract

This document is a collection of Smalltalk exercises that have been developed over the years and that we want to share with others. Note that this document is quite draft. All the sources will be collected and identified clearly.

Smalltalk Exercises

Alexandre Bergel, University of Berne
Noury Bouraqadi, Ecole des Mines de Douai
Marcus Denker, University of Berne
Catherine Dezan, Université de Brest
Stéphane Ducasse, Université de Savoie
Bernard Pottier, Université de Brest
Roel Wuyts, Université Libre de Bruxelles

And many others (please contact stef to update the list) Main Editor: S. Ducasse

March 21, 2006

Contents

I	Robots and Fun	7
1	Fun with Bots Inc	8
1.1	Create and talking	8
1.2	Writing Scripts	8
1.3	Looping	8
1.4	Variables	9
1.5	Methods: Naming Scripts	9
1.6	Composing methods	9
II	Modeling on Paper	11
2	Modeling, modeling...	12
2.1	Classes or Instances	12
2.2	Classification	12
2.3	Capitals and countries	12
2.4	A Basic LAN Application	12
2.5	Mail	13
2.6	Hotel Reservation	13
2.7	Classes/Instances.	13
2.8	Redo the Exercises using UML notation	13
III	First Contact	14
3	Smalltalk Environment Basics	15
3.1	Videos	15
3.2	Starting up	15
3.3	First Impressions	15
3.4	Adding Goodies and setting Preferences	16
3.5	Selecting text, and doing basic text manipulations	17
3.6	Opening a Workspace Window	17
3.7	Evaluating Expressions	17
4	Objects and expressions	19
5	Counter Example	22
5.1	A Simple Counter	22
5.2	Creating your own class	22
5.2.1	Creating a Package	22
5.2.2	Creating a Class	22
5.3	Defining protocols and methods	24
5.3.1	Creating and Testing Methods	24

5.3.2	Adding an instance creation method	25
5.3.3	Adding an instance initialization method	26
5.3.4	Another instance creation method	26
5.4	Saving your Work	27
6	Set, Dictionary et Bag	28
6.1	Collections non-ordonnées	28
6.2	Set	29
6.2.1	Création	29
6.2.2	Accès	29
6.3	Dictionary	29
6.3.1	Création et propriétés héritées de Set	30
6.3.2	Accès, ajouts et suppressions	30
6.3.3	Itérations	31
6.4	Bag	31
6.4.1	Ajouts et suppressions	32
6.4.2	Énumérations	32
6.5	Performances	33
6.5.1	Boucle externe du test et formatage	33
6.5.2	Boucle interne du test	34
6.5.3	Bilan	34
6.6	Décomposition d'un nombre entier en facteurs premiers	35
6.7	Hachage	36
7	SUnit Testing	38
7.1	Set	38
7.2	Dictionary	38
7.3	Bag	38
8	Streams: Les accès	39
8.1	Définition	39
8.2	Créations et accès	39
8.2.1	Créations	39
8.2.2	Extraction du contenu, tests	39
8.2.3	Accès en lecture	40
8.2.4	Ecriture	41
8.3	Application à l'affichage des messages	41
8.4	Cas des fichiers	41
8.5	Exercices	42
8.5.1	Analyse lexicale	42
8.5.2	Analyse de code	42
9	Les objets de Smalltalk-80	43
9.1	Environnement Smalltalk-80	43
9.1.1	Notion de machine virtuelle	43
9.1.2	Accès à l'image et au binaire	43
9.2	Organisation du travail	43
9.3	TP: Observation des objets et règles de priorité	44
9.4	Exercices (A faire en TD)	46
9.4.1	Tableaux	46
9.4.2	Nombres	46
9.4.3	Dates	47
9.4.4	Caractères	47
9.4.5	Chaînes	47

9.4.6	La classe <code>Point</code>	48
10	Les blocs et les énumérateurs	49
10.1	TP	49
10.1.1	Les blocs	49
10.1.2	Les méthodes d'intervalle	49
10.1.3	Les énumérateurs	50
10.1.4	Les structures alternatives	51
10.1.5	Énumérateurs et alternatives	51
10.1.6	Itération de blocs	51
10.2	Exercices (TD)	51
10.2.1	Les blocs	51
10.2.2	Les énumérateurs	52
10.2.3	Les structures alternatives	52
10.2.4	Les intervalles	52
10.2.5	Énumérateurs et alternatives	52
10.2.6	Autres énumérateurs	53
10.2.7	Itération de blocs	53
11	Les collections	54
11.1	Introduction : organisation hiérarchique des collections	54
11.2	TP	54
11.2.1	La classe <code>Collection</code>	54
11.2.2	La classe <code>SequenceableCollection</code>	54
11.2.3	La classe <code>Set</code>	55
11.2.4	La classe <code>Dictionary</code>	56
11.2.5	La classe <code>Bag</code>	57
11.3	Exercices (TD)	58
11.3.1	La classe <code>SequenceableCollection</code>	58
11.3.2	La classe <code>Set</code>	58
11.3.3	La classe <code>Bag</code>	58
11.3.4	La classe <code>Dictionary</code>	58
12	TD1	60
12.1	Rappels	60
12.2	L'objet: c'est du gâteau!	60
12.3	Priorité de messages et variable temporaire	61
12.3.1	Fonctions trigonométriques	61
12.3.2	Maximum	61
12.3.3	Conversion	61
12.4	Tableau et point	62
12.4.1	Analyse de code	62
12.4.2	Exercices sur les tableaux et les points	62
12.4.3	Pour aller plus loin	62
IV	Environnement	63
13	Some Useful Tools in Squeak	64
13.1	SqueakMap Package Loader	64
13.2	Monticello	65
13.3	SqueakSource: the Squeak SourceForge	65

14 Monticello	67
14.1 Packages in Monticello: PackageInfo	67
14.2 Getting Started	67
14.3 Elements of Monticello	68
14.4 Repositories	69
14.5 File Format	70
14.6 The Monticello Browser	71
14.7 The Snapshot Browser	71
14.8 More on PackageInfo	72
 V Seaside	 73
15 Web dynamique avec Seaside	74
15.1 Compléments sur Seaside	74
15.2 Encore des compteurs !	74
15.3 Séparer l'interface du code métier	75
15.4 Une application un peu plus sophistiquée	76
 16 A Simple Application for Registering to a Conference	 77
16.1 RegConf: An Application for Registering to a Conference	77
16.2 Application Building Blocks	78
16.2.1 The Entry Point: RCMain	78
16.2.2 Getting User Information: RCGetUserInfo	78
16.2.3 Getting Hotel Information: RCGetHotelInfo	78
16.2.4 Payment: RCPayment	79
16.2.5 Confirmation: RCConfirmation	79
16.3 Extensions	79
 17 Seaside Tutorial	 81
17.1 Getting Started	81
17.2 Development Tools	81
17.3 Control Flow	81
17.3.1 User Guesses a Number	82
17.3.2 Computer Guesses a Number	82
17.3.3 TicTacToe Game	82
17.4 Components	83
17.4.1 Introduction	83
17.4.2 Choosing a Play	84
17.4.3 Choosing a Show	85
17.4.4 Buying and Printing Tickets	86
17.5 Composition	86
17.5.1 Frame, Subcomponent and Backtracking	86
17.5.2 Reuse of Components	87
17.5.3 Reporting and Batching	87
17.5.4 Editing a Play	87
17.6 Advanced	88
17.6.1 Continuations	88
17.6.2 Bookmark-able URLs	89
 VI Object-Oriented Design	 90
18 A Simple Application: A LAN simulation	91

19	Fundamentals on the Semantics of Self and Super	97
19.1	self	97
19.2	super	98
20	Object Responsibility and Better Encapsulation	99
20.1	Reducing the coupling between classes	99
20.1.1	Current situation	99
20.1.2	Solution.	99
20.2	A Question of Creation Responsibility	100
20.3	Reducing the coupling between classes	101
20.3.1	Current situation	101
20.3.2	Solution.	101
20.4	A Question of Creation Responsibility	101
20.5	Proposing a creational interface	103
20.6	Forbidding the Basic Instance Creation	103
20.6.1	Remarks and Analysis.	104
20.7	Protecting yourself from your children	104
21	Hook and Template Methods	105
21.1	Providing Hook Methods	105
22	Extending the LAN Application	107
22.1	From a Ring to a Star	107
22.2	Handling Loops	107
22.3	Introducing a Shared Initialization Process	108
22.4	Broadcasting and Multiple Addresses	108
22.5	Different Documents	109
22.6	Logging Node	109
22.7	Automatic Naming	109
23	Building an Interface in VW	110
23.1	ApplicationModel: the Glue between Domain and Widgets	110
23.2	Building the interface	110
23.3	Programming the application model	113
23.4	About value models	114
VII	Advanced Smalltalking	116
24	CodeScope	118
25	Implementing Scaffolding Patterns	120
25.1	Introduction	120
25.1.1	Patterns Summary	121
25.2	Pattern: EXTENSIBLE ATTRIBUTES	121
25.3	Pattern: ARTIFICIAL ACCESSORS	123
25.4	Pattern: ARTIFICIAL DELEGATION	124
25.5	Pattern: CACHED EXTENSIBILITY	125
25.6	Pattern: SELECTOR SYNTHESIS	126
25.7	Conclusions	127
25.8	Instructions	127
25.8.1	Interesting Classes and Methods	127
25.8.2	Exercise 0: SUnit	128
25.8.3	Exercise1: Implementation EXTENSIBLE ATTRIBUTES	128
25.8.4	Exercise2: Implementation ARTIFICIAL ACCESSORS	128

25.8.5	Exercise3: Implementation ARTIFICIAL DELEGATION	129
25.8.6	Exercise4: Implementation SELECTOR SYNTHESIS	129
25.8.7	Exercise5: Implementation CACHED EXTENSIBILITY	129
25.9	References	130
26	Generating Bytecodes	131
26.1	Expressions	131
26.2	Parameters	131
26.3	Loops	131
26.4	Instance Variables	132
26.5	Installing a Method in a Class	132
27	Bytecode Analysis	133
27.1	Counting Number of Executed Bytecodes	133
27.2	Methods Coverage Analysis	133
VIII	SmallWiki	135
28	SmallWiki Introduction	136
28.1	Start Smallwiki	136
28.2	Play with Glossary	136
28.3	Diving into SmallWiki Tests	136
28.4	Understanding the Structure	137

Part I

Robots and Fun

Fun with Bots Inc

Main Author(s): Stéphane Ducasse

The goal of this chapter is to get you started with some programming by steering graphical robots. We use the contents of the book *Squeak: Learn programming with Robots* by Stéphane Ducasse, Apress Publishers, 2005 (<http://smallwiki.unibe.ch/botsinc/>).

More information on <http://www.apress.com/book/bookDisplay.html?bID=444>

Getting the environment

Go to <http://smallwiki.unibe.ch/botsinc/> and download the distribution for your machine. Follow the instructions described page <http://smallwiki.unibe.ch/botsinc/installation/> (Unzip it and drag the file name ready.image on the exe file).

Watch the videos you can find at <http://www.iam.unibe.ch/~ducasse/Web/BotsInc/Videos/> and reproduce them during the following exercises.

1.1 Create and talking

Create a robot and talk to it (video 1). You can get the list of action bringing the menu and selecting vocabulary.

- Draw a square of 200 pixels
- Draw a rectangle of 100 and 200 pixels

1.2 Writing Scripts

Direct interaction does not scale. So we propose you to write scripts as shown in the video 3.

- Write a script that draws a square of 200 pixels.
- Write a script that draws a rectangle of 100 and 200 pixels.

1.3 Looping

Read chapter 7 on loop <http://smallwiki.unibe.ch/botsinc/chaptersamples/>

- Using a loop write a script that draws a square of 100.
- Using a loop write a script that draws a rectangle of 100 and 200 pixels.
- Write a script that draws a stair case (experience 7.7)
- Try to reproduce Figure 7-11.

1.4 Variables

Watch video 5 that introduces the use of variables. There the variable `l` is incremented each step of the loop. Redo the experiment shown by the video. Using the same technique, draw a staircase whose step increase regularly. Do not watch at video 6.

1.5 Methods: Naming Scripts

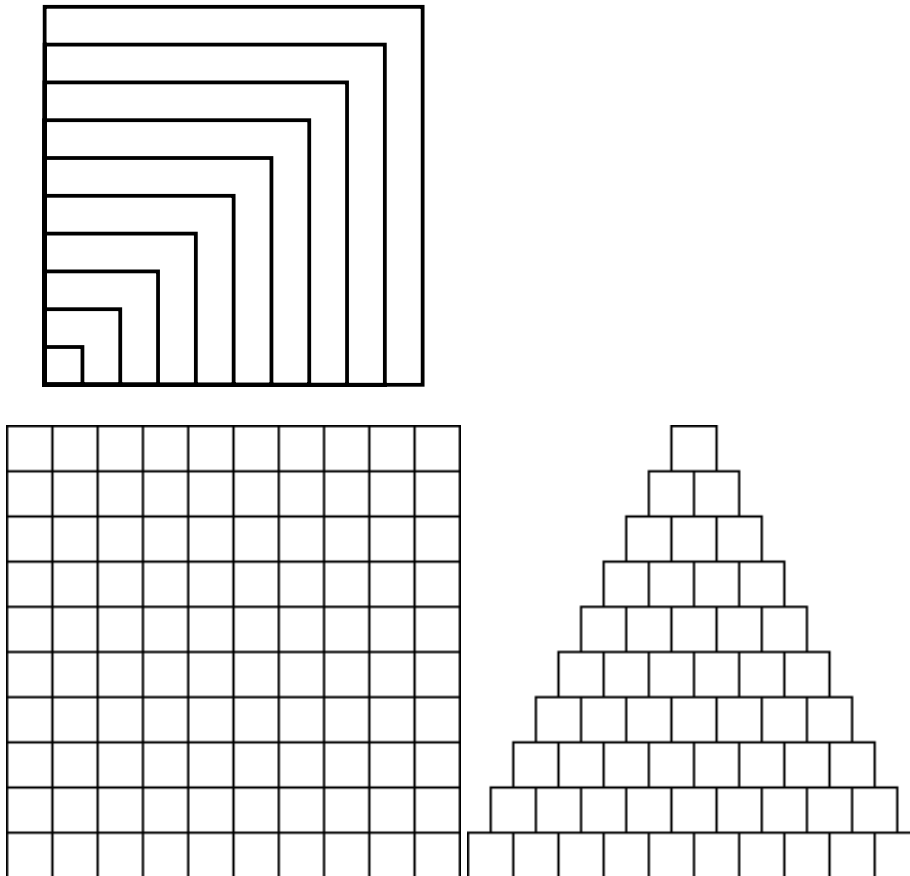
Scripts are powerful but difficult to reuse. First get a code browser (an editor to define method) (video 7, 8 and 9), then read the extra chapter available at http://www.iam.unibe.ch/~ducasse/Teaching/CoursAnnecy/0506-MDSI/4916_Ch12_FINAL.pdf.

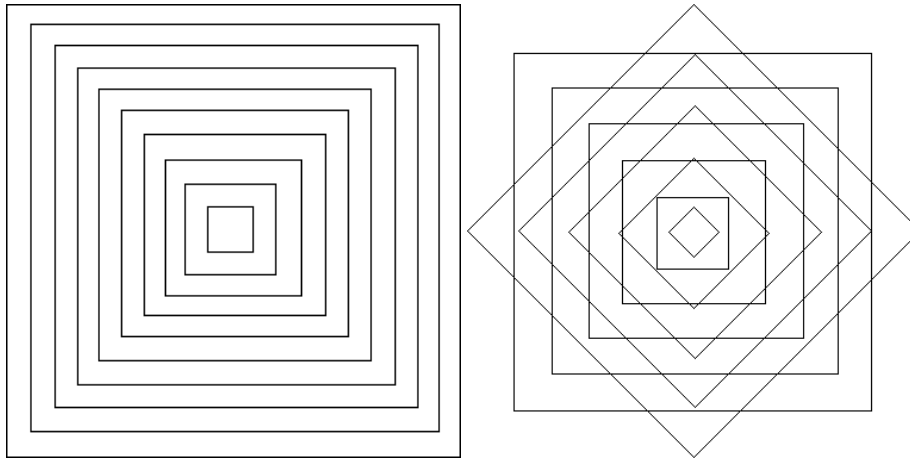
Follow the chapter and define the methods asked there and to the exercises.

1.6 Composing methods

Read chapter 13 available <http://smallwiki.unibe.ch/botsinc/chaptersamples/> and do the exercises.

Now using the method `square` define the following Figures:





Hints: it may be appropriate to define another method `square` that draws a square centered around a point.

Part II

Modeling on Paper

Modeling, modeling...

Main Author(s): Stéphane Ducasse

2.1 Classes or Instances

Classes/Instances. What are instances and what are classes in the following lists:

- Les bijoux de la castafiore, Comix, Asterix le gaulois, Book, Novel, Fahrenheit 451, Da Vinci Code.
- Tokyo, Lisbon, Paris, Capital, Country, State, Madrid, France, Spain, Portugal
- Video, Book, "le fabuleux destin d'amelie Poulain", Client, "la Bible", "Squeak", StarWars4, StarWars5, Dupont, Dupond. GhostInTheShell, LeVoyageDeChihiro.

2.2 Classification

Classify the following abstractions: Opera, Film, Book, Novel, Comics. What could be a possible common superclass? To check if your hierarchy makes sense here is a guideline: It should be possible to substitute any instance of a superclasse by any instance of a subclass without breaking a program of the superclass.

2.3 Capitals and countries

- How do you relate the following entities: Lisbon, Paris, Capital, Country, State, Madrid, France, Spain, Portugal?
- What are the possible relationships from countries to capitals and vice versa? Write some examples.
- How would you characterize countries and states in the context of international income?

2.4 A Basic LAN Application

Identify the classes and their possible responsibilities in the following example. Note that responsibilities are not state or properties of the objects but their behavior.

Imagine a local network simulator **Local Area Network (LAN)**. Workstations, nodes, printers are linked together and form a ring. Each node (workstations, nodes, printers) is pointing to a next node forming a ring. When a packet is received by a node it checks if this is for him, if this is the case, it performs appropriate actions (printing the contents of the packet for a printer), displaying it on a screen if this is a workstation). Only workstation can emit packet to reach other workstations or printers. When a workstation or a printer received a packet that is sent to it it does not forward it to the next node of the network. A packet identifies who sent it, ist addressee and its contents.

2.5 Mail

A mail has a title, one text, one sender, multiple recipients, and multiple attached files. What are the possible classes and their relationships.

2.6 Hotel Reservation

Monsieur Formulain, directeur d'une chaîne d'hôtels, vous demande de concevoir une application de gestion pour ses hôtels. Voici ce que vous devez modéliser :

Un hôtel Formulain est constitué d'un certain nombre de chambres. Un responsable de l'hôtel gère la location des chambres. Chaque chambre se loue à un prix donné (suivant ses prestations).

L'accès aux salles de bain est compris dans le prix de la location d'une chambre. Certaines chambres comportent une salle de bain, mais pas toutes. Les hôtes de chambres sans salle de bain peuvent utiliser une salle de bain sur le palier. Ces dernières peuvent être utilisées par plusieurs hôtes.

Les pièces de l'hôtel qui ne sont ni des chambres, ni des salles de bain (hall d'accueil, cuisine...) ne font pas partie de l'étude (hors sujet).

Des personnes peuvent louer une ou plusieurs chambres de l'hôtel, afin d'y résider. En d'autres termes : l'hôtel héberge un certain nombre de personnes, ses hôtes (il s'agit des personnes qui louent au moins une chambre de l'hôtel...).

2.7 Classes/Instances.

Describe with the best accuracy the information that you would like to have to describe a video for an online catalog. You can get inspired by <http://www.imdb.com/> or <http://wrapper.rottentomatoes.com/>.

Now from the list StarWars4, StarWars5, DVDStarWarsBoite35, DVDStarWarsBoite35, Video, K7, LeVoyageDeChihiro, BoiteNoireLeVoyageDeChihiro145, BoiteNoireLeVoyageDeChihiro146, how would you identify instances and classes. For this item we suggest you to read <http://students.engr.scu.edu/~gvenkata/typeobjectpattern.htm>.

2.8 Redo the Exercises using UML notation

Part III

First Contact

Smalltalk Environment Basics

Main Author(s): Ducasse and Wuyts

3.1 Videos

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

3.2 Starting up

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

There are three important files:

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

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

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

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

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

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

3.3 First Impressions

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

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

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

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

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

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

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

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

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

3.4 Adding Goodies and setting Preferences

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

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

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

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

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

3.5 Selecting text, and doing basic text manipulations

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

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

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

Exercise 1 Try these new selection techniques.

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

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

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

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

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

3.6 Opening a Workspace Window

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

To open a workspace:

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

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

3.7 Evaluating Expressions

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

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

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

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

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

Exercise 7 Print the result of 3+4

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

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

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

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

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

```
self daysInMonth  
self monthName
```

Close the inspector when you are finished.

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

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

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

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

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

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

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

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

Objects and expressions

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

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

3 + 4

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

Date today

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

anArray at: 1 put: 'hello'

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

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

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

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

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

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

Exercise 21 Examine the following expression:

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

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

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

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

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

Exercise 23 Guess what are the results of the following expressions

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

Exercise 24 Examine the following expression:

```
25@50
```

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

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

```
Date today daysInMonth
```

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

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

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

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

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

Exercise 29 Examine the block expression:

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

What is the final result of sum ? How could this piece of code be rewritten to use explicit array indexing (with the method at:) to access the array elements¹? Test your version. Rewrite this code using inject:into:

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

Counter Example

Main Author(s): Bergel, Ducasse, Wuyts

5.1 A Simple Counter

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

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

5.2 Creating your own class

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

When we are using Store, categories are replaced by packages. Therefore in VisualWorks with Store you define a package and define your class within this package. The steps we will do are the same ones every time you create a class, so memorize them well. We are going to create a class `SimpleCounter` in a package called `DemoCounter`. Figure 5.1 shows the result of creating such a package. Note that you all will be versioning your code in the Store database -with the rest of the students of the lecture-, so every package (each one belonging to different group of students) must have a different name. Therefore you should prefix them with your initials or group name.

5.2.1 Creating a Package

In the System Browser, click on the line `Local Image` located in the left-most upper pane (left button of the mouse) and select `New ...Package`. The system will ask you a name. You should write `DemoCounter`, postfixed with your initials or group name. This new package will be created and added to the list (see Figure 5.1).

With the package selected, as shown in Figure 5.1, you can edit its properties by clicking the properties tab of the editor. Properties you will likely have to set one day are the dependencies on other packages (for example, when you subclass a class from another package), post-load actions (an expression that is executed after loading that package from Store, for example to initialize something) and pre-unload actions (an expression that is executed just before unloading a package from your image, for example to close any windows from an application). In the context of this exercise we do not need any of this, so leave the properties alone for now.

5.2.2 Creating a Class

Creating a class requires five steps. They consist basically of editing the class definition template to specify the class you want to create. *Before you begin, make sure that only the package `DemoCounter` is selected.* (See Figure 5.1)

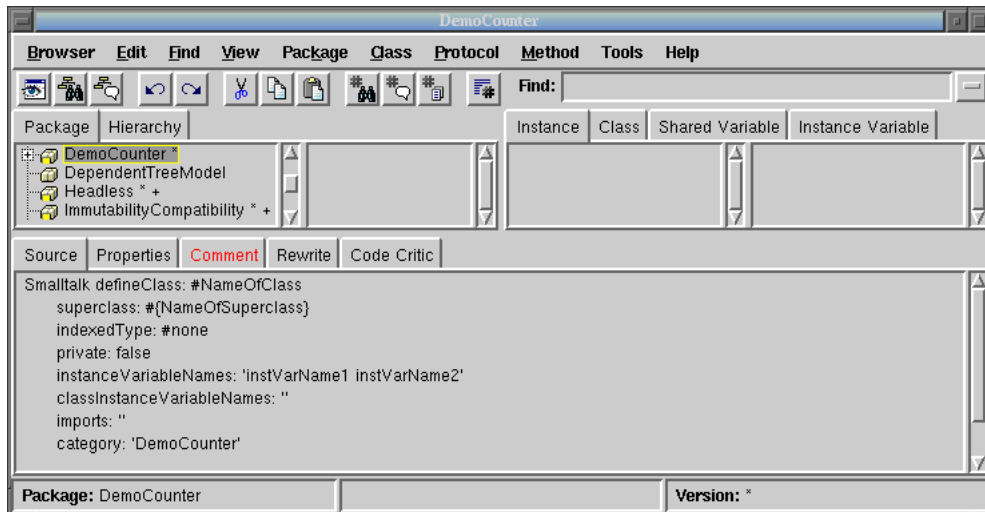


Figure 5.1: Your package is created.

1. **Superclass Specification.** First, you should replace the word `NameOfSuperclass` with the word `Core.Object`. Thus, you specify the superclass of the class you are creating. Note that this is not always the case that `Object` is the superclass, since you may to inherit behavior from a class specializing already `Object`.
2. **Class Name.** Next, you should fill in the name of your class by replacing the word `NameOfClass` with the word `SimpleCounter`. Take care that the name of the class starts with a capital letter and that you do not remove the `#` sign in front of `NameOfClass`.
3. **Instance Variable Specification.** Then, you should fill in the names of the instance variables of this class. We need one instance variable called `value`. You add it by replacing the words `instVarName1` and `instVarName2` with the word `value`. Take care that you leave the string quotes!
4. **Class Variable Specification.** As we do not need any class variable make sure that the argument for the class instance variables is an empty string (`classInstanceVariableNames: ''`).
5. **Compilation.** That's it! We now have a filled-in class definition for the class `SimpleCounter`. To define it, we still have to **compile** it. Therefore, select the **accept** option from the operate menu (right-click button of the mouse). The class `SimpleCounter` is now compiled and immediately added to the system.

As we are disciplined developers, we provide a comment to `SimpleCounter` class by clicking **Comment** tab of the class definition (in the figure 5.1 the **Comment** is highlighted). You can write the following comment:

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

Instance Variables:

`value` <Integer>

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

5.3 Defining protocols and methods

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

5.3.1 Creating and Testing Methods

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

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

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

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

Exercise 30 Decide in which protocol you are going to put the accessor for `value`. We now create the accessor method for the instance variable `value`. Start by selecting the class `DemoCounter` in a browser, and make sure the **Instance** tab is selected (in the figure 5.1, the **Instance** tab is in the middle of the window). Create a new protocol clicking the right-button of the mouse on the pane of methods categories, and choosing **New**, and give a name. Select the newly created protocol. Then in the bottom pane, the edit field displays a method template laying out the default structure of a method. Replace the template with the following method definition:

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

    ^value
```

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

To use a workspace, click on the 'noteblock' icon of the launcher (last icon shown in Figure 5.2).

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

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

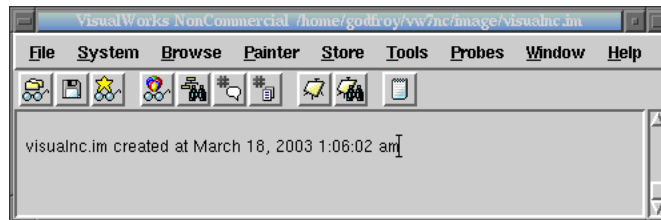


Figure 5.2: The Launcher of VisualWorks.

SimpleCounter new value: 7

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

Exercise 32 Implement the following methods in the protocol operations.

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

Exercise 33 Implement the following methods in the protocol printing

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

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

SimpleCounter new value: 0; increment ; value.

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

5.3.2 Adding an instance creation method

When we create a new instance of the class `SimpleCounter` using the message `new`, we would like to obtain a well initialized instance. To do so, we need to override the method `new` to add a call to an initialization method (invoking an `initialize` method is a very common practice! Ask for the senders of `initialize`). Notice that `new` is always sent to a class. This means that we have to define the new method on the *class side*, not on the *instance side*. To define an instance creation method like the method `new` you should be on the class side, so you click on the **Class** tab (See in the figure 5.1, the **Class** is situated in the same level as the **Instance** tab).

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

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

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

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

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

5.3.3 Adding an instance initialization method

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

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

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

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

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

```
SimpleCounter new increment
```

5.3.4 Another instance creation method

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

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

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

5.4 Saving your Work

Several ways to save your work exist: You can

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

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

Set, Dictionary et Bag

Main Author(s): Bernard Pottier

6.1 Collections non-ordonnées

Les éléments de ces collections ne sont pas rangés selon un ordre prédictible : on ne peut pas accéder aux éléments via une clé externe, tel qu'un index, ou un ordre connu. Cette propriété est liée au mécanisme d'adressage dit *associatif*, où la position d'un élément dans la collection dépend de la valeur de cet élément et de l'historique des accès (voir les explications en section 6.7).

La hiérarchie de classes se présente de la manière suivante:

Object ()

```
Collection ()
  Bag ('contents')
  Set ('tally')
    Dictionary ()
    IdentitySet ()
```

- **Set** : collection garantissant l'unicité des éléments.
- **Dictionary** : accès par une clé, qui est en général un objet d'une classe déterminée. La clé garantit l'unicité.
- **Bag** : chaque élément a un compteur associé,

Une première utilisation de ces collections est la mise en œuvre d'algorithmes très simples reposant uniquement sur leurs propriétés :

Trouver tous les mots apparus dans un texte

```
| bobyADit lesMotsDeBoby |
bobyADit := 'ta pa ta pa tapa tout dit tapa tout dit a ta dou dou'.
lesMotsDeBoby := bobyADit tokensBasedOn: Character space.
lesMotsDeBoby asSet asSortedCollection asArray
```

```
""#('a' 'dit' 'dou' 'pa' 'ta' 'tapa' 'tout')""
```

6.2 Set

6.2.1 Création

Les ensembles sont des collections dynamiques, qui grandissent en fonction des besoins. On crée de nouveaux ensembles par **Set new**, ou, si on est capable d'apprécier correctement une taille idoine, par **Set new: nElements**. Si on connaît déjà les éléments, parce qu'ils sont rangés dans une autre collection, alors on peut aussi instancier par **Set withAll: une Collection**.

L'algorithme (hachage) qui sert à la mise en œuvre des ensembles a de bonnes caractéristiques en temps de recherche d'un élément.

6.2.2 Accès

L'usage simple de Set peut être résumé de la manière suivante :

- **unSet add: unObjet**,
unSet addAll: uneCollection, ajout dans l'ensemble.
- **unSet includes: unObjet** vrai ou faux selon que **unObjet** soit présent ou absent.
- **unSet remove: unObjet** ou
unSet remove: un Objet ifAbsent: unBlocException.

Dans le premier cas, on enlève **unObjet** de **unSet**, si cet objet existe. Sinon, on déclenche une erreur.

Si le programmeur n'est pas certain de la présence de l'objet cherché, il utilise la seconde forme en passant un bloc d'exception, qui peut être vide...

- **unSet size** : nombre d'éléments présents dans l'ensemble,
- **unSet capacity** : capacité de stockage de l'ensemble. Plus le rapport *capacity/size* est grand, plus le temps d'accès risque d'être rapide.

6.3 Dictionary

Un Dictionnaire est un ensemble dont les éléments sont des instances de la classe **Association**, couplant une clé (**key**), et une valeur (**value**). L'unicité d'une association dans un ensemble donné est garantie par l'unicité de la clé, propriété héritée de la classe **Set**.

Les instances de **Association** sont fréquemment créées à l'aide du message binaire **->**. Par exemple, **#boulesRouges->40** associe l'entier 40 au symbole **#boulesRouges**.

La structure de données de dictionnaire implantée par la classe **Dictionary** est intéressante à plusieurs titres:

- Possibilité de désigner un objet par une clé souvent symbolique (**Symbol**, **String**, nombres...)
- Rapidité d'accès, due au hachage (mêmes performances que les ensembles, voir en section 6.5).
- Possibilité d'utiliser les dictionnaires comme des tables de hachage, et pour des clés tout à fait quelconques.
- Dynamisme de la collection qui grandit avec les besoins.

6.3.1 Création et propriétés héritées de Set

On peut se reporter à la description donnée dans **Set**. Si on souhaite utiliser **Dictionary withAll: uneCollection**, il faut cependant retenir que tous les éléments de **uneCollection** doivent être des instances de la classe **Association**. L'exemple qui suit montre comment ajouter un dictionnaire à un autre, comment créer un dictionnaire initialisé.

La clé servant à adresser le dictionnaire est simplement un entier, la valeur est un caractère extrait d'une chaîne. On remplit le premier dictionnaire en notant les index des blancs et les index des ponctuations. Le dictionnaire apparaît alors comme un tableau "creux".

```
testDico1
"self testDico1"

| dico1 dico2 vers |
dico1 := Dictionary new.
dico2 := Dictionary new.
vers := 'le petit homme de la jeunesse, a casse son lacet de soulier,'.
vers keysAndValuesDo:
    [:index :car | car = $
        ifTrue: [ dico1 at: index put: car ]
        ifFalse: [ car isLetter ifFalse: [ dico1 at: index put: car ]]].
dico1 addAll: dico2 associations. "ajout d'un dictionnaire a un autre dictionnaire"
^Dictionary withAll: dico1 associations "creation d'un dictionnaire pre-rempli"
```

Parmi les messages hérités de la classe **Set**, se trouvent les opérations ensemblistes qui portent sur les clés: intersection, union, soustraction... Dans le code ci-dessus, la soustraction des deux dictionnaires serait simplement spécifiée par `dico1 - dico2`.

6.3.2 Accès, ajouts et suppressions

Les méthodes les plus simples pour insérer, supprimer, tester sont les suivantes :

- **unDictionnaire at: uneCle put: unObjet**,
par exemple `dico at: #titi put: 12`.
- **unDictionnaire add: uneAssociation**,
par exemple `dico add: #titi->12`.
- **unDictionnaire removeKey: uneCle**,
par exemple `dico removeKey: #titi`. Le résultat est la valeur associée à la clé (12).
On dispose aussi de la variante **unDictionnaire removeKey: uneCle ifAbsent: unBloc**.
- **unDictionnaire includesKey: uneCle**,
par exemple `dico includesKey: #titi` rendrait `true` dans le contexte ci-dessus.
- **unDictionnaire includes: unObjet**,
par exemple `dico includes: 12` rendrait `true`, alors que `dico includes: #titi` rendrait `false`,
De même **unDictionnaire occurrencesOf: unObjet**, rend le nombre de fois où **unObjet** est apparu en valeur de l'association.
- **unDictionnaire keyAtValue: unObjet** renvoie une clé associée à une valeur, si toutefois cette valeur existe. par exemple `dico keyAtValue: 12` rendrait `#titi`...

6.3.3 Itérations

On peut distinguer les itérations générales, portant sur les *valeurs* plutôt que sur les clés, les itérations portant sur les clés, et enfin celles qui portent sur le couple clé-valeur.

- `unDico do: unBloc`
itération sur toutes les valeurs, par exemple:

```
| dicoDesMots versDePrevert stream |
dicoDesMots := Dictionary new.
stream := WriteStream on: ".
versDePrevert := 'le petit homme qui dansait dans ma tete'.
(versDePrevert tokensBasedOn: $ )
do: [ :mot | dicoDesMots at: mot asSymbol put: mot size ].
dicoDesMots do: [:longueurs | stream nextPutAll: longueurs printString; space].
^stream contents
" '2 4 4 5 7 3 2 5 '"
```

La transformation de la chaîne de caractère en symbole (`asSymbol`) n'est pas indispensable.

- `keysDo:` itère sur les clés du dictionnaires:

```
...
dicoDesMots keysDo: [ :mot | stream nextPutAll: mot; space ].
^stream contents
" 'dans tete homme dansait qui ma petit le '"
```

- `keysAndValuesDo:` permet d'accéder simultanément à la clé et à la valeur:

```
...
dicoDesMots keysAndValuesDo: [:mot :taille |
    stream nextPutAll: mot, '(', taille printString, ')'; space].
..
" 'tete(4) homme(5) ma(2) petit(5) qui(3) le(2) dans(4) dansait(7) '"
```

- `collect:` et fonctionne sur les valeurs associées aux clés.

```
| dicoDesMots versDePrevert |
dicoDesMots := Dictionary new.
versDePrevert := 'les sept éclats de glace de ton rire étoile'.
(versDePrevert tokensBasedOn: $ )
do: [ :mot | dicoDesMots at: mot size put: mot asSymbol ].
^ dicoDesMots collect: [ :mot | mot reverse ]"
" OrderedCollection ('ed' 'not' 'erir' 'ecalg' 'eliote')"
```

- `select:` `unBloc`, `reject:` `unBloc` opèrent sur les valeurs associées aux clés, mais renvoient un dictionnaire.

```
...
^ dicoDesMots select: [:mot | mot size >= 4 ]
" Dictionary (2->#de 3->#ton 4->#rire )"

```

6.4 Bag

Cette classe s'appuie sur `Dictionary` ou `IdentityDictionary` pour compter le nombre d'occurrences d'un élément. L'intérêt est triple: vitesse d'accès (due aux dictionnaires), comptage, et compacité de la structure de données si le nombre des occurrences est élevé.

6.4.1 Ajouts et suppressions

Les sacs (classe Bag) ont un interface permettant d'ajouter ou d'enlever des éléments:

- `unSac add: unObjet`
Le compteur des occurrences de `unObjet` est incrémenté.
- `unSac remove: unObjet`
`unSac remove: unObjet ifAbsent: aBlock`
Le compteur des occurrences de `unObjet` est décrémenté. Si ce compteur tombe à 0, l'objet disparaît du sac.
Dans la seconde version le bloc d'exception est évalué au cas où l'objet n'existe pas: enlever un objet absent est une erreur.
- `unSac addAll: uneCollection` `unSac removeAll: uneCollection`
ajoute ou enlève une collection d'objets.
- `unSac add: newObject withOccurrences: n`
`unSac removeAllOccurrencesOf: anObject ifAbsent: aBlock`
ajoute `n` occurrences du même objet, ou enlève toutes les occurrences d'un objet.

6.4.2 Énumérations

- `do: collect: inject: ...` : Le principe utilisé est d'évaluer les blocs paramètres pour chaque élément de l'ensemble. Par exemple, si `unObjet` apparaît 10 fois dans le sac, il y aura 10 évaluations du bloc paramètre.
Lorsque l'itérateur renvoie une collection, celle-ci est également un `Bag`. Dans l'exemple qui suit voyelles est un `Bag` qui ne comporte que `false` et `true`.

Compter les voyelles dans une chaîne

```
| b voyelles |  
  b := Bag new.  
  b addAll: 'il pleut sans cesse sur Brest'.  
  voyelles := (b collect: [:lettre | lettre isVowel ]).  
  voyelles occurrencesOf: true  
  "8"
```

- `valuesAndCountsDo: unBloc` : S'il est utile d'associer les éléments et leurs compteurs associés, on utilise ce message et on construit un bloc à deux paramètres (élément et son compteur).

Imprimer par ordre croissant le nombre de caractères apparus dans une chaîne

```
| stream sort |  
stream := WriteStream on: (String new:200).  
"preparation d'une collection comportant des Associations triées selon le champs value"  
sort := SortedCollection new sortBlock: [:a1 :a2 | a1 value < a2 value].  
'le petit homme qui chantait sans cesse' asBag valuesAndCountsDo: [:v :c | sort add: v->c].  
sort do: [:association | stream nextPut: association key ; space.  
  stream nextPutAll: association value printString, ' - '].  
stream contents  
"q 1 - p 1 - o 1 - l 1 - u 1 - n 2 - m 2 - h 2 - c 2 -  
i 3 - a 3 - t 4 - s 4 - e 5 - 6 -"
```

6.5 Performances

On veut comparer les performances en insertion et suppression sur les collections `Set`, `SortedCollection` et `OrderedCollection`. Dans un premier temps, on laisse l'ensemble grossir au fur et à mesure des besoins, en testant les 3 collections. Dans un second temps, on fixe d'emblée la taille de l'ensemble à 4 fois la taille des éléments. Dans les deux cas les objets insérés sont des nombres réels aléatoires.

6.5.1 Boucle externe du test et formatage

On y reconnaît un `stream` en écriture sur une chaîne de caractères, qui sera utilisé pour produire une table au format LaTeX (voir la table 6.1).

Le bloc `eval` possède deux paramètres, l'un contient la classe sur laquelle on effectue le test de performance, l'autre est un bloc servant à instancier cette classe. On repère les 6 évaluations de ce bloc en fin de méthode.

Dans le bloc, le temps de calcul insertion/suppression est produit en expédiant le message `testCollection`: `aClass creationBlock: aBlock times: nTimes`. Ce message est décrit plus bas.

Il faut noter que l'on répète `nTimes` fois les opérations et que les temps sont rendus dans une `OrderedCollection` de taille `nTimes`.

```
testNew
  "Test testNew"

  | stream eval resultats nTimes |
  nTimes := 3.
  stream := WriteStream on: ".
  stream nextPutAll: 'Classe & '.
  nTimes timesRepeat: [stream nextPutAll: 'add', ' & ', 'remove', '& '].
  stream nextPutAll: 'add moy.', ' & ', 'remove moy.', '\\\\hline'; cr.
  eval :=
    [ :aClass :aBlock |
      | sumX sumY |
      stream nextPutAll: aClass name, ' & '.
      resultats := self
        testCollection: aClass
        creationBlock: aBlock
        times: nTimes.
      sumX := sumY := 0.
      resultats
        do:
          [ :point |
            stream nextPutAll: point x printString, ' & ', point y printString, ' & '.
            sumX := sumX + point x.
            sumY := sumY + point y ].
            stream nextPutAll: (sumX // resultats size) printString, ' & ',
              (sumY // resultats size) printString, '\\\\hline'; cr ].
    eval value: Set value: [ Set new ].
    eval value: SortedCollection value: [ SortedCollection new ].
    eval value: OrderedCollection value: [ OrderedCollection new ].
    eval value: Set value: [ Set new: 8000 ].
    eval value: SortedCollection value: [ SortedCollection new: 8000 ].
    eval value: OrderedCollection value: [ OrderedCollection new: 8000 ].
    ^stream contents
```

6.5.2 Boucle interne du test

Ici, on effectue une série de mesures identiques sur une classe passée en paramètre, que l'on instancie via un bloc également passé en paramètre.

tRand est le temps de génération de 2000 nombres. *tAdd* est le temps d'insertion, *tRemove*, le temps de suppression.

Le temps est évalué en millisecondes en utilisant le message *Time millisecondsToRun: unBloc*. On renvoie une collection de points comportant en abscisse le temps d'insertion, en ordonnée le temps de suppression (par pure commodité).

```
testCollection: collClass creationBlock: aBlock times: n
| rand set2 tAdd tRemove tRand set1 resultats |
resultats := OrderedCollection new: n.
rand := Random new. "Générateur de nombre aleatoire"
n
timesRepeat:
[ set2 := aBlock value.
tRand := Time millisecondsToRun: [2000 timesRepeat: [rand next * 100]].
tAdd := Time millisecondsToRun: [2000 timesRepeat: [set2 add: rand next * 100]].
set1 := set2 copy asArray.
tRemove := Time millisecondsToRun: [set1 do: [:elt | set2 remove: elt]].
resultats add: tAdd - tRand @ tRemove].
^resultats
```

6.5.3 Bilan

Classe	add	remove	add	remove	add	remove	add moy.	remove moy.
Set	514	946	503	908	507	920	508	924
SortedCollection	769	2983	805	1564	759	2787	777	2444
OrderedCollection	50	1574	41	2876	17	1568	36	2006
Set	267	361	244	353	264	349	258	354
SortedCollection	1380	2869	852	1551	1326	2846	1186	2422
OrderedCollection	49	1544	5	2887	53	1613	35	2014

Table 6.1: Performances comparées. Le premier groupe de tests porte sur des collections créées avec **new**. Le second groupe porte sur des ensembles portés, d'emblée à une capacité de 8000

- On peut constater que le coût du sous dimensionnement de la structure du départ peut être important : dans le cas de Set, cela compte pour un facteur 2.
- Si la structure est grande, le coût en suppression est réduit (le second Set est 4 fois trop grand, et on remarque que les suppressions sont à peine plus lentes que les insertions). Dans le cas contraire, le coût des suppressions peut être jusque deux fois plus élevé que celui des insertions.
L'explication se trouve dans la gestion des collisions et le nombre de celles-ci...
- l'accès en insertion sur les **OrderedCollection** est très rapide, on ne peut pas en dire autant des suppressions.
- les **SortedCollection** marquent un avantage en recherche qui n'apparaît pas ici.

6.6 Décomposition d'un nombre entier en facteurs premiers

Décrire un algorithme de décomposition s'appuyant sur un ensemble de diviseurs possibles. L'ensemble de ces diviseurs est réduit chaque fois qu'un de ses éléments est traité. S'il s'agit d'un diviseur, on réduit les nombres de l'ensemble à l'aide de ce diviseur. Dans le cas contraire on élimine tous les multiples du diviseur. Les diviseurs sont accumulés dans un *Bag*.

```
Integer methodsFor: 'set-exemple'!  
decompose  
| diviseurs diviseursPossibles nb minBlock min maxBlock max sizeColl |  
diviseurs := Bag new.  
diviseurs add: 1. nb := self.  
sizeColl := OrderedCollection new.  
diviseursPossibles := (2 to: self // 2) asSet.  
min := 2. max := 1.  
minBlock := [:number | min := min min: number].  
maxBlock := [:number | max := max max: number].  
[diviseursPossibles isEmpty or: [nb = 1]]  
whileFalse:  
    [| diviseur division |  
    sizeColl add: diviseursPossibles size.  
    diviseur := min.  
    division := nb / diviseur.  
    min := nb // max.  
    (division isKindOf: Integer)  
    ifTrue:  
        [diviseurs add: diviseur.  
        maxBlock value: diviseur.  
        minBlock value: diviseur.  
        nb := division.  
        diviseursPossibles := diviseursPossibles  
        collect:  
            [:n |  
            | x | x := n / diviseur.  
            x := (x isKindOf: Integer) ifTrue: [x] ifFalse: [n].  
            x > (nb / max)  
            ifTrue: [x := min]  
            ifFalse: [(#(0 1) includes: x) ifTrue: [x := max]].  
            minBlock value: x.  
            x]]  
    ifFalse: [diviseursPossibles := diviseursPossibles  
    reject:  
        [:n | minBlock value: n.  
        ((n / diviseur isKindOf: Integer) or: [n > (nb / max)])  
        or: [n = diviseur]]]].  
Transcript cr; show: sizeColl printString ; cr.  
^diviseurs  
"(3*25*8*10*4) decompose  
OrderedCollection (11999 5999 2999 1499 749 374 186 93 93 20 13 13 1 1)  
Bag (1 2 2 2 2 2 3 5 5 5)"
```

6.7 Hachage

Le hachage d'un objet permet d'adresser une table par le contenu. Cette technique permet d'éviter des opérations de recherche par énumération ou dichotomie. Cette technique est à la base des **Set**, donc de **Dictionary**, donc de **Bag** qui encapsule un dictionnaire.

La présente section décrit sommairement le mécanisme. La réalisation d'algorithmes performants requière une plus grande attention que cette première approche.

Les trois figures qui suivent montrent une opération de recherche :

- On utilise une fonction qui rend un index à partir d'un objet (message **hash** expédié à l'objet). Des exemples très simples de hachage sont un modulo appliqué à un objet entier, une fonction $c1 \times 256 + c2$ appliquée aux deux premiers caractères d'une chaîne. . .

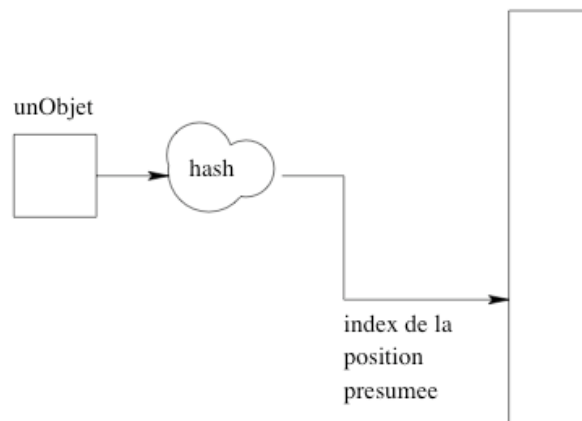


Figure 6.1: L'objet est haché et on obtient un index dans la table

- L'index présumé ayant été produit, on accède à la table. Si la table ne contient rien (nil), alors l'objet n'est pas dans la table. Sinon on compare l'objet cherché et l'objet trouvé ($a = b$). Voir figure 6.2. S'il y a égalité, on a trouvé l'objet. . .

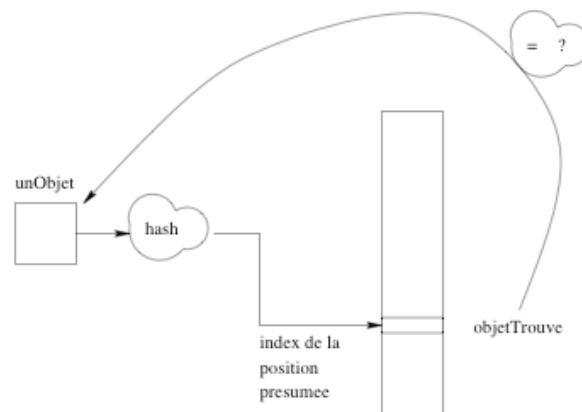


Figure 6.2: On compare l'objet implanté dans la table et l'objet cherché

- Sinon, on descend en séquence en cherchant soit nil, soit l'égalité (voir figure 6.3).

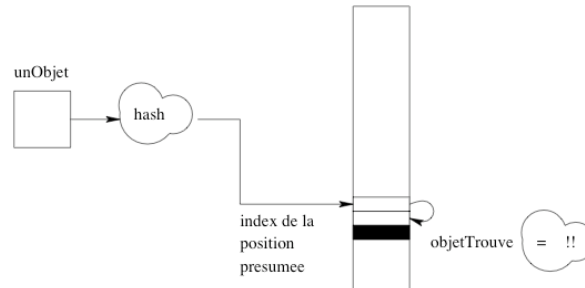


Figure 6.3: Recherche en séquence

Les opérations d'insertion sont menées de la même manière, il s'agit ici de trouver le premier index libre lors de la recherche en séquence. Lorsque la quantité de collisions grandit (place libre de plus en plus petite), il est préférable de reconstruire la table en doublant sa taille.

Les suppressions requièrent une reconstruction au moins partielle de la table.

SUnit Testing

Main Author(s): Stéphane Ducasse

For each class described in the Chapter on collection "Set, Dictionary and Bag", define unit tests for each of the described behavior.

7.1 Set

Exercise 35 create a Unit test for each of the following methods

- aSet add: anObject and aSet addAll: aCollection,
- aSet includes: anObject
- aSet remove: un Objet and aSet remove: anObjet ifAbsent: aBlocException,
- aSet size and aSet capacity.

7.2 Dictionary

- aDictionary at: aKey put: anObject,
- aDictionary add: uneAssociation,
- aDictionary removeKey: aKey, aDictionary removeKey: aKey ifAbsent: aBloc,
- aDictionary includesKey: aKey,
- aDictionary includes: anObject,
- aDictionary occurrencesOf: anObject,
- aDictionary keyAtValue: anObject,
- aDictionary do: aBloc,
- aDictionary keysDo: aBloc,
- aDictionary keysAndValuesDo: aBloc.

7.3 Bag

- aBag add: unObjet
- aBag remove: unObjet and aBag remove: unObjet ifAbsent: aBlock
- aBag addAll: aCollection and aBag removeAll: aCollection
- aBag add: anObject withOccurrences: n and aBag removeAllOccurrencesOf: anObject ifAbsent: aBlock

Streams: Les accs

Main Author(s): Bernard Pottier, Université de Brest

8.1 Définition

Stream et les sous-classes de **Stream** fournissent un mécanisme d'accès séquentiel sur des objets, internes ou externes.

Lorsqu'une instance de **Stream** est utilisée, elle a seule le contrôle de l'objet accédé, qui ne devrait pas être utilisé directement. La resynchronisation de l'objet lu ou écrit peut impliquer une opération de mise à jour (**flush**, ou **commit** pour assurer la mise à jour d'une entrée-sortie...). La raison est que le mécanisme séquentiel tamponne les données en lecture ou en écriture afin d'économiser les transferts.

8.2 Créations et accès

8.2.1 Créations

On crée un stream¹ en s'appuyant sur des classes telles que **ReadStream**, **WriteStream**, **ReadWriteStream**, lors que l'objet est interne. D'autres classes sont utilisées pour les accès séquentiels externes, pour les entrée-sorties. Certains objets savent rendre des **Stream** en réponse au message **readStream** ou **writeStream**; c'est le cas des fichiers **Filename**.

On obtient un stream en expédiant le message **on**: à la classe à instancier. On passe un objet vide lorsque l'on veut écrire, une collection séquençable pleine lorsque l'on veut lire. Le stream s'occupe de faire grossir votre objet lorsque le besoin s'en fait sentir, en respectant sa classe.

- **ReadStream on:** 'A Plouescat, la joie eclate!'
- (**WriteStream on:** (Array new: 3))

8.2.2 Extraction du contenu, tests

Le contenu du stream ne devrait pas être accédé directement. Il faut récupérer ce contenu en expédiant le message **contents** au stream. L'exemple qui suit montre l'insertion et l'extraction de l'objet:

```
| monStream |
monStream := ReadStream on: 'A Plouescat, la joie eclate!'.
monStream contents. " 'A Plouescat, la joie eclate!'"
```

On peut noter qu'il est possible de savoir si on est au bout d'une lecture (fin de fichier, dernier élément d'un objet), à l'aide du message **atEnd**. On peut se positionner en fin de stream à l'aide de **setToEnd**. La position dans le stream est contrôlable à l'aide du message **position**.

¹ en français, un flôt

```
| monStream |
monStream := ReadStream on: 'A Plouescat, la joie eclate!'.
monStream atEnd. "(printlt) false"
monStream setToEnd ; position. "(printlt) 28"
monStream position: 12; position "(printlt) 12"
```

8.2.3 Accès en lecture

Lecture séquentielle d'un élément

La lecture est opérée en expédiant le message `next`. Le stream fait progresser son index interne et effectue d'autres opérations si nécessaire.

```
| array stream |
array := #( 2 3 6 7 9 0 ).
stream := ReadStream on: array.
stream next. stream next. " 3"
```

lecture d'une séquence

On peut prélever un nombre arbitraire d'éléments en utilisant le message `nextAvailable: unEntier`. Bien entendu, on ne peut dépasser la capacité de l'objet accédé, si celle-ci est bornée.

```
| array stream |
array := #( 2 3 6 7 9 0 ).
stream := ReadStream on: array.
stream nextAvailable: 3. " (printlt) #(2 3 6)"
stream nextAvailable:12." (printlt) #(7 9 0)"
```

lecture d'une séquence sur condition

Il est souvent intéressant de prélever des éléments jusqu'à ce qu'une condition soit obtenue.

- `upToEnd`: épuise le contenu de l'objet.

```
| array stream |
array := #( 2 3 6 7 9 0 ).
stream := ReadStream on: array.
stream nextAvailable: 3. " (printlt) #(2 3 6)"
stream upToEnd." (printlt) #(7 9 0)"
```

- `through: limit`: lit jusque la prochaine occurrence de l'objet `limit`.

```
| stream |
stream := ReadStream on: 'C'est a Morlaix que l'on se plait'.
stream through: '$'.
stream through: '$'. " (printlt)'est a Morlaix que l'""
```

8.2.4 Ecriture

En écriture, on envoie `nextPut: unElement`, qui insère l'élément au bout de l'objet. On peut provoquer l'insertion d'une collection d'éléments avec `nextPutAll: uneCollection`

```
| stream |  
stream := WriteStream on: (Array new: 4).  
stream nextPut: 3/2.  
stream nextPutAll: #( 2 3 6 7 9 0 ).  
stream contents " #((3/2) 2 3 6 7 9 0)"
```

8.3 Application à l'affichage des messages

La fenêtre Transcript sert à l'affichage des messages. On peut obtenir le texte contenu dans cette fenêtre via le message `value`:

Transcript value asString

On écrit dans le Transcript comme dans un stream de caractères:

```
Transcript nextPut: Character cr.  
Transcript nextPutAll: 'A Plouarzel, on fait du ze!e'.  
Transcript flush.
```

Noter que le message `flush` est indispensable pour provoquer l'affichage immédiat des caractères insérés. Les Stream savent gérer l'insertion de caractères non imprimables plus simplement:

```
Transcript cr; tab; nextPutAll: 'A Plouarzel, on fait du ze!e';  
space;cr; flush
```

8.4 Cas des fichiers

Un fichier est repéré par un *nom*, instance de la classe `Filename`. On peut choisir un nom de fichier (String) via un dialogue spécialisé:

```
| nom |  
nom := Dialog requestFileName: 'nom du fichier?'
```

Une fois ce nom choisi, on peut lui associer un fichier en convertissant la chaîne en une instance de `Filename`:

```
| nom fileName |  
nom := Dialog requestFileName: 'nom du fichier?'.  
fileName := nom asFilename.
```

On obtient ensuite aisément un stream en lecture ou écriture en expédiant les messages `writeStream`:

```
| stream |  
stream := 'Transcript.file' asFilename writeStream.  
stream nextPutAll: Transcript value.  
stream commit ; close.
```

ou `readStream`, en lecture:

```
| stream nomDuFichier |  
nomDuFichier := Dialog requestFileName: 'nom du fichier?'.  
stream := nomDuFichier asFilename readStream.  
Transcript nextPutAll: stream contents.  
Transcript flush. stream close.
```

8.5 Exercices

8.5.1 Analyse lexicale

En phrases

On peut obtenir le texte de la fenêtre Transcript en expédiant le message `value`. Construire une collection des phrases contenues dans ce texte:

```
| rStream phrases |  
rStream := ReadStream on: Transcript value asString. "lecture sur la chaine"  
phrases := WriteStream on: (Array new: 100). "écriture sur des tableaux"  
[ rStream atEnd ] whileFalse:  
    [ phrases nextPut: ( rStream upTo: $. ) ].  
phrases contents
```

Renverser les mots

On utilise le texte de la fenêtre Transcript. Produire un texte où les mots sont écrits à l'envers. . .

```
| rStream texte ligneStream mot |  
rStream := ReadStream on: Transcript value asString. "lecture sur la chaine"  
texte := WriteStream on: (String new: 1000). "écriture du nouveau texte."  
[ rStream atEnd ] whileFalse:  
    [ ligneStream := ReadStream on: ( rStream upTo: Character cr ).  
      [ ligneStream atEnd ] whileFalse:  
          [ mot := ligneStream upTo: Character space .  
            texte nextPutAll: mot reverse ; space ].  
          texte cr.  
    ].  
texte contents
```

8.5.2 Analyse de code

D'autres classes ont des comportements de Stream. C'est par exemple le cas de l'analyseur lexical Smalltalk, qui peut être réutilisé à d'autres tâches:

```
| texteScanner item items |  
item := ''.  
items := OrderedCollection new.  
texteScanner := Scanner new on: 'begin Carthago delenda est. 7 +9 = 15 . end' readStream.  
[ item = 'end' ]  
    whileFalse: [ item :=texteScanner scanToken. items add: item].  
items
```

Les objets de Smalltalk-80

Main Author(s): to be fixed: B. Pottier, Université de Brest, Bernard.Pottier@univ-brest.fr

9.1 Environnement Smalltalk-80

9.1.1 Notion de machine virtuelle

Smalltalk est non seulement un langage de programmation mais aussi un système d'exploitation, un environnement de développement et une bibliothèque de code très importante.

Comme Smalltalk est un interpréteur, il a besoin de conserver le texte initial (fichier *source*) et sa forme interne (fichier *image* - format byte code). Les modifications effectuées sont enregistrées dans un fichier *change*. La représentation interne générée peut être vue comme un langage d'assemblage pour une machine virtuelle.

9.1.2 Accès à l'image et au binaire

L'image et le binaire se trouve :

```
C:
  vwncl30
    vwncl30
      bin
      image
```

Pour lancer une image, cliquer 2 fois sur l'icône de visualnc.im dans le repertoire image

9.2 Organisation du travail

Utilisation des boutons de la souris :

bouton de gauche : sélection de zone, activation de fenêtre

bouton de droite : pop-up menu (menu contextuel associé à la fenêtre)

bouton de droite dans la barre de titre : menu d'édition de la fenêtre

Vous utiliserez le *FileList* se trouvant dans l'icône bleue représentant un tiroir pour créer des nouveaux fichiers (créer a:TP1.txt) dans le répertoire Smalltalk.

Ecrire le texte du fichier (les exercices commentés) dans la fenêtre du bas et sauvegarder en utilisant le menu contextuel ("Pop Up Menu"- bouton milieu de la souris- et *Save as*) de la fenêtre.

Toute sauvegarde doit être faite sur votre disquette, soit sous *a* :

9.3 TP: Observation des objets et règles de priorité

Evaluer : sélectionner une zone et faire “print it”

Inspecter : sélectionner une zone et faire “inspect”

- Inspecter les expressions suivantes :

1
2.0
\$a
'une chaine'
1@2
1.0@2.0
7/2

Parmi les messages, on distingue

les messages unaires comme new, sin, sqrt, size, first, last, negated)

les messages binaires + - * / ** // \< <= > >= = ~= == ~~ & | @ ,

les message à mot clé comme at: put:, x: y:, bitOr:, bitAnd:

Dans une expression, on évalue en priorité en respectant le parenthésage, les messages unaires puis binaires puis à mots clés. Si l’expression ne comporte que des messages de même priorité, l’évaluation se fait classiquement de la gauche vers la droite.

- Evaluer et inspecter les expressions suivantes :

7.0/2.0
1 + 1
(1 + 1) printString
(1/2) class

Expliquer pourquoi le parenthésage est obligatoire.

- Pas de priorité des opérateurs, l’évaluation suit l’ordre des messages. Evaluer :

2 + 3*4
2 + (3*4)
2 + 1/2
2 + (1/2)

- Uniformité des messages. Un même message peut être adressé à des objets de types différents. Evaluer et inspecter :

2 sqrt
2.0 sqrt
(3/2) sqrt
(3/5) + (6/7)

- Arithmétique exacte et conversion de type. Evaluer :


```
aPoint:= Point x:2 y:1.  
aPoint x: aPoint x * 2
```

```
| x |  
x:=1.5.  
x negated rounded.  
Fraction numerator: x*2 denominator: 3 + x negated rounded.
```

9.4 Exercices (A faire en TD)

9.4.1 Tableaux

1. Multiplier par 2 le 2^{ème} élément d'un tableau,
2. Remplacer la valeur du 2^{ème} élément d'un tableau par son opposé.
3. Remplacer la valeur du 3^{ème} élément par la valeur du 2^{ème} élément.
4. Remplacer la valeur du 3^{ème} élément par la somme des 2^{ème} et 3^{ème} (ancienne valeur) éléments.
5. Le 2^{ème} du tableau étant une fraction, remplacer cette fraction par la fraction inverse dans le tableau.

9.4.2 Nombres

Maximum

La méthode `max`: `unAutreNombre` appliqué à un nombre renvoie le plus grand des deux nombres.
Exemple : `2 max: 6` renvoie 6.

1. Calculer le maximum de 3 variables `a b c` contenant des valeurs quelconques.
2. Calculer le maximum de 3 variables `a b c` contenant des valeurs quelconques **sans utiliser de variables intermédiaires**

Fonctions trigonométriques

Un nombre (en radians) comprend les messages correspondant aux fonctions trigonométriques `sin` `cos` `tan` `arcSin` `arcCos` `arcTan`.

On convertit un nombre de Degré à Radian en lui envoyant le message `degreesToRadians`.

Calculer (à l'aide d'une fonction trigonométrique) le côté d'un carré dont la diagonale mesure 1.41421 mètres.

Conversion Celsius-Fahrenheit

La formule de conversion Celsius-Fahrenheit est : $C = (5/9) (F - 32)$.

Convertir une variable contenant un nombre (en degrés Fahrenheit), en degrés Celsius.

Conversion binaire-hexadécimale

`//` est l'opérateur de division entière.

`\\` est l'opérateur de modulo.

Soit un nombre hexadécimal (de 0 à 15 en base 10, de 0000 à 1111 en base 2), on désire le convertir en binaire (sans utiliser `printStringRadix`).

En effectuant une série de divisions entières, les chiffres binaires sont obtenus **de la droite vers la gauche** grâce au reste de la division entière (le modulo).

1. Appliquer cet algorithme pour convertir le nombre hexa 15 en base 2.
2. Vérifier en utilisant `printStringRadix`:

9.4.3 Dates

On obtient la date du jour grâce au message `today` envoyé à la classe `Date`.

```
| d |  
d := Date today.
```

On peut créer une date grâce au message `newDay:unNumeroJour monthNumber:unNumeroMois year:unNumeroAnnee` envoyé à la classe `Date`.

On peut aussi créer une date grâce au message `fromDays:nombreJours` envoyé à la classe `Date`.

La méthode `asDays` envoyé à une date renvoie le nombre de jours depuis le début du siècle (1/01/1901).

```
| d |  
d := Date newDay:12 monthNumber:10 year:1998.  
d asDays. "35713"
```

1. Calculer le nombre de jours que vous avez déjà vécu.
2. Calculer quelle serait la date, si vous aviez l'âge que vous avez aujourd'hui et si vous étiez né le 1 Janvier 1901.

9.4.4 Caractères

On peut créer un caractère à partir de son code ASCII en envoyant le message `value: unCodeAscii` à la classe `Character`

On obtient le code ASCII d'un caractère en lui envoyant le message `asInteger`

```
| c |  
c := Character value: 65.  
c asInteger. "65"
```

1. Calculer le code ASCII de \$a et de \$A
2. Convertir un caractère de minuscule à majuscule

9.4.5 Chaînes

On obtient une chaîne à partir d'un caractère en envoyant le message `with: unCaractere` à la classe `String`.

Le caractère "blanc" s'obtient en envoyant le message `space` à la classe `Character`.

On concatène des chaînes avec l'opérateur ,

```
| s1 s2 |  
s1 := String with: $1.  
s2 := String with: $2.  
s1, s2      "12"
```

1. Fabriquer une chaîne contenant le caractère "blanc"
2. Fabriquer une chaîne contenant votre prénom, un caractère "blanc", votre nom

9.4.6 La classe **Point**

Dans la classe **Point**, les messages unaires **r** et **theta** permettent de récupérer les coordonnées polaires d'un point.

1. Définir une translation de point par rapport à un vecteur donné.
2. Définir une homothétie.
3. Calculer l'angle formé par deux vecteurs.

Les blocs et les enumerateurs

Main Author(s): to be fixed: B. Pottier, Université de Brest, Bernard.Pottier@univ-brest.fr

10.1 TP

On affiche une chaîne dans le Transcript en lui envoyant le message `show: uneChaine`. On passe à la ligne dans le Transcript en lui envoyant le message `cr`

Exemple : Transcript show: 'Salut'. Transcript cr.
en utilisant la cascade (plusieurs messages envoyés au même objet)
Transcript show: 'Salut'; cr.

10.1.1 Les blocs

L'évaluation d'un bloc sans paramètre s'obtient en envoyant le message `value` au bloc. Les instructions du bloc sont exécutées et le résultat de la dernière instruction est retourné.

Un bloc avec une variable s'évalue en envoyant le message `value: uneValeur`.

Un bloc avec deux variables s'évalue en envoyant le message
`value: unePremiereValeur value: uneDeuxiemeValeur` (ceci jusqu'à 4 variables).

1. Ecrire un bloc (sans variables) qui calcule la racine carrée de 1, puis de 2, puis de 3. Evaluer ce bloc.
2. Ecrire un bloc avec une variable, qui renvoie le carré de cette variable. Evaluer ce bloc.
3. Ecrire un bloc avec deux variables, qui renvoie la plus grande de ces deux variables (utiliser la méthode `max:`). Evaluer ce bloc.

10.1.2 Les méthodes d'intervalle

Les nombres comprennent les messages suivants : `to: uneValeurArret do: unBloc`
et : `to: uneValeurArret by: unPas do: unBloc`

Exemple qui affiche les nombres de 1 à 10 dans le Transcript

```
1 to: 10 do: [:i | Transcript show: i printString]
```

Exemple qui affiche, par pas de 2, les nombres de 1 à 10 dans le Transcript

```
1 to: 10 by: 2 do: [:i | Transcript show: i printString]
```

1. Convertir le nombre 65 (représentant un code ASCII) en `Character (asCharacter)`, puis en `Symbol (asSymbol)`, puis en `String (asString)`.
2. Afficher dans le Transcript les caractères compris entre les codes ASCII 65 et 122 (bornes incluses).
3. Afficher les nombres impairs de 1 à 33 dans le Transcript

10.1.3 Les énumérateurs

Toutes les sous-classes de **Collection** comprennent ces messages, appelés *énumérateurs* :

- **do**: unBloc évalue unBloc sur chaque élément de la collection,
- **collect**: unBloc comme **do**: mais renvoie une collection des résultats,
- **select**: unBloc évalue unBloc sur chaque élément et renvoie ceux pour qui l'évaluation renvoie **true**,
- **reject**: unBloc évalue unBloc sur chaque élément et renvoie ceux pour qui l'évaluation renvoie **false**,
- **detect**: unBloc renvoie le premier élément pour qui l'évaluation renvoie **true**,
- **detect: unBloc ifNone: unAutreBloc** a le même comportement que **detect**: mais permet d'exécuter le deuxième bloc (unAutreBloc) s'il n'y a pas d'élément pour qui l'évaluation de unBloc renvoie **true**.
- **inject: uneValeur into: unBlocBinaire** injecte le résultat de l'exécution précédente de unBlocBinaire (un bloc à deux paramètres) dans la suivante.
uneCollection inject: valeur into: [arg1 arg2 ...]
arg1 est initialisé avec **valeur**,
arg2 prend successivement la valeur de chaque élément et évalue le bloc avec cette valeur (comme un **do**:),
à l'issue de l'évaluation courante, le résultat de l'évaluation est affectée dans **arg1**.

L'énumérateur **do**:

1. Faire la somme des éléments d'un tableau.
2. La méthode **constantNames** envoyée à la classe **ColorValue** renvoie un tableau **constant** de symboles, chaque symbole ayant le nom d'une couleur.
Afficher, dans le Transcript, les couleurs **constantes** de la classe **ColorValue**

L'énumérateur **collect**:

1. Construire un premier tableau, **et avec la méthode collect**:, construire un deuxième tableau identique au premier.
2. A partir d'un premier tableau, construire un deuxième tableau dont la valeur de chaque élément est le double de l'élément correspondant dans le premier tableau.

Autres énumérateurs

1. A partir du tableau **constant** de symboles ayant le nom d'une couleur, détecter la première couleur commençant par un **d**
(on obtient le premier caractère d'un symbole avec **first**).
2. A partir du tableau **constant** de symboles ayant le nom d'une couleur, construire un tableau des noms de couleur commençant par un **d**
3. A partir du tableau **constant** de symboles ayant le nom d'une couleur, construire un tableau des noms de couleur ne commençant pas par un **d**
Afficher (avec un **do**:) ce tableau dans le Transcript

10.1.4 Les structures alternatives

Smalltalk définit sur la classe `True` et sur la classe `False` quatre méthodes d'instance `ifTrue:`, `ifFalse:`, `ifTrue:ifFalse:`, `ifFalse:ifTrue:`¹.

	Classe <code>True</code>	Classe <code>False</code>
Méthode:	<code>ifTrue: unBloc</code>	<code>ifTrue: unBloc</code>
Action:	renvoyer l'évaluation de <code>unBloc</code>	renvoyer nil
Méthode:	<code>ifFalse: unBloc</code>	<code>ifFalse: unBloc</code>
Action:	renvoyer nil	renvoyer l'évaluation de <code>unBloc</code>
Méthode:	<code>ifTrue: unBloc ifFalse: unAutreBloc</code>	<code>ifTrue: unBloc ifFalse: unAutreBloc</code>
Action:	renvoyer l'évaluation de <code>unBloc</code>	renvoyer l'évaluation de <code>unAutreBloc</code>
Méthode:	<code>ifFalse: unBloc ifTrue: unAutreBloc</code>	<code>ifFalse: unBloc ifTrue: unAutreBloc</code>
Action:	renvoyer l'évaluation de <code>unAutreBloc</code>	renvoyer l'évaluation de <code>unBloc</code>

Par ailleurs, les opérateurs logiques `&` (conjonction, le ET), `Eqv` (équivalence), `not` (négation), `xor` (ou exclusif), `|` (disjonction, le OU) sont définies sur les classes `True` et `False`.

1. Tester si un nombre est impair (en lui envoyant le message `odd`) et sonner la cloche (`Screen default ringBell`) si c'est vrai.
2. Sans utiliser `max:`, écrire un bloc avec deux paramètres qui renvoie le maximum des deux paramètres

10.1.5 Enumérateurs et alternatives

1. Faire la somme des éléments positifs d'un tableau
2. Créer un tableau de 0 ou 1 à partir d'un tableau existant, un nombre du tableau existant est remplacé par un 0 s'il est supérieur ou égal à 10, par un 1 s'il est inférieur à 10.

10.1.6 Itération de blocs

Utilisation des messages `timesRepeat:`, `repeat`, `whileTrue:`

1. Ecrire 10 fois la chaîne 'coucou' dans le Transcript (en passant à la ligne après chaque 'coucou').
2. Ecrire un bloc avec une variable `n` qui écrit `n` fois la chaîne 'coucou' dans le Transcript (en passant à la ligne après chaque 'coucou'). Evaluer ce bloc.
3. Itérer avec un `repeat` un bloc qui incrémente un compteur de 1. On s'arrête quand le compteur est supérieur à 10.
4. Créer un objet de la classe `Time` à 3 secondes du temps courant.
`Time now addTime: (Time fromSeconds: 3)`
Boucler jusqu'à ce que le temps courant dépasse cet objet, en passant à la ligne dans le Transcript

10.2 Exercices (TD)

10.2.1 Les blocs

1. Ecrire un bloc avec une variable, qui renvoie la conversion en degrés Celsius de cette variable (supposée être en Fahrenheit). Evaluer ce bloc ($C = (5/9) (F - 32)$).
2. Ecrire un bloc avec deux variables, qui convertit ces variables en `String`, les concatène en les séparant avec un blanc et renvoie le résultat de la concaténation.

¹Un truc : Si on tape `< Control > t` (ou `< Control > f`) dans une fenêtre de code, le système insère `ifTrue: (ifFalse:)`

10.2.2 Les énumérateurs

L'énumérateur **do**:

1. Compter le nombre d'éléments d'un tableau. Vérifier avec **size**
2. Faire la moyenne des éléments d'un tableau.

L'énumérateur **collect**:

1. Une chaîne instance de la classe **cString** est un tableau, donc on peut avoir son premier élément (**first**), son *i*-ème élément (**at: i**), affecter une **Valeur** dans son *i*-ème élément **at: i put: uneValeur**, etc
A partir du tableau **constant** de symboles ayant le nom d'une couleur, construire un tableau de **String** où le nom de la couleur commence par une majuscule (**asUppercase**).
2. Construire un tableau dont la valeur des éléments est la somme de l'élément précédent avec l'élément courant.

10.2.3 Les structures alternatives

1. Sans utiliser **max:**, écrire un bloc avec trois paramètres qui renvoie le maximum des trois paramètres
2. Ecrire un bloc avec un paramètre, qui teste si le paramètre est une minuscule (**isLowercase**), une majuscule (**isUppercase**), un chiffre (**isDigit**) ou autre.
Le bloc renvoie 'minuscule' ou 'majuscule' ou 'chiffre' ou 'autre'.

10.2.4 Les intervalles

Méthodes d'intervalles

1. Afficher les 10 premiers carrés.
2. Afficher les multiples de 10.

Créer et utiliser des intervalles

Envoyé à un nombre, la méthode **to: borneSuperieure** renvoie un **Interval** allant du nombre à la **borneSuperieure** par pas de 1.

Envoyé à un nombre, la méthode **to: borneSuperieure by: lePas** renvoie un **Interval** allant du nombre à la **borneSuperieure** par pas de **lePas**.

Les **Interval** étant des collections, on peut utiliser les énumérateurs **do:**, **collect:**, etc.

Exemple : **tab := (1 to: 100)** crée un tableau des 100 premiers entiers.

1. Calculer la somme des 100 premiers nombres entiers à l'aide d'un tableau. Vérifier avec la formule $n*(n+1)/2$
2. Créer un tableau avec les nombres de 0 à 360 de 30 en 30. utiliser ce tableau pour afficher la table des sinus de 30 degrés en 30 degrés.

10.2.5 Enumérateurs et alternatives

1. A partir d'un premier tableau, construire un tableau en inversant tous les éléments qui sont des fractions.

10.2.6 Autres énumérateurs

1. A partir d'un premier tableau, construire un deuxième tableau en supprimant les éléments négatifs.
2. Définir la somme des éléments d'un tableau avec `inject:into:`.
3. Définir la somme des éléments positifs d'un tableau avec `select:` puis `inject:into:`.
4. Définir la somme des éléments négatifs d'un tableau avec `reject:` puis `inject:into:`.

10.2.7 Itération de blocs

1. Afficher 10 * dans le Transcript, puis passe à la ligne.
2. Afficher 10 lignes de 10 * dans le Transcript (en passant à la ligne après chaque ligne de 10 *).
3. Ecrire un bloc à deux paramètres `ctl` et `c`, qui affiche `l` lignes de `c` colonnes de * dans le Transcript (en passant à la ligne après chaque ligne).

Les collections

Main Author(s): to be fixed: B. Pottier, Université de Brest, Bernard.Pottier@univ-brest.fr, catherine dezan

11.1 Introduction :organisation hiérarchique des collections

La figure suivante présente une partie de l'organisation hiérarchique des collections.

```

Object ()
  Collection ()
    Bag ('contents')
    SequenceableCollection ()
      ArrayedCollection ()
        Array ()
        CharacterArray ()
        String ()
        Symbol ()
        Text ('string' 'runs')
      IntegerArray ()
      ByteArray ()
      WordArray ()
      Interval ('start' 'stop' 'step')
      OrderedCollection ('firstIndex' 'lastIndex')
      SortedCollection ('sortBlock')
    Set ('tally')
    Dictionary ()
  
```

11.2 TP

11.2.1 La classe **Collection**

On trouve dans cette classe, les messages (les énumérateurs) compréhensibles par toutes les sous-classes.

On retiendra les messages suivants :

- les énumérateurs correspondant aux messages : collect:, do:, detect:, reject:, select, inject: into:.
- les messages de conversion : asArray, asBag, asSet, asSortedCollection.
- le message size donne la taille de la collection

11.2.2 La classe **SequenceableCollection**

Un ordre est défini entre les éléments de la collection. Les objets de cette classe comprennent les messages first, last.

- La classe **Array** est une collection d'éléments de taille fixe. La création d'un objet de cette classe peut se faire grâce au message **new:**.

Ex : **Array new: 12** crée un tableau de taille 12 dont les éléments sont initialisés à nil.

Les accès aux éléments du tableau sont possibles avec la méthode **at:**. La modification d'un élément du tableau est faite par la méthode **at: put:**.

- La classe **Interval** ne possède pas d'ordre explicite mais implicite. La création d'un intervalle se fait avec les messages **to:** ou **to: by:** selon le pas de l'intervalle choisi.

Ex: **1 to: 10** crée un intervalle avec tous les éléments de 1 à 10.

1 to: 10 by: 2 crée un intervalle de 1 à 10 avec uniquement les éléments impairs.

- La classe **OrderedCollection** définit des collections de taille dynamique. La création est faite par le message **new**, puis l'extension par l'utilisation du message **add:** ou **addAll:**.

Ex: **OrderedCollection new add:1; add:2** définit une collection avec les éléments 1 et 2.

Le message **addAll:** permet de définir une collection à partir de celle passée en paramètre.

Ex: **OrderedCollection new addAll: #(1 4) 5 6)** crée une instance de la classe **OrderedCollection** contenant les éléments **#(1 4)**, 5 et 6. La suppression d'un élément dans un tableau se fait par la méthode **remove:**.

- La classe **SortedCollection** est une sous-classe de la classe **OrderedCollection**. Les éléments de cette classe peuvent être triés selon un certain critère défini dans un bloc en utilisant la méthode **sortBlock:**.

Ex :

```
| aCol |
aCol:= SortedCollection new.
aCol add:2; add:8; add:1.
    "la collection obtenue est triée dans l'ordre croissant"
aCol sortBlock: [ :elem1 :elem2 | elem1 > elem2].
    "on trie la collection dans l'ordre décroissant"
```

Exercices

Exercice: 37. Créer un tableau de taille 5, trier ses éléments dans l'ordre décroissant et donner le tableau résultat.

Exercice: 38. soit **matrice := #(1 0 0) #(1 2 0) #(1 2 3)**, faire la somme des éléments se trouvant sur la diagonale de cette matrice.

Exercice: 39. Définir par un bloc avec un paramètre (représentant la taille des tableaux) les tableaux de la forme suivantes :

```
##(1)           pour taille=1
##(1) #(1 2))   pour taille=2
##(1) #(1 2) #(1 2 3)) pour taille=3
##(1) #(1 2) #(1 2 3) #(1 2 3 4)) pour taille=4
....
```

11.2.3 La classe Set

Un ensemble est une collection dont les éléments n'ont pas d'ordre, pas de clés d'accès. Les doublons n'y sont donc pas autorisés.

Les méthodes suivantes sont disponibles sur un ensemble :

- **add: unElement** méthode d'ajout de l'élément **unElement** (si il n'appartient pas déjà à l'ensemble)

- **remove:** `unElement ifAbsent:` `unBloc` supprime l'élément `unElement` s'il est présent et évalue le bloc `unBloc` si l'élément est absent.
- **includes:** `unElement` teste si l'élément `unElement` appartient à l'ensemble (renvoie `true` si il s'y trouve, `false` sinon).
- **occurrencesOf:** `unElement` donne le nombre d'occurrences de l'élément `unElement` (0 ou 1)

Exemples.

```
| aSet |
"Construction d'un ensemble vide"
aSet := Set new.          "Set ()"
"Ajouts"
aSet add: 1; add: 1@3.
aSet.                    "Set (1 1@3)"

"Suppression"
aSet remove:3 ifAbsent:[aSet add: 4/5].
aSet.                    "Set (1@3 (4/5) 1)"

"Appartenance"
aSet includes:1@3.        "true"

"Transformation d'un Array en Set"
#(2 7 t 7 9) asSet        "Set (2 #t 7 9)"
#(3 6 6) asSet occurrencesOf: 6    "1"
```

Exercices

- Exercice: 40. A partir d'un premier tableau, définir un nouveau tableau où les doublons du tableau initial sont supprimés. On peut utiliser la méthode `asArray` pour transformer un receveur en tableau.
- Exercice: 41. Construire un ensemble contenant tous les nombres de 1 à 100.
- Exercice: 42. Transformer l'ensemble précédent en remplaçant chaque nombre par son reste de la division entière par 5 (modulo 5). Quelle est la nature et la taille de cette nouvelle collection? Refaire la même opération directement sur un intervalle de (1..100).

11.2.4 La classe Dictionary

Les dictionnaires sont des ensembles dont les éléments sont des instances de la classe *Association*. Une association est un couple d'objets, le premier élément étant une clé, le deuxième représentant une valeur.

Exemple : `2->3` est une association dont la clef est 2 et la valeur 3.

Quelques méthodes disponibles sur les dictionnaires :

- `at:k` renvoie la valeur référencée par la clé `k`
- `at:k put:o` installe l'objet `o` comme valeur à la clé `k`
- `add:uneAssociation` ajoute au dictionnaire, l'argument *uneAssociation* (qui est une association)
- `associations` renvoie une collection ordonnée des associations du dictionnaire
- `keys` renvoie un set contenant toutes les clés

- `values` renvoie une collection ordonnée comportant toutes les valeurs
- `includesKey:key` teste si la clé `key` appartient au dictionnaire
- `associationAt:key` renvoie l'association dont la clef est `key`
- `keyAtValue:uneValeur` renvoie une clef associée à la valeur `uneValeur` passée en paramètre
- `keysAndValuesDo:unBloc` exécute le bloc `unBloc` pour chaque association clé-valeur du dictionnaire.

Exercices

- Exercise: 43. Créer un ensemble avec les éléments appartenant à l'intervalle (1 .. 100) et dont le reste de la division entière par 5 est égal à 0. Construire l'association avec 0 comme clé et cet ensemble comme valeur.
- Exercise: 44. Créer un dictionnaire où les clés sont les restes de la division par 5 des éléments appartenant à l'intervalle (1 .. 100) et les valeurs sont les sous-ensembles constitués d'éléments de l'intervalle qui ont le même reste pour la division par 5 (reste représenté par la clé).
- Exercise: 45. Transformer le dictionnaire précédent en un nouveau dictionnaire où les sous-ensembles associés aux différentes clés ont été remplacés par la somme de leurs éléments.

11.2.5 La classe Bag

Une instance de la classe **Bag** peut être considéré comme un **Set** qui autorise les doublons. Un **Bag** ne duplique pas physiquement les doublons, mais maintient le nombre d'occurrences de chaque élément. Un **Bag** a un contenu qui est un dictionnaire associant chaque élément à son nombre d'occurrences.

Bag hérite des méthodes classiques de la classe **Collection**.

Quelques méthodes spécifiques de **Bag** :

- `occurrencesOf:unElement` donne le nombre d'occurrences de l'élément `unElement`
- `add:unElement withOccurrences:nbOccurrences` ajoute le nombre d'occurrences `nbOccurrences` de l'objet `unElement`
- `remove:unElement ifAbsent:unBloc` supprime une occurrence l'élément `unElement` s'il est présent et évalue le bloc `unBloc` si l'élément est absent
- `removeAllOccurrencesOf:unElement ifAbsent:unBloc` supprime toutes les occurrences de l'élément `unElement` s'il est présent et évalue le bloc `unBloc` si l'élément est absent
- `valuesAndCountsDo:unBloc` permet d'itérer le bloc `unBloc` sur des couples (objet nb_occ_de_objet) en supposant que le bloc `unBloc` a deux arguments (analogie avec la méthode `keysAndValuesDo:` des dictionnaires).

Exercices

- Exercise: 46. Récupérer dans un **Bag** les restes de la division par 2 des nombres compris entre 1 et 100 000. Inspecter l'objet ainsi obtenu.
- Exercise: 47. Faire la somme des éléments se trouvant dans le **Bag**. Faire la même opération sur le tableau correspondant à ce bag. Quel est le code qui offre le temps d'exécution le plus intéressant? Pour connaître le temps d'exécution d'un code, utilisez l'expression `Time millisecondsToRun:` suivi d'un paramètre étant un bloc sans variable comportant le code à évaluer. Par exemple `Time millisecondsToRun: [(1 to: 1000) asBag]`.
- Exercise: 48. Généraliser le test précédent sur une collection de tableaux ayant des valeurs constantes et sur une collection de bag comportant les mêmes éléments afin de mieux comparer les temps d'exécution pour l'opération de la somme.

11.3 Exercices (TD)

11.3.1 La classe **SequenceableCollection**

Exercice: 49. Créer en utilisant un bloc avec un paramètre (le paramètre caractérisant la taille des matrices générées) les matrices carrés suivantes :

<code>##(1))</code>	pour valeur=1
<code>##(1 2) #(1 2))</code>	pour valeur=2
<code>##(1 2 3) #(1 2 3) #(1 2 3))</code>	pour valeur=3

Exercice: 50. Définir à partir d'une matrice (tableau de lignes), un nouveau tableau où les éléments sont les sommes des lignes du tableau initial.

Exercice: 51. Soit le tableau `##('toto' 'lulu' 'guillaume' 'luc' 'laurent')`, définir un nouveau tableau où les noms sont triés par ordre décroissant de taille.

11.3.2 La classe **Set**

Exercice: 52. Convertir le tableau `##(1 2 3 4 2 4)` en set. Quelle est la taille du résultat?

Exercice: 53. Comment afficher tous les éléments d'un set dans le Transcript?

Exercice: 54. Créer un bloc à deux paramètres — deux sets — qui renvoie un set correspondant à l'intersection des deux sets

Exercice: 55. Créer un bloc à deux paramètres — un set et un objet quelconque — Si l'objet appartient à l'ensemble il est supprimé de celui-ci. Si il n'y appartient pas, le message "l'objet monObjet n'appartient pas a l'ensemble" est affiché sur le Transcript.

11.3.3 La classe **Bag**

Exercice: 56. Créer un bag et y ajouter tous les éléments du tableau `##(1 1 1 2 3 4)`

Exercice: 57. Convertir le tableau `##(1 1 1 2 3 4)` en Bag

Exercice: 58. Ecrire un bloc avec une variable qui prend un bag et qui renvoie un bag. Les valeurs du second bag correspondent au double des valeurs du premier bag

Exercice: 59. Ecrire un bloc avec une variable qui prend un bag et qui renvoie un dictionnaire. Le dictionnaire contient les associations élément du bag -> nombre d'occurrences de l'élément.

Exercice: 60. Modifier le bloc précédent pour qu'il renvoie la somme de tous les éléments du bag paramètre

11.3.4 La classe **Dictionary**

Exercice: 61. Créer un nouveau dictionnaire. Quelle est sa taille? Pourquoi?

Exercice: 62. Y ajouter l'objet 15 à la clé 1

Exercice: 63. Y ajouter l'association 2 ->5

Exercice: 64. Que se passe-t-il si on ajoute l'association 2 ->6?

Exercice: 65. Inspecter les clés du dictionnaire

Exercise: 66. On décrit un parcours sous la forme d'une liste de trajets. Créer un dictionnaire dont les clés sont les points de départ et les valeurs les points d'arrivée à partir du tableau suivant:

Brest	Quimper
Quimper	Lorient
Nantes	Vannes
Rennes	Nantes

Exercise: 67. Créer un block qui prend en paramètre le dictionnaire et qui écrit sur le Transcript l'ensemble des trajets. exemple: 'De Brest je peux aller à Quimper'

Main Author(s): Brest

But du TD: Structuration d'expression Smalltalk: objet, message. Les littéraux du langage. Messages unaires, binaires et à mots clés. Les règles de priorités.

12.1 Rappels

1. Répondez aux questions suivantes :
 - Lister les différents littéraux
 - Lister trois objets qui ne sont **pas** des littéraux
 - Lister les catégories de messages. Pour chacun d'entre elles donner le nombre de paramètres, et les signes distinctifs.
 - Lister les règles de priorité entre messages.
2. Si on vous donne le code suivant, trouvez les variables, les littéraux, les messages, les receveurs et les paramètres des messages, donner l'ordre d'évaluation des messages.

```
| toto titi z |
toto := 34 glop glop. titi := 23 +- toto.
toto pas: #te glop: #ti pas: 'tte' te' glop: $r r.
z := toto = titi
schmilblick.
((titi z z: z z: #(z)) z: 12) z: 'titi'
```

On rappelle que l'ensemble des messages binaires est connu par le système. Un message binaire est composé de un ou deux caractères parmi les suivants: ! % & * + , - / < = > ? @ \ ~

12.2 L'objet: c'est du gâteau!

Dans cet exercice on souhaite décrire une opération courante (une recette de cuisine) en utilisant le formalisme objet message receveur de SmallTalk. Soit la recette suivante (crumble simplifié sans garniture) : *“Dans un plat disposez un poids égal de farine, beurre et sucre (150 g), mélangez, puis faite cuire à thermostat 6 pendant 40 minutes. Le gâteau est prêt”*.

On donne les indications suivantes :

- Classe PlatAGateau :
 - message pour créer une instance de PlatAGateau : new
 - message pour ajouter un ingrédient dans une instance de PlatAGateau : ajouter: avec en argument l'ingrédient à ajouter
 - message pour mélanger le contenu d'une instance de PlatAGateau : mixer

- message pour cuire une instance de `PlatAGateau` : `cuireThermostat: duree:` avec en premier argument la valeur du thermostat et en deuxième argument la durée en minutes
 - message pour démouler le contenu d’une instance de `PlatAGateau` : `démouler` qui renvoi en retour une instance de la classe `Gateau`.
- Classe `Beurre` :
 - message pour créer une instance de `Beurre` : `new:` avec en argument le poids en grammes
 - Classes `Farine` et `Sucre` :
 - Mme message de création que la pour la classe `Beurre`
1. En utilisant les Classes et les méthodes fournies, écrivez en Smalltalk la recette. Vous ferez en sorte de ne pas mettre plus d’un message par ligne. Vous pouvez utiliser des variables locales avec des affectations.
 2. Ecrivez l’équivalent de ce code sur une seule ligne de manière à avoir directement un gateau. Prenez soin de correctement positionner des parenthèses si nécessaire.

12.3 Priorité de messages et variable temporaire

12.3.1 Fonctions trigonométriques

Un nombre (en radians) comprend les messages correspondant aux fonctions trigonométriques `sin` `cos` `tan` `arcSin` `arcCos` `arcTan`.

On convertit un nombre de Degré à Radian en lui envoyant le message `degreesToRadians`.

Calculer (à l’aide d’une fonction trigonométrique) le côté d’un carré dont la diagonale mesure 1.41421 mètres.

12.3.2 Maximum

La méthode `max:` `unAutreNombre` appliqué à un nombre renvoie le plus grand des deux nombres.

Exemple : `2 max: 6` renvoie 6.

1. Calculer le maximum de la somme des couples formés à partir de trois variables `a` `b` `c` contenant des valeurs quelconques, en stockant dans des variables les résultats intermédiaires.
2. Calculer le maximum de la somme des couples formés à partir de trois variables `a` `b` `c` contenant des valeurs quelconques **sans utiliser de variables intermédiaires**

12.3.3 Conversion

Le message binaire `//` correspond à l’opération de division entière. Le message binaire `%` correspond à l’opération de modulo (reste de la division entière).

On utilise ces messages pour convertir en binaire un nombre hexadécimal (de 0 à 15 en base 10, de 0000 à 1111 en base 2).

En effectuant une série de divisions entières, les chiffres binaires sont obtenus **de la droite vers la gauche** grâce au reste de la division entière (le modulo).

Appliquer cet algorithme pour convertir le nombre hexadécimal $(F)_{16}$ (valeur décimale 15) en base 2.

12.4 Tableau et point

Les messages à mots clés `at: unIndex` et `at: unIndex put:uneValeur` permettent l'accès en lecture ou écriture d'un élément se trouvant dans un tableau. Les messages `x, y, x: uneValeur` et `y: uneValeur` permettent l'accès en lecture ou écriture de l'abscisse ou de l'ordonnée d'un point.

12.4.1 Analyse de code

Repérer dans les lignes de code suivantes, les messages unaires, binaires et à mots clés. En déduire l'évaluation des expressions et des portions de code.

```
| aPoint |  
aPoint:= Point x:2 y:1.  
aPoint x: aPoint x * 2.  
aPoint
```

```
| x tab |  
tab := #(4 5.0 'toto' 1111).  
x := tab at:1.  
tab at: 1 put: (Fraction numerator: x*2 denominator: 7 + x negated).  
tab
```

12.4.2 Exercices sur les tableaux et les points

1. Multiplier par 2 le 2ème élément d'un tableau,
2. Remplacer la valeur du 2ème élément d'un tableau par son opposé.
3. Remplacer la valeur du 3ème élément par la valeur du 2ème élément.
4. Remplacer la valeur du 3ème élément par la somme des 2ème et 3ème (ancienne valeur) éléments.

12.4.3 Pour aller plus loin

1. Le 2ème du tableau étant une fraction, remplacer cette fraction par la fraction inverse dans le tableau.
2. On désire définir le barycentre de points. La notion de barycentre s'applique à un ensemble de points affectés par une masse. Soit p_1, p_2, \dots, p_n un ensemble de points affectés des masses m_1, m_2, \dots, m_n , on appelle barycentre le point p défini de la manière suivante:

$$m.p = \sum_{i=1}^n m_i.p_i \text{ avec } m = \sum_{i=1}^n m_i$$

Définir des expressions smalltalk permettant de définir le barycentre p , on prendra $n = 4$ et les valeurs des masses et des points seront donnés dans des tableaux que l'on définira.

Part IV

Environment

Some Useful Tools in Squeak

Main Author(s): Bergel, Denker, Ducasse

13.1 SqueakMap Package Loader

Before starting the exercises provided in this booklet, you need to install some useful tools. These are installable packages offered from the SqueakMap package loader. If you are behind a proxy, you need to set it: in a workspace, evaluate `HTTPSocket useProxyServerNamed: 'proxy.unibe.ch' port: 80`. To open a SqueakMap package loader, click on the background, this will bring the so-called World Menu, select open... SqueakMap Package Loader. You obtain a list of all the packages available in Squeak. We suggest you to load the packages:

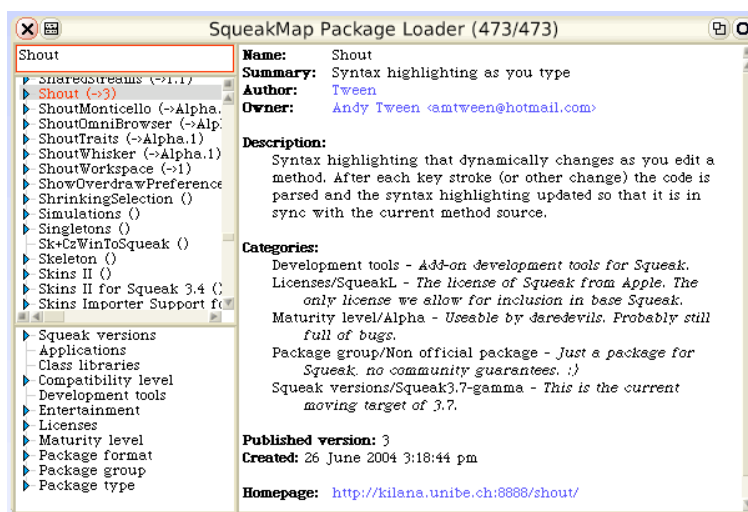


Figure 13.1: SqueakMap Package Loader on Shout

1. Monticello: Monticello is a package support for Squeak (normally already included in 3.7 full release).
2. Shout (syntax highlighter while typing),
3. KomHttpServer (web server): answer yes to the first two questions, and then **always** no,
4. Seaside (the dynamic web application framework): it asks you for a login and password,
5. Refactoring Browser for Squeak 3.7.

13.2 Monticello

Monticello is a CVS-like tool for Squeak. You can find the documentation at: <http://www.wiresong.ca/Monticello/UserManual/>. Open Monticello using open... Monticello. Monticello allows you to save projects in various kind of servers: http, ftp, file system, data bases, You can save your project on SqueakSource, if you want (<http://www.squeaksource.com>).

By convention, the name of a package should be the same as a class-category. As in Smalltalk this is possible to extend classes, you can associate a class extension with a package by putting a * followed by the name of the package in the method category. For example in Figure 13.3, the method named `stylerAboutToStyle:` is defined in the `*Shout-Styling` category, therefore it belongs to the package `Shout-Styling`.

You can browse the contents of a package by clicking on the browse button and in particular you can see the extensions associated to a package. See the Monticello chapter.

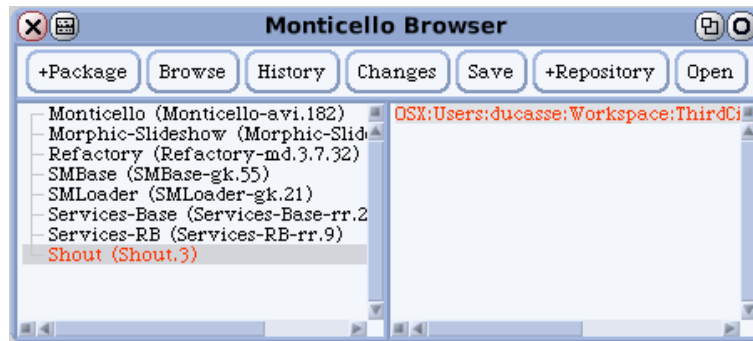


Figure 13.2: Monticello

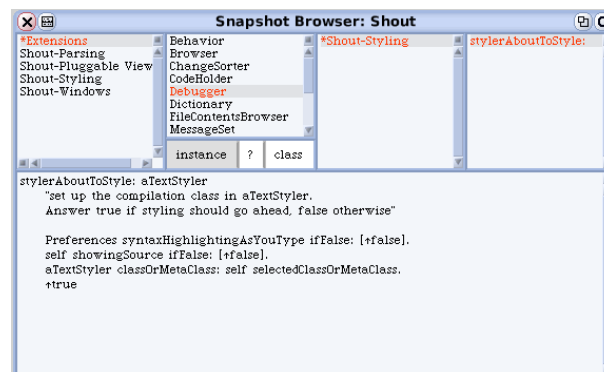


Figure 13.3: Browsing the changes associated to a package.

13.3 SqueakSource: the Squeak SourceForge

SqueakSource (www.squeaksource.com) is a free source forge like open-source code repository. You can manage your squeak source there. For that you should define a project there and add it into your Monticello list of repositories.

You can define a new repository in Monticello and publish automatically to this repository. For that you should paste the project information specified in SqueakSource into the repository dialog as shown in Figure 13.5

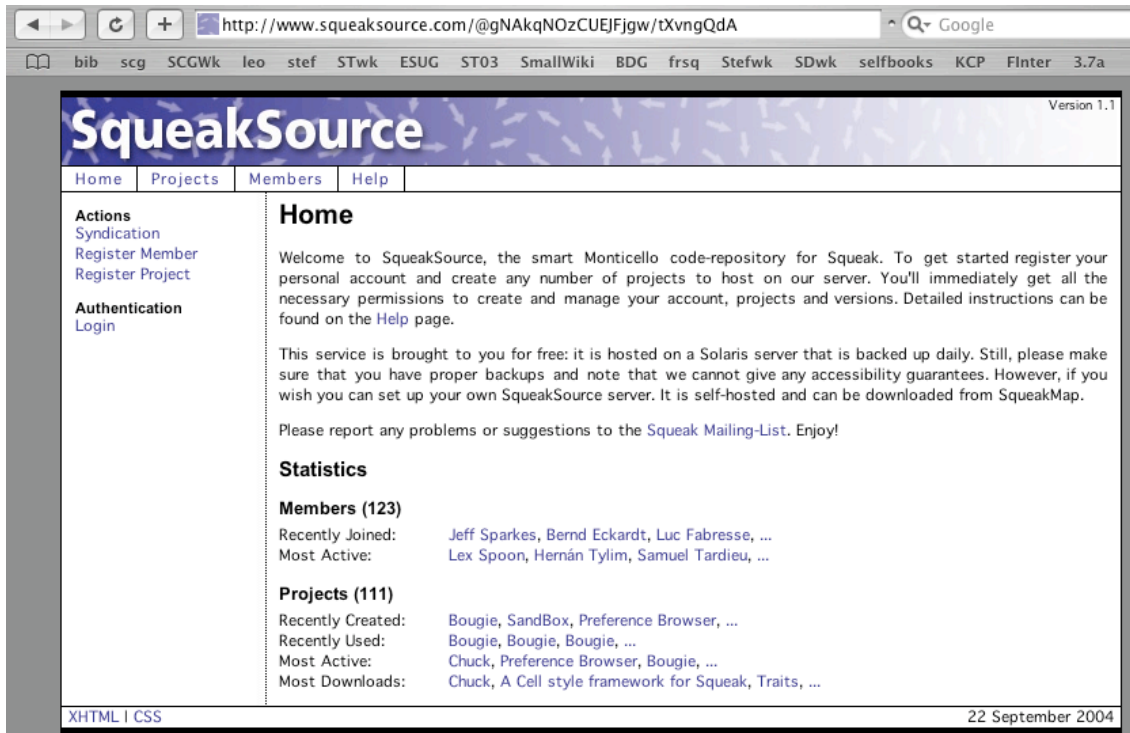


Figure 13.4: SqueakSource is a source forge like server for Squeak.



Figure 13.5: Adding a repository to your monticello repository list.

Monticello

14.1 Packages in Monticello: PackageInfo

The PackageInfo system is a simple, lightweight way of organizing Smalltalk source: it is nothing more than a naming convention, which uses (or abuses) the existing categorization mechanisms to group related code. Let me give you an example: say that you are developing a framework named SqueakLink to facilitate using relational databases from Squeak. You will probably have a series of system categories to contain all of your classes (e.g., category SqueakLink-Connections containing the classes OracleConnection, MySQLConnection and PostgresConnection) (SqueakLink-Model containing DBTable, DBRow and DBQuery) and so on. But not all of your code will reside in these classes - you may also have, for example, a series of methods to convert objects into an SQL friendly format: Object>>asSQL, String>>asSQL and Date>>asSQL.

These methods belong in the same package as the classes in SqueakLink-Connections and SqueakLink-Model. You mark this by placing those methods in a method category (of Object, String, Date, and so on) named *squeaklink (note the initial star). The combination of the SqueakLink-... system categories and the *squeaklink method categories forms a package named "SqueakLink".

The rules, to be precise, are this: a package named "Foo" contains

- All class definitions of classes in the system category Foo, or in system categories with names starting with "Foo-".
- All method definitions in any class in method categories named *Foo or with names starting with *foo-.
- All methods in classes in the system category Foo, or in system categories with names starting with Foo-, except those in method categories with names starting with * (which must belong to other modules).

14.2 Getting Started

Installing The best way to install Monticello is via SqueakMap. Note however, that MC has two dependencies, both are part of the standard image, so it's usually not necessary to install them explicitly. However, the update stream tends to lag behind the versions on SqueakMap, so it's often a good idea to upgrade them before installing MC.!

- PackageInfo groups classes and methods into packages using a simple naming convention. It became part of the standard image in update 5250.
- MCInstaller provides a way to load Monticello Versions into an image that doesn't have Monticello installed. Since Monticello is self hosting, it's used for bootstrapping. It's present in images updated through 5710 and later.

Creating a Working Copy Once Monticello is installed, the Monticello Browser will be available from the 'open...' menu. Open it by selecting World / open... / Monticello Browser.

The first thing you need to do is tell Monticello about the package you are interested in versioning. You do this by creating a Working Copy.

From an .mcz version file Open a FileList and navigate to the version file. Click on the 'Load' button to load the package into your image.

From a version in a repository First connect to the repository, either local or remote, that contains the version you want to load. See below for details. Then open the repository: select the repository in the list on the right-hand side of the Monticello Browser, and click the 'Open' button. This will open a Repository Inspector. Select your version and click the 'Load' button.

From scratch Click on the '+Package' button, and enter the name of a PackageInfo package. It doesn't matter whether or not the code for the package already exists.

Once the Working Copy has been created, the name of the package will appear in the package list on the left side of the Monticello Browser. If you loaded an existing version, the version name will be displayed in parenthesis after the package name, otherwise the parenthesis will be empty, indicating that your working copy has no ancestors.

Connecting to a Repository If you've already got a Working Copy, click on the package name on the left side of the Monticello Browser, so that your repository will be associated with your package. To connect to a repository, click on the '+Repository' button in the Monticello Browser. A pop-up menu will appear, allowing you to select the type of repository you want to connect to.

The simplest repository type is 'directory.' When you select this type of repository, Monticello will open a FileList2 to allow you to select an existing directory in which to store versions. Other types of repositories typically require more configuration, and will open a text pane to allow you to enter it.

Saving Changes Changes to your working copy are automatically logged in your changes file, so you only need to create a new version of your package when you want to share the changes with others. Select the package on the left side of the Monticello Browser and the repository to save to on the right, then click the 'Save' button. See Repositories for discussion of how to publish to shared repositories.

Merging Changes If you or some other developer have made changes to the same version of a package, load one version as your working set and then select the repository containing the other version in the Monticello Browser, open a Repository Browser and select the other version. Clicking the 'Merge' button will automatically load all non-conflicting changes from the other version. If you need to control which changes to accept, you may instead click 'Changes' to browse every difference.

14.3 Elements of Monticello

Packages Packages are the units of versioning used by Monticello; the classes and methods they contain are recorded and versioned together. Monticello uses the packages defined by PackageInfo.

Snapshots A Snapshot is the state of a Package at a particular point in time

Versions A Version is a Snapshot of a Package and it's associated metadata - author initials, the date and time the snapshot was taken, and the Version's ancestry - the list of Versions from which it is derived.

A Version is the standard currency of the system. You save them, load them, give them to others, merge them, delete... you get the picture. Versions are often stored in mcz files - see File Format

Working Copies Each package in an image that is being versioned with Monticello has a Working Copy. The Working Copy represents the Version of the package that is currently active in the image, and which may be modified by the Smalltalk development tools.

Repositories These are places to store your Versions. Unlike CVS, in which a Package is associated with one Repository, a Monticello Package can have Versions in many repositories. When adding a new Repository to use, you can choose from SqueakMap Cache, FTP, HTTP (webdav), SqueakMap Release, SMTP, or a directory somewhere on your hard drive (or network drive).

For example, if I have six versions of package Foo, I could have Foo versions 1-4 being on my local harddrive, and 5-6 being on an ftp server. You could download version 5, make some changes and commit a new version (7) to your WebDAV repository. I can download and merge that version with my own work to produce version 8, which I save to my ftp repository.

This is a key element of Monticello's distributed development model.

Package cache The package-cache is a local repository the Monticello uses to cache any package that is loaded into a particular image in a directory. That means it is filled with .mcz files, whether it is a package you create in your image, or one you download from somewhere else.

When you use images in different directories you will have multiple package-caches, and may hold many of the same packages. If MC is loaded into an image which is subsequently moved, MC will continue to use the package-cache in the directory the image was moved from. Otherwise MC creates a new package-cache in the local directory. This can become a real mess and so some have used symlinks on unix systems to centralize it.

Why cache packages at all? When a Version is loaded into the image, it is likely to become the ancestor of new versions that are created as part of the development process. During merges, Monticello needs to examine the Snapshots of these ancestors in order to detect conflicts. By caching these ancestors as it loads them, MC reduces the chance that the necessary version will be unavailable - either because the repository it's in is no longer available or because it was loaded directly from a file and isn't in any repository.

14.4 Repositories

There are currently 8 types of repositories, each with different characteristics and uses. Repositories can be read-only, write-only or read-write.

HTTP HTTP Repositories are often general purpose read-write repositories for day-to-day development using a shared server. (Although the server can be configured for read-only access. Saving Versions via HTTP uses the PUT method, which must be enabled on the server.)

The nice thing about HTTP repositories is that it's easy to link directly to specific versions from web sites or SqueakMap. With a little configuration work on the HTTP server, HTTP repositories can be made browseable by ordinary web browsers, WebDAV clients, etc.

FTP Similar to an HTTP repository, except that it uses an FTP server instead.

GOODS This repository type stores Versions in a GOODS object database. It's a read-write repository, so it makes a good "working" repository where Versions can be saved and retrieved. Because of the transaction support, journaling and replication capabilities of GOODS, it is suitable for large repositories used by many clients.

directory A directory repository stores Versions in a directory in the local filesystem. Since it requires very little work to set up, it's handy for private projects or disconnected development. The Versions in a directory repository can be uploaded to a public or shared repository at a later time.

SMTP SMTP repositories are useful for sending Versions by mail. When creating an SMTP repository, you specify an destination email address. This could be the address of another developer - the package's maintainer, for example - or a mailing list such as squeak-dev. Any Versions save to the repository will be emailed to this address.

SqueakMap Release This is a write-only repository used for publishing releases of a package to SqueakMap. To configure the repository enter the name of the package on SqueakMap, your SM initials and your SM password. Now any Versions saved to the repository will be uploaded to your SM account, and registered as a new release with SqueakMap.

SqueakMap Cache When packages are installed through SqueakMap, the downloaded files are stored in a cache. In order to make these files, which are often Versions in .mcz format, available to Monticello for loading, merges etc, a SqueakMap Cache repository is created when these files are loaded for the first time.

package-cache The package cache is a special repository that Monticello creates automatically. Like a directory repository, the package cache stores files in a directory on your local filesystem. See Elements of Monticello for more information.

14.5 File Format

Versions are often saved in binary files for storage in repositories, distribution to users etc. These files are commonly call 'mcz files' as they carry the extension .mcz.

Archive contents Mcz files are actually ZIP archives that follow certain conventions. Conceptually a Version contains four things:

- **Package.** A Version is related to a particular Package. Each mcz file contains a member called 'package' which contains information about the Version's Package.
- **VersionInfo.** This is the meta-data about the Snapshot. It contains the author initials, date and time the Snapshot was taken, and the ancestry of the Snapshot. Each mcz file contains a member called 'version' which contains this information.
- **Snapshot.** A Snapshot is a record of the state of the package at a particular time. Each mcz file contains a directory named 'snapshot/'. All the members in this directory contain definitions of program elements, which when combined form the Snapshot. Current versions of Monticello only create one member in this directory, called 'source.st'.
- **Dependencies.** A Version may depend on specific Versions of other packages. An mcz file may contain a 'dependencies/' directory with a member for each dependency. These members will be named after the Package depended upon.

Source code encoding The member named 'snapshot/source.st' contains a standard fileout of the code that belongs to the package.

Metadata encoding The other members of the zip archive are encoded using S-expressions. Conceptually, the expressions represent nestable dictionaries. Each pair of elements in a list represent a key and value. The following example needs little explanation:

```
(key1 'value1' key2 (sub1 'sub value 1'))
```

Distributing mcz files The metadata for a Version ends up being fairly compact, so it's not unreasonable to distribute with a release. It's also important that it be present if somebody decides to start hacking on your Package. Then they can create a mcz with their Version of your package and it will have the correct ancestry information, enabling you to easily and correctly merge it back into your work.

Stated another way, a Version doesn't contain a full history of the source code. It's a snapshot of the code at a single point in time, with a UUID identifying that snapshot, and a record of the UUIDs of all the previous snapshots it's descended from. So it's a great thing to distribute.

14.6 The Monticello Browser

The Monticello Browser is the central window of the interface. All versioning operations begin with the Monticello Browser.

The browser contains two panes. The left pane contains the list of packages that have Working Copies in the image. In parenthesis, the immediate ancestors of the Working Copies are also listed. Packages that have been modified since they were loaded are displayed with an asterisk before their names. The list on the right shows the repositories that are configured for the selected package. The buttons across the top are enabled and disabled depending on the selections in the two panes; many commands require you to first select a package and repository.

+Package The '+Package' button is used to create a Working Copy for a package. When you click on it, Monticello will ask for the name of the Package you want to version, the same name that PackageInfo uses to identify the package. Once the Working Copy has been created, the name of the package will appear in the left pane.

The '+Package' button should only be used to create a Working Copy for a brand-new package, one that has not previously versioned with Monticello. To create a Working Copy from an existing Version, you should load the version from a repository or directly from an .mcz file using the FileList. See Getting Started for details.

Browse The 'Browse' button takes a Snapshot of the current state of the selected package and opens a Snapshot Browser on it.

History The 'History' button opens a History Browser on the Working Copy for the selected package.

Changes The 'Changes' button is used to display the changes made to the selected package since it was last saved or loaded. Monticello first takes a Snapshot of the package and compares it to the package's first immediate ancestor. If any changes have been made, a Patch Browser is opened to display them.

Save The 'Save' button is for saving new Versions of the selected package. It opens a dialog that allows you to enter the name of the new version and a log message describing the changes made since the last version. If you click 'accept,' Monticello will take a Snapshot of the package and save it as a Version to the selected repository.

+Repository The '+Repository' button is used to connect to a Repository. It opens a menu allowing you to choose the type of repository you wish to connect to, and depending on the repository type, a configuration dialog for the connection.

Open The 'Open' button opens a Repository Inspector on the selected repository. This is useful when you need to find a specific Version to load, merge, browse etc.

14.7 The Snapshot Browser

The Snapshot browser is much like the standard Smalltalk System Browser except that it displays the contents of a Snapshot, rather than the code that is active in the image. Since Snapshots are immutable, the Snapshot browser does not allow editing.

One difference between the Snapshot Browser and the familiar system browsers is that the Snapshot browser uses the special system category '*Extensions' to categorize classes that do not belong to the package, but which have extension methods that do.

14.8 More on PackageInfo

To get a feel for this, try filing the Refactoring Browser. The Refactoring Browser code uses PackageInfo's naming conventions, using "Refactory" as the package name. In a workspace, create a model of this package with `refactory := PackageInfo` named: 'Refactory'.

It is now possible to introspect on this package; for example, `refactory` classes will return the long list of classes that make up the Refactoring Browser. `refactory coreMethods` will return a list of `MethodReferences` for all of the methods in those classes. `refactory extensionMethods` is perhaps one of the most interesting queries: it will return a list of all methods contained in the Refactory package but not contained within a Refactory class. This includes, for example, `String>>expandMacrosWithArguments:` and `Behavior>>parseTreeFor:`.

Since the PackageInfo naming conventions are based on those used already by Squeak, it is possible to use it to perform analysis even of code that has not explicitly adapted to work with it. For example, (`PackageInfo` named: 'Collections') `externalSubclasses` will return a list of all `Collection` subclasses outside the Collections categories.

You can send `fileOut` to an instance of `PackageInfo` to get a changeset of the entire package. For more sophisticated versioning of packages, see the Monticello project.

Part V

Seaside

Web dynamique avec Seaside

Main Author(s): N. Bouraqadi, Université Libre de Bruxelles, bouraqadi@ensm-douai.fr

15.1 Compléments sur Seaside

Quelques messages pour générer du html. Le destinataire de ces messages est l'objet passé en paramètre de la méthode `renderOn:` (instance de `WAHtmlRender`).

- `text:` 'chaîne de caractères' affiche simplement la chaîne de caractères.
- `heading:` 'texte du titre' `level:` niveau affiche un titre. Le deuxième paramètre est un entier qui correspond au niveau hiérarchique du titre (1 correspond au le plus grand)
- `break` introduit un retour à la ligne
- `horizontalRule` introduit une ligne horizontale
- `form:` ["définition de boutons, zones de saisies, "] définit un formulaire au sens Html. Nécessaire pour avoir des boutons et autres zones de saisies dans une page Html. Reçoit en paramètre un bloc qui contient les messages de création des boutons, zones de saisie,
- `textInputWithValue:` `valeurInitiale` `callback:` [:valeur | "traitements"] crée une zone de saisie simple (sans barre de défilement). La valeur initiale est celle qui est affichée au démarrage (nil pour ne rien afficher). Le dernier argument est un bloc qui reçoit comme paramètre la valeur saisie (valeur) dans le champ. Cette valeur peut être utilisée dans le traitement défini par le bloc. Ce bloc est exécuté quand la touche "Entrée" est pressée ou quand on clique sur un bouton du formulaire dans lequel se trouve la zone de saisie.
- `submitButtonWithAction:` ["traitements"] `text:` 'titre du bouton' ajoute un bouton qui a pour titre la chaîne de caractères passée comme deuxième argument. Un clic sur le bouton provoque l'exécution des traitements définis dans le bloc passé comme premier paramètre.

15.2 Encore des compteurs !

Il s'agit de réaliser encore un compteur, mais cette fois, il devra être accessible via le web (utilisation de Seaside). De plus, il devra être personnalisable dans la mesure où l'utilisateur doit pouvoir modifier directement la valeur du compteur et modifier l'incrément. Concrètement, vous devez définir une classe `CompteurPersonnalise` sous-classe de `WAComponent` qui représente une application Seaside. `CompteurPersonnalise` sera munie de :

- deux champs (`value` et `increment`),
- une méthode d'initialisation (`initialize`),
- ainsi que la méthode de génération du code html (`renderOn:`).

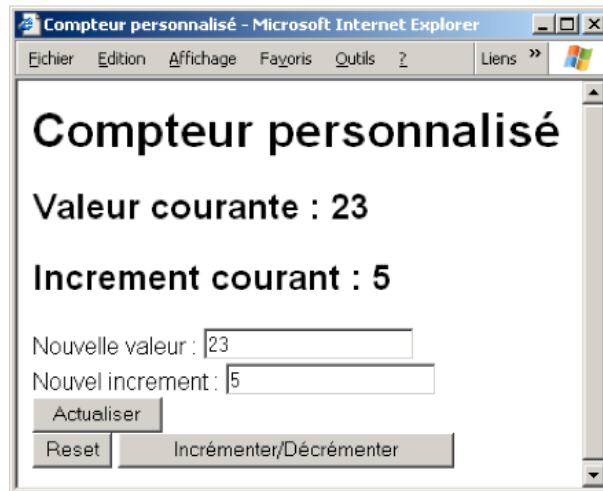


Figure 15.1: L'interface du compteur personnalisé

L'interface utilisateur doit être analogue à celle de la figure 15.1. Deux champs de saisie permettent de modifier la valeur du compteur et son incrément après clic sur le bouton "Actualiser". Le bouton "Reset" réinitialise le compteur (value mise à 0 et incrément mis à 1). Enfin, le bouton "Incrémenter/Décrémenter" permet d'ajouter l'incrément au compteur et donc de l'incrémenter si l'incrément est positif ou de le décrémenter dans le cas contraire.

15.3 Séparer l'interface du code métier

La structure suggérée pour l'exercice précédent n'est pas très propre. En effet, un même objet prend en charge à la fois le traitement (code métier : incrémenter/décrémenter, modification de l'incrément,) et l'interface utilisateur. Ce choix de conception rend difficile les éventuelles évolutions ou réutilisation. En particulier, si l'on souhaite changer d'interface utilisateur, voire de modèle de communication distante.

Dans cet exercice, on se propose de faire la séparation entre code métier et code d'interface et en illustrer l'utilité à l'aide d'un exemple simple. Cet exemple tourne autour d'une calculatrice arithmétique. Vous définirez tout d'abord la classe `Calculatrice` qui dispose de deux champs qui représentent respectivement l'opérande gauche et l'opérande droite. Munissez la classe d'accesseurs en lecture écriture à ces deux champs, ainsi que de 4 méthodes pour réaliser les 4 opérations arithmétiques. Bien entendu, ces quatre méthodes :

- ne prennent pas de paramètres,
- effectuent le calcul en utilisant les champs représentant les deux opérandes,
- et retournent le résultat du calcul

Définissez ensuite la classe `CalculatriceWeb` sous-classe de `WAComponent` qui représente une application Seaside. `CalculatriceWeb` permet l'utilisation à travers le web des opérations fournies par `Calculatrice`. Son interface s'apparente à celle donnée par la figure 15.2.

Vous allez maintenant exploiter la séparation entre code métier et code d'interface utilisateur. En effet, vous allez réutiliser la classe `Calculatrice` pour faire un nouveau compteur accessible via le web. L'interface devra être identique à celle du compteur de l'exercice précédent.

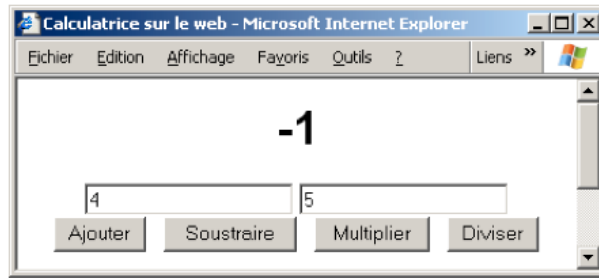


Figure 15.2: L'interface de la calculatrice.

15.4 Une application un peu plus sophistiquée

Il s'agit ici de définir un outil qui permet de gérer des tableaux blancs partagés via le web. Un tableau blanc est une zone de texte que plusieurs utilisateurs peuvent modifier. Chaque tableau est caractérisé par un nom et dispose d'une liste identifiant les utilisateurs qui ont le droit d'y accéder.

Chaque utilisateur dispose d'un identifiant et d'un mot de passe qu'il fournit pour se connecter. Une fois connecté il a le choix entre créer un nouveau tableau ou modifier tableau existant. Les utilisateurs qui ont accès à un tableau peuvent en modifier le contenu ainsi que la liste des utilisateurs qui ont accès au tableau.

16

A Simple Application for Registering to a Conference

Main Author(s): A. Bergel, Universitaet Bern, bergel@iam.unibe.ch

The goal of this tutorial is to give you a feeling on creating a web application using Seaside. RegConf is a tool intended to help people to register to a conference.

16.1 RegConf: An Application for Registering to a Conference

Four steps are necessary to complete a registration:

1. A participant has to enter some personal data such as firstname, name, the institute where she is attached, and her email address.
2. Then some information about the hotel are required. For instance a room can be single or double in an hotel ranked between 1 and 4 stars. A price has then to be computed.
3. Finally informations regarding the payment are required. Once the credit card number, the issue date, and the type are entered,
4. A confirmation screen shows a summary of what was entered.

The flow of the application is described in the following figure.

The dashed rectangle designate the part of the application which is *isolated*. This means that once the flow of the running application leaves this box, there is no way to come back in it, specially using the back button.

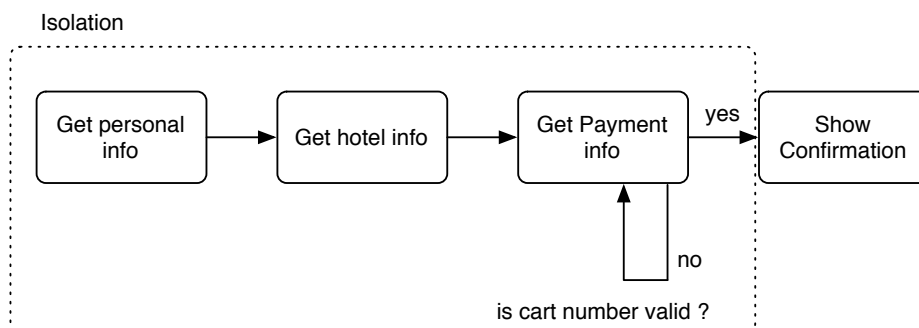


Figure 16.1:

16.2 Application Building Blocks

16.2.1 The Entry Point: RCMain

The control flow of the application has to be described in a task's `go` method. This method also represent the entry point of the application. Thus a name like `RCMain` sounds appropriated (`RC` stands for `RegConf`).

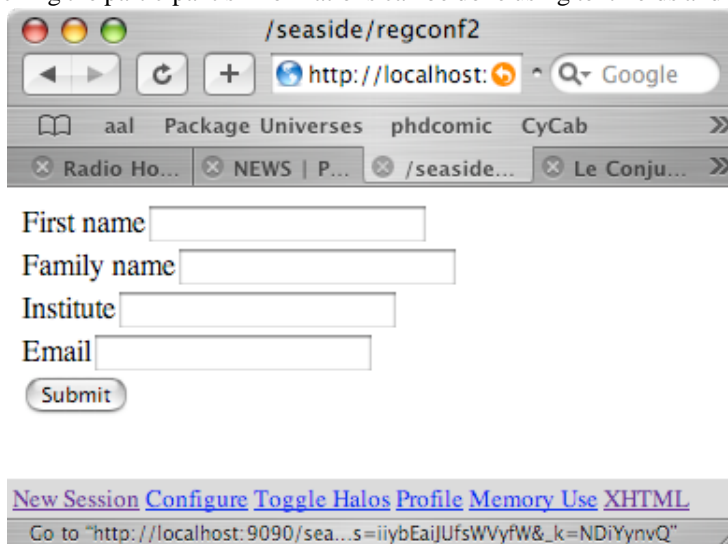
Your job: Create a task `RCMain` with a `go` method that describes the control flow of the application.

Your job: Start the web server on by executing `WAKom startOn: 9090`.

Your job: Create an `initialize` method on the class side to register your application in Seaside under the name `regconf`.

16.2.2 Getting User Information: RCGetUserInfo

All the control flow is defined in the class you previously defined. Getting user information is implemented as a normal seaside component (i.e., subclass of `WACComponent`). Instance variables of this class should reflect the structure of a user. Pressing the `submit` button returns to the caller component using `answer:`. Fetching the participant's informations can be done using text fields and submit button. Here is an example:



Your job: Write the method `renderContentOn:` in `RCGetUserInfo`.

Your job: Try your application using your favorite web browser. Make it point to `http://localhost:9090/seaside/regconf`.

The information passed around different states of the application can be contained in a dictionary. A more advanced design would require a class `User` for which an instance is passed around through.

16.2.3 Getting Hotel Information: RCGetHotelInfo

A list of choices is pleasant to fetch informations of the hotel.

Arrival date 07 Mars

Departure date 12 Mars

Hotel rank ***

Room single

Submit

[New Session](#) [Configure](#) [Toggle Halos Profile](#) [Memory Use](#) [XHTML](#)

Go to "http://validator.w3.org/check/referer"

Your job: Write the class RCGetHotelInfo

16.2.4 Payment: RCPayment

The payment is valid only if 16 number was provided and if the issue date is not over.

Cart Number 1234 1234 1234 1234

Issue date 0205

Card type mastercard

Submit

[New Session](#) [Configure](#) [Toggle Halos Profile](#) [Memory Use](#) [XHTML](#)

Your job: Write the class RCPayment

16.2.5 Confirmation: RCConfirmation

Once the payment is done, it is nice to show a summary of what was done.

Your job: Write the class RCConfirmation

16.3 Extensions

Your job: Study the class MiniCalendar of Seaside. Create a calendar starting from today. **Your job:** Use

the mini calendar to add the possibility to say when and until which day the person wants to keep the room.

Seaside Tutorial

Main Author(s): lukas renggli

17.1 Getting Started

Follow the instructions given on the slides to install Seaside. Make sure your Seaside server is up and running by accessing the example application at <http://localhost:8080/seaside/counter> in Squeak or at <http://localhost:8008/seaside/go/counter> in VisualWorks.

In Squeak load the monticello package `tutorial.mcz` and in VisualWorks the parcel `tutorial.pcl`. Both packages contain examples shown during the presentation and some class skeletons that will assist you to do these exercises.

Save your image. From now on work within a copy of this image, so that you can easy go back to a working configuration, in case you severely screw something up.

17.2 Development Tools

Exercise 67 Use your web browser to navigate to the counter example application. Toggle on the halos to see the border of the component this application is built of. Experiment and interact with the application in render- and source-mode.

Exercise 68 Change the behaviour of the increase and decrease buttons: edit the methods `#increase` and `#decrease` from within the web browser to increase by 2 and decrease by 3.

Exercise 69 Inspect the living component from within the web browser. There are two instance variables visible, whereas `COUNT` is representing the state of the component. The other instance variable is defined in a super-class of `WACounter` and will be discussed later on.

Exercise 70 Change the background color of the web application by using the style editor from within your web browser. Try using something like `body { background-color: yellow; }`.

Question 71 Why do you think the style editor is used more often in industrial settings than the system browser?

Exercise 72 Introduce an error to the method `#increase` using your web browser. Play with your application so that the error occurs. Click on the *debug* link which opens a debugger within your image. Fix the bug and proceed the evaluation.

17.3 Control Flow

During the theoretical part an example was shown where the user had to guess a number the computer was thinking of. In this exercise we will have a look at the implementation of two similar games. Some skeletons are provided, so you don't need to implement all by yourself.

17.3.1 User Guesses a Number

Exercise 73 Have a look at the source code of `STUserNumberGuesser` in the package *Tutorial-Flow* and play the game several times to make sure it works as expected.

Exercise 74 Modify the method `#go` in `STUserNumberGuesser` to count the number of guesses. Show the total number of guesses the user required to get the right number in the end of the game.

Question 75 Try using the back button while playing the game. How does the application handle this?

Question 76 What happens if you open multiple windows in the same session and play within the different windows independently?

Question 77★ Is it possible to cheat the counter by using the back button or by opening new windows within the same session? Does this behavior change if you use an instance variable instead of a temporary one for counting?

17.3.2 Computer Guesses a Number

Exercise 78 Write a new web application that allows the computer to guess a number the user is thinking of. In case you run into troubles, you can always have a look at the implementation of `STUserNumberGuesser`.

1. Create a subclass of `WATask` called `STComputerNumberGuesser`.
2. Create an initialization method on the class side of the newly created class, registering the component as a new web application with the path segment `cng`.
3. Implement the method `#go` following the rules of the game. Use `#inform`: to tell the user what he should do and `#confirm`: to ask the user if the guess of the computer is too big.
4. Play the game several times to make sure it works as expected.

Exercise 79 Implement yet another task asking the user if he wants to guess or not. Depending on the answer either call `STUserNumberGuesser` or `STComputerNumberGuesser`. Modify those two classes to answer the numbers of steps required and call them from within your new task. Don't forget to register your new application with a class initialization method.

17.3.3 TicTacToe Game

There are three prepared classes for this game in the package *Tutorial-TicTacToe* following the *MVC-Pattern*:

Model `STTicTacToeController` is a simple model of a game holding the current board configuration. It includes methods to access and modify its configuration (`#boardAt`: and `#boardAt:put:`) and to call an algorithm in order to look for the best possible move of a given player (`#find`:).

View `STTicTacToeView` is a simple Seaside view onto the game model. You will learn later on how to create views with Seaside.

Controller `STTicTacToeController` is a subclass of `WATask` and this is the place that needs your work now. It already implements a few convenience methods like `#newModel`, `#computerMove` and `#userMove`.

Exercise 80 Register `STTicTacToeController` as a new web-application, but this time don't use a class initialization method but the configuration interface. Make sure that you have a method `#canBeRoot`

on the class-side so that Seaside recognizes this class as a possible root of a web application. Browse to <http://localhost:8080/seaside/config> when using Squeak or <http://localhost:8008/seaside/go/config> when using VisualWorks, enter your password, add a new entry point with the name `ttt` and select `STTicTacToeController` as the root component.

Exercise 81 Implement the game in the method `#go` using the provided convenience methods. You will also need some testing methods of the model to check if the game is finished (`#isFinished`) and who was the winner (`#winner`). Don't put all your code into one single method, split it among different ones to ensure readability. Ask the user in the beginning of the game if he prefers to start playing or not.

Exercise 82★ Ensure that the user can't cheat the game by using the back button of the web browser. Don't wrap too much or too few of your code into `#isolate:` blocks.

17.4 Components

For the rest of this tutorial we will be working on an example of a possible real-world web application: it should be useable by a theater having different plays in its program. The application should manage the plays, the shows and the booking of the tickets.

17.4.1 Introduction

Here we will be starting step by step building up this project. Follow the exercises one by one as they depend on each other. However don't let you hinder from bringing in your own ideas and from implementing some extra features, if you think they could be useful for this project.

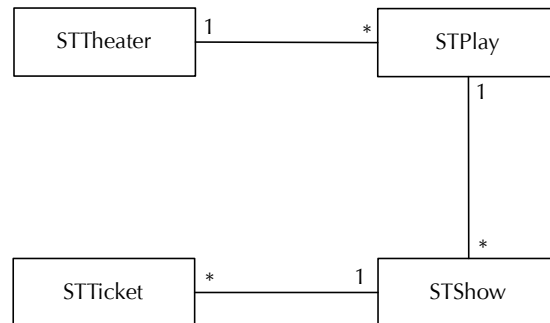


Figure 17.1: Theater-Model

All the code altogether should be put into the bundle *Tutorial-Theater* that contains some packages, namely *Theater-Model*, *Theater-View* and *Theater-Tests*. The package *Theater-Model* contains a very simple model, as seen in Figure 17.1, to be used to build up a web-interface around. Feel free to enhance the model when you need to do so, but do run the tests and add new ones to make sure that all the features work as expected after your modifications.

On the class side of `STTheater` you can find a method `#default` returning the domain model to be used for the web application. Usually you do not keep your model just within the image, but use a proper external storage mechanism instead: this can be simply done by dumping out the object graph to the filesystem from time to time or by using a relational- or object-database. However, as possible storage strategies are out of the scope here, we will just keep everything within the image.

Exercise 83 Start out by creating a new task called `STBuyTicketTask` that will model the steps required to buy a ticket. Register it as a new Seaside application as you will need it later on to test your components. Leave the method `#go` empty for now. This method should define the flow as seen in Figure 17.2 by the end of the tutorial.

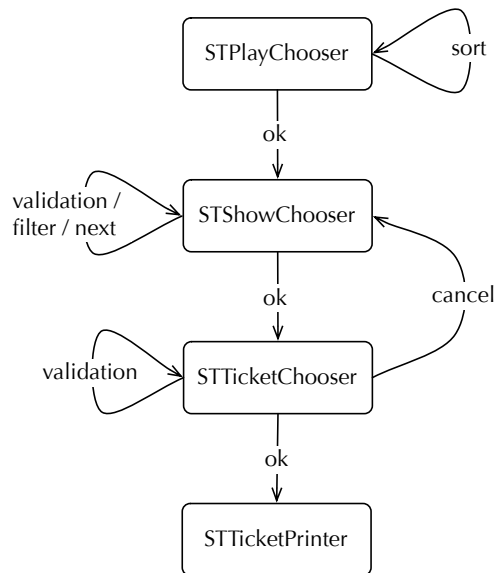


Figure 17.2: Theater-Flow as defined by STBuyTicketTask

17.4.2 Choosing a Play

[Title](#) [Kind](#) [Author](#)

Der Bus (Schauspiel) - Lukas Bärfuss

Ein Bus nachts am Waldrand. Hermann, der Busfahrer, entdeckt einen blinden Passagier: Erika. Sie ist eine Pilgerin. Und sie hat ein Problem, sie sitzt im falschen Bus. Hermann will sie aussetzen, bedroht sie. Keiner hilft. Der Harte ihrer Mitmenschen hält Erika ihre radikale Gläubigkeit entgegen. Was, wenn die Bibel doch Recht hat, Satz für Satz, Wort für Wort? Der junge Berner Autor Lukas Bärfuss befragt in seinem ersten Stück am Stadttheater Bern den Zusammenhang von Religion und Gewalt, Erlösung und Unmenschlichkeit. Schweizer Erstaufrührung.

Gespenster (Schauspiel) - Henrik Ibsen

Der junge Maler Oswald kehrt aus Paris zurück zu seiner Mutter Helene Alving. Pastor Manders, den sie einst zurückgewiesen hatte, kommt ebenfalls in die Stadt, um das Kinderheim zu eröffnen, das Frau Alving mit dem Erbe ihres verstorbenen und verhassten Mannes stiftete. Doch die Toten ruhen nicht. Der Kern der anständigen bürgerlichen Familie ist verrotten. Die Vergangenheit hat einen langen Atem: Er reicht in die Zukunft.

Figure 17.3: View of STPlayChooser

Exercise 84 Create a subclass of WAComponent called STPlayChooser that will give the user the possibility to choose a theater-play. Add an instance variable `plays` and create accessors to hold a collection of plays that should be displayed with this component. Call your newly created component from STBuyTicketTask, but don't forget to initialize it with the collection of plays. If you browse to your application, you should get a blank page as you haven't defined any view yet.

Exercise 85 Implement the method `#renderContentOn:`. As a first step, enumerate the plays and display the title of each. If you go back to your web browser and refresh, you should see the titles now. Then display the other information you get from the model. Use your own style sheet or copy the example from Figure 17.4 to make the output look like Figure 17.3.

Exercise 86 So far there is no interaction possible with the component. Create an anchor-callback `#anchorWithAction:do:` around the title and answer the selected play to the caller. Test your code by extending the task that is calling your component and inform the user about the selected play.

Exercise 87★ To set up the list of the plays more convenient, add three links at the top of the page to

```

.sort {
  background: #eeeeee;
  padding: 5px;
}
.play {
  margin-top: 10px;
}
.play .head {
  font-size: 16pt;
}
.play .body {
  margin-left: 10px;
  width: 490px;
}

```

Figure 17.4: Stylesheet of STPlayChooser

make it possible to sort the plays according to `#title`, `#kind` or `#author`. To remember the state of the selected sort order you need to add another instance variable. Make it also possible to sort in reverse order by clicking a second time onto the same link.

17.4.3 Choosing a Show

Exercise 88 Create another subclass of `WACComponent` called `STShowChooser` that allows the user to choose a show. Add instance variables to hold a collection of shows to choose from and one for the current selection. Create appropriate accessors and call your newly created and properly initialized component from `STBuyTicketTask`.

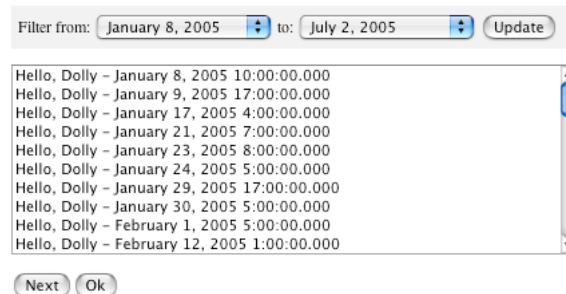


Figure 17.5: View of STShowChooser

Exercise 89 Implement the method `#renderContentOn:` using Figure 17.5 as a reference; don't worry about the filter yet. Make sure hitting `ok` only answers if the user actually selected a valid show, else show a message that a selection is missing and return to the dialog. Add a button to select the next possible show automatically.

Exercise 90★ Implement a facility to allow filtering for a certain date range. Write a method returning a possible list of dates and add two instance variables to keep the selected date for start and end of the period to be filtered. Render two drop-down boxes and a button to update the filtered list. Use live-callbacks to update the list of shows without the need to press the update button anymore.

Exercise 91★ Experiment with other form controls. How does the interface look like when using option-boxes instead of the list? What do you need to change in the code?

17.4.4 Buying and Printing Tickets

Exercise 92 Write a component that allows the user to select the number of tickets he wants to buy. Give an error message, if there are not enough places available for the selected show or if the user doesn't enter a valid number. Update the domain model according to the tickets sold and answer a collection of tickets to the task. The view of a minimal implementation can be seen in Figure 17.6.

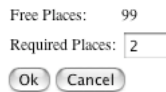


Figure 17.6: View of STTicketChooser

Exercise 93 Last but not least write yet another component printing out a collection of tickets. This might look like Figure 17.7. No links or form elements are required in this component. Update your flow accordingly.

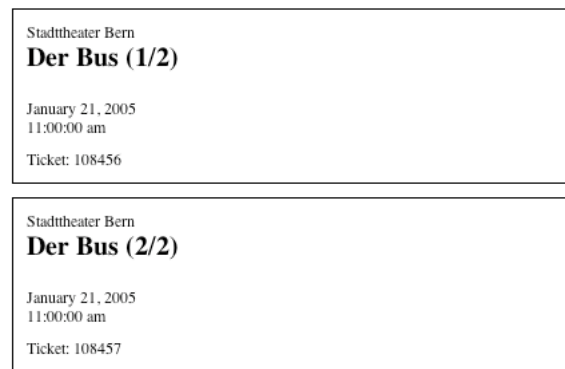


Figure 17.7: View of STTicketPrinter

Exercise 94 Make sure that your application implements all the paths that are visible in the state diagram in Figure 17.2. Make sure that the user cannot go back after having bought the tickets.

17.5 Composition

In this section we will compose different components we have written before. Create a few more components and plug together an appealing and simple user interface.

17.5.1 Frame, Subcomponent and Backtracking

Exercise 95 Create a new subclass of WAComponent and register it as a new entry point to your application. Render into different div-tags the name of the theater and the current season; you can find this information in the model. Also create a simple menu that is empty for now. Create a style-sheet to make the application look nicer.

Exercise 96 Add an instance variable to your main-frame to hold a child component. Create a method #buyTicket that initializes the variable with a new instance of STBuyTicketTask and send #buyTicket in the initialization method of the component. Place the child beside the menu you have created before. Don't

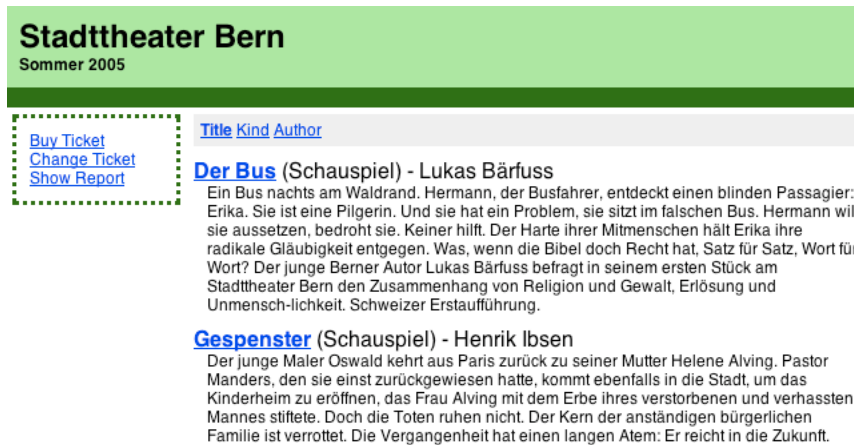


Figure 17.8: View of STMainFrame

forget to implement the message `#children`, else you will sooner or later run into troubles. Create a menu item called *Buy Ticket* that sends the message `#buyTicket` when clicked. Enjoy the application with the halos turned on.

Exercise 97★ Test the new functionality you implemented. Especially try out the behavior of the application when using the back-button. Try clicking on *Buy Ticket*, hit the back-button of your web-browser and then click on any link or control within the child-component. Why do you get an error? Fix the problem and make sure everything works as expected.

17.5.2 Reuse of Components

In this part of the exercises you are basically free about the implementation details of a new requirement of the application: The theater company wants to be able to let the customers return tickets and exchange them with another one from the same play but a different show.

Exercise 98 Use the id of the ticket to identify the one to be replaced. Probably you need to improve the model to make the necessary mutations possible. Also write tests to ensure it works as expected. For the web interface try to write as few lines of code as possible. Reuse the existing components that you have written in the previous steps. You might also want to use components provided by the framework. The example solution requires 7 lines of code, including the validation of the ticket id. Can you do it with less lines of code?

17.5.3 Reporting and Batching

Exercise 99 Create a new component called `STShowReport` showing a report of all the shows from the model as seen in Figure 17.9. Use `WABatchedList` to enable the batching of the huge list and only display 10 items at once. For the reporting you might want to use `WATableReport` or write your own component. By default the list should be sorted according to the timestamp. Add the new component to the menu in the main-frame.

17.5.4 Editing a Play

In this part we are going to implement a dialog to edit the attributes of a play. Have a look at Figure 17.10 to get an idea of the look. Add a link to the title of every play in your report that calls the component you are going to create in the following exercise:

STShowReport							[R S]
WATableReport							[R S]
Play	Kind	Author	Timestamp	Free	Sold	Total	
Wiener Blut	Operette	Johann Strass	January 2, 2005 23:00:00.000	100	0	100	
Hidden Garden	Tanzabend	-	January 3, 2005 0:00:00.000	100	0	100	
Hidden Garden	Tanzabend	-	January 4, 2005 1:00:00.000	100	0	100	
I Puritani	Oper	Vincenzo Bellini	January 8, 2005 11:00:00.000	100	0	100	
Der Bus	Schauspiel	Lukas Bärfuss	January 8, 2005 11:00:00.000	100	0	100	
Hidden Garden	Tanzabend	-	January 8, 2005 21:00:00.000	100	0	100	
Wiener Blut	Operette	Johann Strass	January 13, 2005 7:00:00.000	100	0	100	
I Puritani	Oper	Vincenzo Bellini	January 15, 2005 6:00:00.000	100	0	100	
Hidden Garden	Tanzabend	-	January 15, 2005 20:00:00.000	100	0	100	
Der Bus	Schauspiel	Lukas Bärfuss	January 16, 2005 16:00:00.000	100	0	100	
WABatchedList							[R S]
<< 1 2 3 4 5 6 7 8 9 10 >>							

Figure 17.9: View of STShowReport with halos toggled on

Exercise 100 Create a new subclass of WAComponent and add an instance variable to hold the play. In the method #initialize wrap the component with two decorations:

1. WAFromDecoration to render a form around the component and display *ok* and *cancel* buttons.
2. WValidationDecoration to validate the input fields and display an error message if necessary.

Ensure that the validation errors are properly displayed and that the model isn't touched when hitting cancel.

An author is required.

Title:

Kind:

Author:

Description:

Dollys Levis Betätigungsfeld ist die Ehevermittlung. Ein besonders schwieriger Kunde ist der geizige Mr. Vandergelder, schwer vermittelbar und schwerreich, den sich Dolly zu ihrem eigenen Gatten heranziehen will. Um Einfälle und Intrigen ist Dolly nie verlegen, und so zwingt sie nicht nur Vandergelder in sein Glück, sondern auch noch drei weitere Liebespaare.

Figure 17.10: View of STEditPlay

Exercise 101★ Load Mewa and try to write the same dialog using a descriptive meta model.

17.6 Advanced

17.6.1 Continuations

To answer the following question it might be useful to have a look at the class Continuation. You might also want to run the different tests of ContinuationTest and type and evaluate a few expressions in the workspace.

Question 102 When should one *not* use a continuation based web framework?

Question 103 How are continuations implemented in Smalltalk? Why are there no primitives required?

Question 104 What about the time- and space-performance of continuations?

Question 105★ Why is the implementation of the class Continuation polymorphic to BlockClosure? What are the differences?

Question 106★ When are ensure-blocks evaluated, if you create a continuation within a protected context?

17.6.2 Bookmark-able URLs

Exercise 107 Implement #updateUrl: in the three top-level sub-components of your web application and add an appropriate path-element to the URL. Depending on the context of your application, the URL should now look like: ../theater/buy, ../theater/change and ../theater/report.

Exercise 108 So far it isn't possible to navigate to these sub-components directly using an URL. To get the desired result, create a subclass of WARenderLoopMain called STRenderLoopMain and override the message #start: to parse the URL and to setup the root component as requested.

Part VI

Object-Oriented Design

A Simple Application: A LAN simulation

Main Author(s): Ducasse, Wuyts

Basic LAN Application

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

Creating the Class Node

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

Node inherits from **Object**

Collaborators: **Node** and **Packet**

Responsibility:

name (aSymbol) - returns the name of the node.

hasNextNode - tells if a node has a next node.

accept: aPacket - receives a packet and process it.

By default it is sent to the next node.

send: aPacket - sends a packet to the next node.

Exercise 109 Create a new package **LAN**, and create a subclass of **Object** called **Node**, with two instance variables: **name** and **nextNode**.

Exercise 110 Create accessors and mutators for the two instance variables. Document the mutators to inform users that the argument passed to **name**: should be a **Symbol**, and the arguments passed to **nextNode** should be a **Node**. Define them in a **private** protocol. Note that a node is identifiable via its name. Its name is part of its public interface, so you should move the method **name** from the **private** protocol to the **accessing** protocol (by drag'n'drop).

Exercise 111 Define a method called **hasNextNode** that returns whether the node has a next node or not.

Exercise 112 Create an instance method **printOn:** that puts the class name and name variable on the argument **aStream**. Include my next node's name **ONLY** if there is a next node (Hint: look at the method

printOn: from previous exercises or other classes in the system, and consider that the instance variable name is a symbol and nextNode is a node). The expected printOn: method behavior is described by the following code:

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

```
Node named: Node1 connected to: PC1
```

Exercise 113 Create a **class** method new and an **instance** method initialize. Make sure that a new instance of Node created with the new method uses initialize (see previous exercise). Leave initialize empty for now (it is difficult to give meaningful default values for the name and nextNode of Node. However, subclasses may want to override this method to do something meaningful).

Exercise 114 A node has two basic messages to send and receive packets. When a packet is sent to a node, the node has to accept the packet, and send it on. Note that with this simple behavior the packet can loop infinitely in the LAN. We will propose some solutions to this issue later. To implement this behavior, you should add a protocol send-receive, and implement the following two methods -in this case, we provide some partial code that you should complete in your implementation:

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

```
...
```

```
send: aPacket
  "Precondition: self have a nextNode"

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

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

```
...
```

Creating the Class Packet

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

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

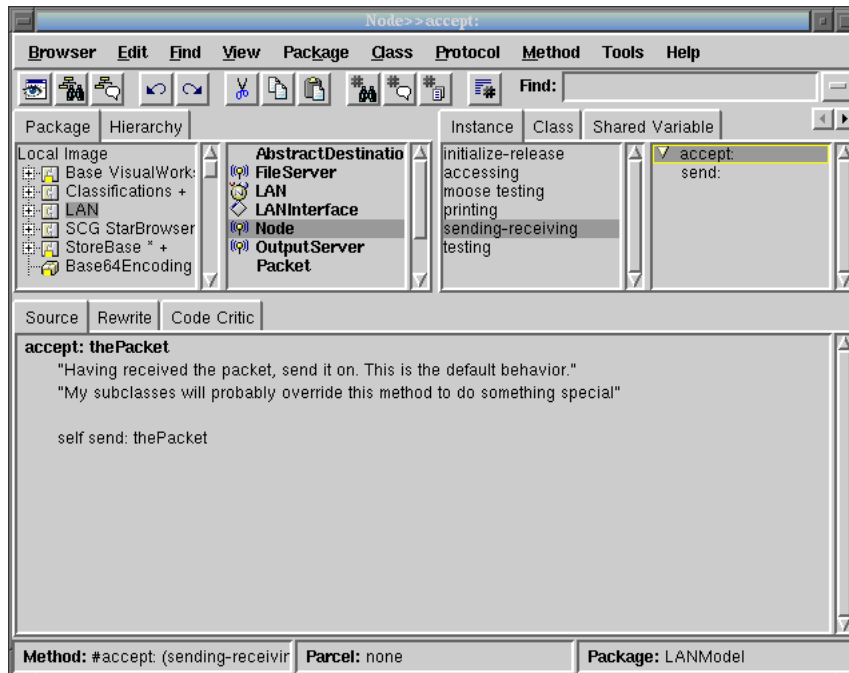


Figure 18.1: Definition of accept: method

Exercise 115 In the LAN, create a subclass of **Object** called **Packet**, with three instance variables: **contents**, **addressee**, and **originator**. Create accessors and mutators for each of them in the **accessing** protocol (in that particular case the accessors represents the public interface of the object). The addressee is represented as a symbol, the contents as a string and the originator has a reference to a node.

Exercise 116 Define the method **printOn: aStream** that puts a textual representation of a packet on its argument **aStream**.

Creating the Class Workstation

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

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

Exercise 117 In the package **LAN** create a subclass of **Node** called **Workstation** without instance variables.

Exercise 118 Define the method `accept: aPacket` so that if the workstation is the destination of the packet, the following message is written into the Transcript. Note that if the packets are not addressed to the workstation they are sent to the next node of the current one.

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

A packet is accepted by the Workstation Mac

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

Exercise 119 Write the body for the method `originate:` that is responsible for inserting packets in the network in the method protocol `send-receive`. In particular a packet should be marked with its originator and then sent.

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

Creating the class `LANPrinter`

Exercise 120 With nodes and workstations, we provide only limited functionality of a real LAN. Of course, we would like to do something with the packets that are travelling around the LAN. Therefore, you will now create a class `LanPrinter`, a special node that receives packets addressed to it and prints them (on the Transcript). Note that we use the name `LanPrinter` to avoid confusion with the existing class `Printer` in the namespace `Smalltalk.Graphics` (so you could use the name `Printer` in your namespace or the `Smalltalk` namespace if you really wanted to). Implement the class `LanPrinter`.

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

Simulating the LAN

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

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


— mac pc node1 node2 igPrinter —

"create the nodes, workstations, printers and fileserver"

```
mac := Workstation new name: #mac.  
pc := Workstation new name: #pc.  
node1 := Node new name: #node1.  
node2 := Node new name: #node2.  
node3 := Node new name: #node3.  
igPrinter := Printer new name: #IGPrinter.
```

"connect the different nodes."

```
mac nextNode: node1.  
node1 nextNode: node2.  
node2 nextNode: igPrinter.  
igPrinter nextNode: node3.  
node3 nextNode: pc.  
pc nextNode: mac.
```

"create a packet and start simulation"

```
packet := Packet new  
    addressee: #IGPrinter;  
    contents: 'This packet travelled around  
to the printer IGPrinter.
```

```
mac originate: packet.
```

anotherSimpleLan

"create the nodes, workstations and printers"

```
|mac pc node1 node2 igPrinter node3 packet |  
mac:= Workstation new name: #mac.  
pc := Workstation new name:#pc.  
node1 := Node new name: #node1.  
node2 := Node new name: #node2.  
node3 := Node new name: #node3.  
igPrinter := LanPrinter new name: #IGPrinter.
```

"connect the different nodes."

```
mac nextNode: node1.  
node1 nextNode: node2.  
node2 nextNode:igPrinter.  
igPrinter nextNode: node3.  
node3 nextNode: pc.  
pc nextNode: mac.
```

"create a packet and start simulation"

```
packet := Packet new  
    addressee: #anotherPrinter;  
    contents: 'This packet travels around  
to the printer IGPrinter'.  
pc originate: packet.
```

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

Creating the Class **FileServer**

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

FileServer inherits from **Node**

Collaborators: **Node** and **Packet**

Responsibility:

accept: aPacket - if the packet is addressed to the file server save it (Transcript trace) else send the packet to the following node.

save: aPacket - save a packet.

Fundamentals on the Semantics of Self and Super

Main Author(s): Ducasse, Wuyts

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

19.1 self

When the following message is evaluated:

```
aWorkstation originate: aPacket
```

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

```
Workstation>>originate: aPacket
```

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

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

```
Packet>>originator: aNode
```

```
originator := aNode
```

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

```
Node>>send: thePacket
```

```
self nextNode accept: thePacket
```

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

19.2 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 121 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.

Object Responsibility and Better Encapsulation

20.1 Reducing the coupling between classes

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

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

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

20.1.1 Current situation

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

```
Workstation>>accept: aPacket
```

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

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

20.1.2 Solution.

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

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

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

20.2 A Question of Creation Responsibility

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

We will find a solution to these problems.

Exercise 122 Define a class method named `withName:` in the class `Node` (protocol ‘instance creation’) that creates a new node and assign its name.

```
withName: aSymbol
```

```
....
```

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

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

```
....
```

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

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

```
send: aString to: aSymbol
```

```
....
```

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

Exercise 123 Rewrite the `new` method of the class `Node` and `Packet` as the following:

```
new
```

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

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

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

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

Exercise 124 Update and rerun your tests to make sure that your changes were correct.

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

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

20.3 Reducing the coupling between classes

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

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

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

20.3.1 Current situation

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

```
Workstation>>accept: aPacket
```

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

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

20.3.2 Solution.

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

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

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

20.4 A Question of Creation Responsibility

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

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

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

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

```
Packet class>>new
```

```
...
```

```
Packet>>initialize
```

```
...
```

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

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

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

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

Instead of writing:

```
Packet>>hasDefaultContents
```

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

```
Packet>>initialize
```

```
...
```

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

```
...
```

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

```
Packet>>defaultContents
```

```
^ 'no contents'
```

```
Packet>>initialize
```

```
...
```

```
contents := self defaultContent
```

```
...
```

```
Packet>>hasDefaultContent
```

```
^ contents = self defaultContents
```

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

20.5 Proposing a creational interface

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

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

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

```
....
```

```
Packet class>>to: aSymbol
```

```
....
```

The class Node. Now apply the same techniques to the class **Node**. Note that you already implemented a similar schema that the default value in the previous lessons. Indeed by default instance variable value is `nil` and you already implemented the method `hasNextNode` that to provide a good interface.

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

```
Node class>>withName: aSymbol
```

```
....
```

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

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

```
....
```

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

20.6 Forbidding the Basic Instance Creation

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

Let us imagine that we want to ensure that no instance can be created without calling the methods we specified. We simply redefine the creation method `new` so that it will raise an error. Rewrite the `new` method of the class **Node** and **Packet** as the following:

```
Node class>>new
```

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

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

20.6.1 Remarks and Analysis.

A first solution could be the following code:

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

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

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

The solution is to rewrite the method such as:

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

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

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

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

20.7 Protecting yourself from your children

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

```
Node class>>withName: aSymbol
```

```
^ self new name: aSymbol
```

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

```
Node class>>withName: aSymbol
```

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

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

```
Node class>>withName: aSymbol
```

```
^ self new name: aSymbol ; yourself
```

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

Hook and Template Methods

Main Author(s): Ducasse and Wuyts

In this chapter you will learn how to introduce hooks and template methods to favor extensibility. First we look at the current situation and introduce changes step by steps.

21.1 Providing Hook Methods

Current situation. The solution proposed for printing a `Node` displays the following string `Node` named: `Node1` connected to: `PC1` obtained by executing the following expression:

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

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
```

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:

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

```
'Node Mac connected to Printer PC1'
```

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

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

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

Exercise 125 Study the method `printOn:` on the class `Object`. Check its implementors and senders.

Exercise 126 Study the method `copy` on the class `Object`. Check its implementors and senders. What do you think about the method `postCopy` check its senders and implementors.

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.

Main Author(s): Ducasse

22.1 From a Ring to a Star

Right now your lan is a ring, the token has to pass through the nodes one by one and this is not possible to send a packet to multiple nodes at once as this is the case in star architecture.

Exercise 127 Propose a solution to this problem. Hint having a new kind of node holding references to its connected node but acting as a normal node can be a good solution.

22.2 Handling Loops

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

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

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

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

1. Each node keeps track of the packets it already received. When a packet already received is asked to be accepted again by the node, the packet is not sent again in the LAN. This solution implies that packet can be uniquely identified. Their current representation does not allow that. We could imagine to tag the packet with a unique generated identifier. Moreover, each node would have to remember the identity of all the packets and there is no simple way to know when the identity of treated node can be removed from the nodes.
2. Each packet keeps track of the node it visited. Every time a packet arrived at a node, it is asked if it has already been here. This solution implies a modification of the communication between the nodes and the packet: the node must ask the status of the packet. This solution allows the construction of different packet semantics (one could imagine that packets are broadcasted to all the nodes, or have to be accepted twice). Moreover once a packet is accepted, the references to the visited nodes are simply destroyed with the packet so there is no need to propagate this information among the nodes.

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

Packet inherits from Object

Collaborators: Node

Responsibility:

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

contents describes the contents of the message sent.

originator references the node that sent the packet.

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

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

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

-
- New instance variable. A packet needs to keep track of the nodes it visited. Add a new instance variable called `visitedNodes` in the class **Packet**. We want to collect the visited nodes in a set. Browse the class **Set** and its superclass to find the function you need.
 - Initialize the new instance variable. Modify the initialize methods of the class **Packet** so that the `visitedNodes` instance variable is initialized with an empty set.
 - Node Acceptation Methods. In a protocol named 'node acceptance', define the method `isAcceptableBy:` and `hasBeenAcceptedBy:`.
 - Test if your implementation works by sending a 'bad' node with a bad originator into the LAN.

22.3 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 have such a behavior specified only once and shared by all our Lan classes.

Define a class **LanObject** that inherits from **Object**, implements an instance method `initialize` and a class method `new` that automatically calls the `initialize` method on the newly created object and return 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.

22.4 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 file servers 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.

22.5 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.

22.6 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.

22.7 Automatic Naming

The name of a node has to be specified by its creator. We would like to have an automatic naming process that occurs when no name is specified. Note that the names should be unique. As a solution we propose you to use a counter, as this counter has 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 functionality. We also would like that all the printer names start with Pr. Propose a solution.

Workstation Mac connected to Printer PC

Building an Interface in VW

In this lesson you will define a user interface for your application in VisualWorks.

Main Author(s): Wuyts

23.1 ApplicationModel: the Glue between Domain and Widgets

The class ApplicationModel already:

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

Our application subclass will have to implement

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

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

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

- building the interface
- programming the applicationModel

23.2 Building the interface

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

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

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

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



Figure 23.1: to do

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

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

Step 2: painting the canvas We will now paint the widgets such that our interface looks 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).



Figure 23.2: to do

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

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

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

1. to select a widget: click it once
2. to deselect a widget: hold down the shift button while clicking on the selected widget, or click somewhere outside the widget
3. to resize a widget: select the widget, click on one of the handles of the widget (the black squares at the outside of the widget) and resize it.
4. to move a widget: select it, press the select button between the handles of the widget and move it
5. to cut/copy a widget: select the widget, bring up the operate menu and select cut/copy in the edit menu

6. to paste a widget (once you have cut/copied it): bring up the operate menu anywhere on a canvas, and select paste from the edit menu. The pasted widget is automatically placed at the same position as the widget that is cut/copied, and is automatically selected. You can now move it to another position.

Exercise 128 copy the one button widget that is currently on the canvas to make a second one, and position the two buttons according to the picture of the application.

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

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

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

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

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

Exercise 129 : Set and apply the following properties for the left button:

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

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

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

To install a canvas:

- click Install... in the canvas tool
- a dialog box comes up where you have to provide the name of the application model and the class method in which to install the canvas. Provide `SimpleCounterApp` as class name. Leave `windowSpec` (the default name of the class method where the interface specification is stored) as name of the selector. Press OK when finished.

- since your application model does not exist yet, you get another dialog box where you have to provide some information concerning your application model. Leave the name of the class, but provide **DemoCounter** as name for the category. Since we are creating a normal application (and not a dialog box or so), choose the application check box. Note that VisualWorks then fills in **ApplicationModel** as superclass. Leave this and select OK. Select a second time OK to close the first dialog box.

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

23.3 Programming the application model

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

23.4 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:

```
SimpleCounter>>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 lazy 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 lazy initialization can be replaced by the following methods:

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

```
SimpleCounter>>counterValue
  ^ counterValue
```

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

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

Exercise 130 Provide the implementation for increment and decrement, and test it.

Part VII

Advanced Smalltalking

missing compiler td of bernard, actalk, objVlisp implementations

CodeScope

Main Author(s): S. Ducasse, Université de Savoie, stephane.ducasse@univ-savoie.fr

The fact that Smalltalk classes and methods are objects too make easy to analyze applications written in Smalltalk. We ask you to build an application analyzing applications, classes and methods and producing reports using Seaside.

Here are some screenshots of a possible implementation of CodeScope.

CodeScope

Package:

Number of classes: 13
 Number of inst vars: 45
 Number of lines of code: 4254
 Method average lines of code: 8.42s2
 Class average lines of code: 327.23s2
 Class average of inst vars: 3.46s2
 Number of uncommented classes: 1

Class:

Hierarchy:

- Collection
- SequenceableCollection
- ArrayedCollection
- ByteArray
- CompiledMethod

Hierarchy nesting level: 7
 Number of methods: 108
 Number of inst vars: 0
 Number of extending methods: 0
 Number of overriding methods: 10
 Number of accesses to inst vars: 0
 Number of instances: 54788
 Number of uncommented methods: 32

Method:

Number of lines: 10
 Number of arguments: 2
 Has comment? yes

Class Method:

Number of lines: 2
 Number of arguments: 1
 Has comment? no

CodeScope

Package:

Number of classes: 21
 Number of inst vars: 21
 Number of lines of code: 3638
 Method average lines of code: 7.65s2
 Class average lines of code: 173.23s2
 Class average of inst vars: 1s0
 Number of uncommented classes: 7

Class:

Hierarchy:

- Collection
- SequenceableCollection
- ArrayedCollection
- String
- Symbol

Hierarchy nesting level: 6
 Number of methods: 213
 Number of inst vars: 0
 Number of extending methods: 3
 Number of overriding methods: 39
 Number of accesses to inst vars: 0
 Number of instances: 217932
 Number of uncommented methods: 58

Method:

Number of lines: 14
 Number of arguments: 0
 Has comment? yes

Class Method:

Here is the list of information that we would like to gather from one application and some hints how you can get this information. We consider a category as a package.

Methods

- Number of lines (without entry comments)
- Number of arguments

Classes

- Hierarchy nesting level
- Number of methods / class methods
- Number of instance variables
- Number of uncommented classes
- Number of extending methods (sending super)
- Number of overriding methods
- Number of accesses to instance variables
- Number of instances existing in the system (allInstances)
- Number of uncommented methods (aClass firstCommentAt: sel)

Packages

- Number of classes in a package (inspect SystemOrganization, Smalltalk at: #Array).
- Method average line of codes
- Class average line of codes
- Number of class reference inside/outside the application
- Total and average number of (lines of code, instance variables....)

Implementing Scaffolding Patterns

The following is an excerpt from Expedient Smalltalk Programming "Smalltalk Scaffolding Patterns" Jim Doble, Allen Telecom Systems, Ken Auer, President, RoleModel Software, Inc. published in Patterns for Rapid-Prototyping in Smalltalk written with Jim Doble for the PLoP'97 conference and eventually published in Pattern Languages of Program Design 4 and available at: <http://www.rolemodelsoftware.com/more-AboutUs/publications/articles/scaffold.php>

The goal of the current exercise is to write the code to implement the presented scaffolding patterns. Therefore read them and follow the instructions.

25.1 Introduction

"Get it right the first time that's the main thing" Billy Joel [Joel77]

Billy Joel's credo is compelling. Then again, it's quite possible that Billy Joel has never written a lick of software in his life! And he's probably happier for it because software is notoriously difficult to get right the first time. Any software development plan that assumes software can be completed right the first time fails to recognize the experimental nature of software development. For a software design team, the course of a project is a learning experience. If an important bit of learning, such as discovery of requirements misunderstandings or design inadequacies, occurs too late in the project, the result can be project failure.

Prototyping tackles this issue head-on. The goal of prototyping is to carry out development experiments designed to allow important learning to occur earlier in the project cycle. By moving this learning forward in time, the development team has time to react to what is learned, and the risk of project failure is significantly reduced. Unfortunately, when projects are under significant schedule pressure, expenditure of time and resources on prototyping, rather than the final product, is counter-intuitive. As a result, prototyping usually needs to be done in a hurry, or it will not be done at all.

Since the goal of prototyping is learning, the key to rapid prototyping is a ruthless focus on what you are trying to learn, at the expense of everything else. This requires designers to maintain a clear mental separation between the essence of their experiment and the scaffolding necessary to complete it. If the goal of a prototyping effort is to validate some design concepts, those design concepts need to be represented faithfully, but anything else is scaffolding and should be implemented in the most expedient means available. If the only goal is clarification of requirements, the entire implementation is scaffolding, because the focus of the experiment is functionality, not how that functionality should be implemented.

Smalltalk is an ideal language for rapid prototyping due to its economy of expression and interpreted execution, which allows a rapid code/test cycle, along with its extensive available class libraries which assist with rapid creation of both design essence and scaffolding. This paper presents a small set of patterns that have proven to be useful in the rapid development of prototypes using Smalltalk. The primary goal of these patterns is simple expedience. Expedience can be achieved in multiple ways:

- Reducing the amount of code that needs to be typed.
- Automatic generation of code.
- Reducing the probability of coding errors, and the associated test/debug time.

- Reducing the time and delays associated with communication and coordination between team members in a multi-person prototyping effort.

We call these patterns "scaffolding patterns" because they are particularly well suited to the development of the scaffolding required for rapid prototyping. Some of the artifacts of these patterns may well be suited to production code. In highly dynamic systems such as reflective architectures [Foote88] or certain black-box frameworks [Yoder97] some of the patterns themselves are viable through production. Yet, in most software development, the negative impacts these patterns have on clarity and maintainability (described in the Consequences section for each pattern) begin to overshadow the benefits of expediency as the focus shifts from exploration to production.

Although the patterns presented in this paper can be applied to single-person or team prototyping projects, some of the patterns are most ideally suited to teams (indicated in the Context section for each pattern).

25.1.1 Patterns Summary

The patterns described in this paper include the following:

- **EXTENSIBLE ATTRIBUTES** adds attributes to an object without modifying the object's class.
- **ARTIFICIAL ACCESSORS** emulates accessor methods for an object's **EXTENSIBLE ATTRIBUTES**, without adding these methods to the object's class.
- **ARTIFICIAL DELEGATION** allows an object to delegate an operation to one of its attributes, without modifying the delegating object's class.
- **CACHED EXTENSIBILITY** automatically generates attributes, accessors, and/or delegation methods as they are used.
- **SELECTOR SYNTHESIS** provides an expedient means for a state-dependent object to dispatch to a specific handler method based on state and event.

The context and consequences sections of each describe when a particular pattern is appropriate and the conditions under which one pattern might lead to another.

Some of which led to systems that are in production today. The authors have also come across many other Smalltalk developers who have used one or more of these patterns in their prototyping efforts. However pointing to particular systems to direct one to these patterns is analogous to pointing to the Golden Gate Bridge to discover how one might use a ladder. When the system is done, the scaffolding has been removed, as are any real traces that it ever was present.

25.2 Pattern: EXTENSIBLE ATTRIBUTES

Context:

You are participating in a multi-person prototyping effort where each designer has been assigned an area of focus. You are introducing a class within a prototype, and you anticipate that other designers will want to add additional attributes to your class as the prototype grows.

Problem:

How can you minimize the effort required to add additional attributes to the class?

Forces:

- It is easy to add an attribute to a Smalltalk class, provided that you are allowed to change the class.
- Communication and coordination effort is required to change a class if you are not the class owner.
- Requesting a coworker to change a class is no problem as long as the coworker is available at the moment you need him/her, and you can access the changed class immediately once the change has been completed.
- Coworkers may be unavailable when you need a change, resulting in delays or time-wasting work-arounds.

Solution:

Add a dictionary attribute to your class that can be used to store additional attributes against symbol keys. Provide an accessor for the dictionary so that other classes can access additional attributes as follows:

anExtensibleObject attributesDictionary at: #attributeName put: value. or value := anExtensibleObject attributesDictionary at: #attributeName

Rationale:

This approach allows other prototypers to add and access additional attributes without needing to change your class definition. Note that this approach is only useful in cases where the class is required to carry the additional attributes about, but does not require additional methods, or if you are using a tool that allows you to partition additional methods cleanly (e.g. ENVY/Developer). Examples where this approach is especially applicable include information records (such as database records) and protocol messages. This is also very useful when using a persistence strategy such as GemStone where adding attributes in standard fashions while maintaining existing data can be relatively time-consuming.

Consequences:

It is easy to add additional attributes to an extensible class, but users of this class must access these attributes in a unique way (i.e. through the attributes dictionary). As a result, users need to know which attributes in a class are normal (i.e. accessed using normal accessors) and which attributes are extended. If an extended attribute is converted to a normal attribute, user's code must be changed. Additionally, the performance hit taken from accessing these attributes via a dictionary versus explicit instance variables can be perceptible in contexts where the attribute is accessed repeatedly. Since the context is one where expediency reigns over performance, this is mostly irrelevant until the context changes. Then traditional techniques for these attributes (e.g. explicit instance variables) may be used.

As a result of these consequences, it is recommended that this pattern be used in combination with either ARTIFICIAL ACCESSORS or CACHED EXTENSIBILITY.

Related Patterns:

This implementation of EXTENSIBLE ATTRIBUTES is identical to that of VARIABLE STATE [Beck96]. Kent Beck suggests VARIABLE STATE as a means to deal with classes that have instance variables "whose presence varies from instance to instance." EXTENSIBLE ATTRIBUTES is not focused on allowing instances of the same class to have different attributes (although clearly this can be supported), but rather the addition of attributes to all instances of a class without modifying the class itself.

25.3 Pattern: ARTIFICIAL ACCESSORS

Context:

You are participating in a multi-person prototyping effort where each designer has been assigned an area of focus, and are designing a class within a prototype and have applied EXTENSIBLE ATTRIBUTES so that other designers can dynamically add other attributes to your class as the prototype grows.

Problem:

How do you make it easier for other classes to access your extended attributes?

Forces:

- EXTENSIBLE ATTRIBUTES provide an expedient means to add attributes to another class.
- Code that accesses extended attributes can be messy due to the need to access them through the dictionary.
- If extended attributes are changed to normal attributes during the course of prototyping, code which accesses these attributes through the attribute dictionary will need to be changed.

Solution:

Simulate the presence of accessors for the extended attributes by overriding the `doesNotUnderstand:` method, and using the selector which was not understood (with the ":" removed in the case of a setter method) as the key into the attributes dictionary. Thus within the `doesNotUnderstand:` method,

`anExtendibleObject widgets: 4` will be converted to `self attributesDictionary at: #widgets put: 4` and `anExtendibleObject widgets` will be converted to `^self attributesDictionary at: #widgets`

Rationale:

This approach provides all of the advantages of EXTENSIBLE ATTRIBUTES, but uses the syntax of normal attributes, simplifying access to these attributes and hiding the fact that EXTENSIBLE ATTRIBUTES have been used. Over the course of the prototyping effort, the class owner can change extended attributes to normal attributes and provide normal accessors, without needing to modify the class's collaborators.

Consequences:

This approach effectively hides the distinction between normal and extended attributes. However, it is important to remember that the pseudo accessor methods do not actually exist, which can lead to confusion when you are trying to trace method calls or browse implementors. If it is desired to make these accessors visible and the environment is such that the addition of methods without explicit communication is permissible, consider using CACHED EXTENSIBILITY. Alternatively, consider also overriding the `respondsTo:` or `canUnderstand:` method of classes which apply this pattern to further hide the fact that methods don't exist for messages that the object can handle. Note however, that this is only useful when other classes (scaffolding or otherwise) exploit these rarely used features with respect to the class(es) that employ this pattern.

Note that the use of `doesNotUnderstand:` adds an additional performance hit to the already suspect EXTENSIBLE ATTRIBUTES that may sometimes be perceptible. Since the context is one where expediency reigns over performance, this is mostly irrelevant until the context changes. Then traditional techniques for these attributes (e.g., explicit instance variables) may be used. Alternatively, if the new context allows migration to CACHED EXTENSIBILITY this additional overhead goes away.

25.4 Pattern: ARTIFICIAL DELEGATION

Context:

You are participating in a multi-person prototyping effort where each designer has been assigned an area of focus, and are implementing a class that delegates specific operations to specific attributes. You anticipate that other designers will want to add additional attributes and delegated operations to your class as the prototype grows.

Problem:

How do you prepare for additional delegated operations with minimal effort?

Forces:

Addition of a delegated operation will typically require addition of a method to the delegator as follows:

```
anOperation
  ^self delegate anOperation.
```

- Smalltalk (out of the box) does not support multiple inheritance or other mechanisms that would easily get the effect of delegation.
- We are learning little from writing these simple methods, so they are basically scaffolding.
- Writing these methods are simple but take time to create and are prone to human error (typos, forgetting to return a value, etc.).
- The compiler is accessible, as are the attributes of a class and whether or not the attributes understand a particular message.
- Communication and coordination effort are required to change a class if the person needing the change is not the class owner.

Solution:

Override the `doesNotUnderstand:` method of the delegator class to iterate through its attributes looking for an attribute which supports the method selector which was not understood. The first attribute found which supports the method selector is assumed to be the delegate.

Rationale:

This approach allows delegated operations to be added to a class without needing to modify the class itself. In addition to its expediency with respect to normal attributes, this approach is particularly easy to use when `EXTENSIBLE ATTRIBUTES` has been applied as the delegator can simply iterate through the values in its attribute dictionary.

Consequences:

`ARTIFICIAL DELEGATION` is hidden in the `doesNotUnderstand:` method, which can lead to confusion when you are trying to trace method calls or browse implementors. If it is desired to make this delegation visible and the environment is such that the addition of methods without explicit communication is permissible, consider using `CACHED EXTENSIBILITY`. Alternatively, as with `ARTIFICIAL ACCESSORS`, consider also overriding the `respondsTo:` or `canUnderstand:` method of classes that apply this pattern to further hide the fact that methods don't exist for messages that the object can handle. Note however, that this is only

useful when other classes (scaffolding or otherwise) exploit these rarely used features with respect to the class(es) that employ this pattern.

Extensive use of ARTIFICIAL DELEGATION versus explicit delegation can add potentially noticeable performance hits due to the overhead of the `doesNotUnderstand:` mechanism. Again, we only use this pattern when expediency overrides these issues. If the context later allows migration to CACHED EXTENSIBILITY this additional overhead goes away.

Although it is usually not the case, ARTIFICIAL DELEGATION and ARTIFICIAL ACCESSORS can be used in the same class at the same time as long as some means is provided for the `doesNotUnderstand:` method to distinguish between delegated operations and ARTIFICIAL ACCESSORS. One way to do this would be to prefix all accessor methods with "get" or "set". The `doesNotUnderstand:` method could then assume that method selectors beginning with "get" or "set" are associated with ARTIFICIAL ACCESSORS, while all others are for ARTIFICIAL DELEGATION. Alternatively, one can look for delegators first, and treat not understood messages as an ARTIFICIAL ACCESSOR only as a last resort.

25.5 Pattern: CACHED EXTENSIBILITY

Context:

You are developing a prototype and have implemented ARTIFICIAL ACCESSORS, ARTIFICIAL DELEGATION, or other automatic functionality via overriding `doesNotUnderstand:`. As class owner, you would now like to identify how these facilities are being used to determine what the class is actually doing (as opposed to what methods you can see).

Problem:

How do you explicitly identify the implicit behavior defined by the actual use of a class or its instances and make it part of the explicit behavior of the class?

Forces:

Asking everyone (including yourself) what they've done to your class via "automatic facilities" is time-consuming and error-prone.

If you ignore the fact that people are using the facilities you've provided, you may miss out on some essential point about the class, defeating the purpose of prototyping.

Automatic facilities can introduce unnecessary performance problems, and it may be important to know whether perceived performance problems are due to these facilities or some fundamental flaw in the class.

Traditional debugging techniques (like Transcript show:) might offer some benefit, but will be difficult to trace and are only temporary, leaving no permanent record of the virtual methods.

Solution:

Again, override the `doesNotUnderstand:` method, substituting code to generate explicit methods for the virtual methods that are invoked the first time the implicit message is sent. As these facilities are used, the class will reveal how via the methods generated. One can then examine these methods and their senders to determine both what is happening and what could be done better.

Rationale:

Again, the time spent writing generic code to write common code patterns will quickly pay for itself. The quickly thrown up scaffolding which gives developers "automatic" functionality can be torn down just as quickly and replaced with new scaffolding which provides a different service.

Consequences:

At first glance, this pattern seems to overcome the negative consequences of patterns such as ARTIFICIAL ACCESSORS or ARTIFICIAL DELEGATION, by generating the necessary code as it is needed. However, it is important to note that this approach only generates code for methods that are executed, not for all methods that could potentially be executed. As such, the actual methods that are generated will be dependent on the extent to which the prototype is exercised during (usually informal) testing. Also in a configuration controlled environment, such as ENVY, determining the home of methods desired by non-owners may not always be trivial, and ARTIFICIAL ACCESSORS postpones the necessity of addressing this issue.

CACHED EXTENSIBILITY cannot be used simultaneously with ARTIFICIAL ACCESSORS or ARTIFICIAL DELEGATION because they handle the same doesNotUnderstand: conditions differently. However, they can be used sequentially, starting out with ARTIFICIAL ACCESSORS and ARTIFICIAL DELEGATION, then switching to CACHED EXTENSIBILITY (by modifying or replacing the doesNotUnderstand: method) when the owner is ready to transition to explicit code.

25.6 Pattern: SELECTOR SYNTHESIS

Context:

You are developing a prototype and are designing a class that needs to change its behavior (i.e. how it handles particular events) based on its state. You expect that additional states and events will be added to your class as the prototype grows.

Problem:

How can you implement a state-dependent object with minimal effort?

Forces:

- The STATE pattern [GOF94] provides an elegant solution for implementing state-dependent objects, but it involves defining a new class for each state, resulting in a lot of typing.
- In a prototyping effort, while the state-dependent objects overall behavior is typically important to the essence of the experiment, how this state dependence is implemented is unimportant (i.e. scaffolding).

Solution:

Define states and events as symbols. For a given event/state pair synthesize a method selector by concatenating the state and event symbols, then dispatch based on the resulting selector as follows:

```
selector := 'handle', anEvent asString, 'In', aState asString.  
self perform: selector asSymbol.
```

For each supported state/event combination provide an appropriately-named handler method. State/Event combinations that are not supported can be detected in the doesNotUnderstand: method and routed to a default handler.

Rationale:

Once SELECTOR SYNTHESIS has been implemented all it takes to implement new states, events or state/event combinations is to add appropriately-named handler methods. There is no additional dispatch code to write or classes to create.

Consequences:

State/Event handler methods are never explicitly called (i.e. browsing senders will come up empty). On the other hand figuring out where the calls are coming from shouldn't be a big problem as long as the prototyping team is familiar with the pattern.

The combination of the use of `doesNotUnderstand:` and the runtime creation of Symbols can sometimes cause a perceptible performance hit. As long as the identification of the state transitions are the focus versus the performance of those transitions, this is irrelevant.

SELECTOR SYNTHESIS can be used at the same time as ARTIFICIAL ACCESSORS and ARTIFICIAL DELEGATION as long as some means is provided for the `doesNotUnderstand:` method to distinguish unimplemented state/event handlers. One way to do this would be to assume that all state/event handler method names begin with "handle".

Related Patterns

A number of patterns, including STATE [GOF94] and STATE ACTION MAPPER [Palfinger97] also address the issue of state-dependent object behavior. SELECTOR SYNTHESIS has the advantage of expedience, but these other patterns may be better suited to production software.

25.7 Conclusions

The patterns in this paper all take advantage of capabilities inherent in the Smalltalk environment to facilitate rapid prototyping. There are a number of relationships between these patterns:

EXTENSIBLE ATTRIBUTES and ARTIFICIAL ACCESSORS should be used together.

ARTIFICIAL DELEGATION and SELECTOR SYNTHESIS are special-purpose patterns, applicable primarily to specific design situations (i.e. responsibility delegation, and state-dependent behaviour).

CACHED EXTENSIBILITY is essentially a "clean-up" pattern improving on ARTIFICIAL ACCESSORS, EXTENSIBLE ATTRIBUTES, and ARTIFICIAL DELEGATION by generating the necessary method code as it is used.

Scaffolding patterns, like scaffolding in a building, may be used for a while, then be torn down and thrown away. Notwithstanding, these patterns can play an important role in expediting prototype development, and it is in this way that they can make a meaningful contribution to the successful completion of software development projects.

25.8 Instructions

The goal of the exercise is to implement a trivial example for each of the scaffolding patterns. The example itself is not important, it can be as simple as you like. In the following, we provide some tips and examples (in the form of tests), but you are not forced to use them. If you think you have found a nicer example for a pattern, you can change the tests.

In the following, we use SUnit tests for describing the goal of the exercises. SUnit is a small framework that makes writing and running tests quite easy. The first Exercise will have a short introduction.

25.8.1 Interesting Classes and Methods

Have a look at the following methods. A solution for the exercises will at least need to use some of those.

- `Object>>instVarNamed:`
- `Object>>perform:`
- `Object>>perform:withArguments:`
- `Object>>respondsTo:`

- Object>>doesNotUnderstand:
- Behaviour>>canUnderstand:
- Behavior>>allInstVarNames
- ClassDescription>>compile:classified:

25.8.2 Exercise 0: SUnit

Goal: write and run a simple SUnit testcase. First, make a new test class, something like this:

```
TestCase subclass: #ScaffoldingsTest
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Scaffolding'
```

Then, add this method:

```
testMyFirstTest
  self assert: (1 $i$ 2).
  self deny: (2 $i$ 1).
  self shouldnt: [ 1 + 2 ] raise: MessageNotUnderstood.
```

Now open the TestRunner. Deselect all tests, select your testclass and press "run". The TestRunner should be green.

25.8.3 Exercise1: Implementation EXTENSIBLE ATTRIBUTES

As a warm-up, the first exercise is to implement an example for EXTENSIBLE ATTRIBUTES.

The following test should be green:

```
testExtensibleAttributes

| anExtensibleObject res |
anExtensibleObject := ExtensibleAttributes new.

anExtensibleObject attributesDictionary
  at: #attributeName
  put: 4.
res := anExtensibleObject attributesDictionary at: #attributeName.
self assert: res = 4.
```

25.8.4 Exercise2: Implementation ARTIFICIAL ACCESSORS

Extend the class you implemented in the preceding exercise to use ARTIFICIAL ACCESSORS.

A test for that would look like this:

```
testArtificialAccessors

| anExtensibleObject |
anExtensibleObject := ArtificialAccessors new.

self shouldnt: [anExtensibleObject hallo: 4]
  raise: MessageNotUnderstood.
```

```
self assert: anExtensibleObject hallo = 4.  
  
self should: [anExtensibleObject test]  
    raise: MessageNotUnderstood.  
self should: [anExtensibleObject test: 1 hallo: 2]  
    raise: MessageNotUnderstood.
```

25.8.5 Exercise3: Implementation ARTIFICIAL DELEGATION

The class `ArtificialDelegation` has two instance variables: one is a color, the other one a collection. After implementing `ARTIFICIAL DELEGATION`, a test like this one would pass:

```
testArtificialDelegation  
| o |  
o := ArtificialDelegation new.  
self deny: o isBlack.  
o add: 1.  
self assert: o last = 1.  
self deny: (o respondsTo: #isBlack).  
self deny: (o respondsTo: #last).  
self deny: (o respondsTo: #add:).
```

25.8.6 Exercise4: Implementation SELECTOR SYNTHESIS

This pattern is useful for modelling state. The number one example for a statefull object is, of course, the traffic light. So as an example for the pattern, you can build one class that models the state of a traffic light. The state is kept as a `Symbol` in an instance variable. Switching state is done by calling the `#handle:` method with the parameter

- `#Switch`: go to next state
- `#Stay`: don't change state

A test for this could look like:

```
testSelectorSynthesis  
| o |  
o := SelectorSynthesis new.  
o state: #Red.  
o handle: #Switch.  
self assert: o state = #Green.  
o handle: #Switch.  
self assert: o state = #Red.  
o handle: #Stay.  
self assert: o state = #Red.
```

25.8.7 Exercise5: Implementation CACHED EXTENSIBILITY

This exercise builds upon Exercise 3. Instead of just only delegating to the instance variables, this time we want to generate the delegation methods on the fly.

For that, you need to build the code of the delegation method as a `String` (or a `Stream`). Then compile and add it as a method to the class.

```
testCachedExtensibility
```

```
| o |
```

o := CachedExtensibility new.

self deny: (CachedExtensibility canUnderstand: #isBlack).

self deny: (CachedExtensibility canUnderstand: #add:).

self deny: (CachedExtensibility canUnderstand: #last).

self deny: o isBlack.

o add: 1.

self assert: o last = 1.

self assert: (CachedExtensibility canUnderstand: #isBlack).

self assert: (CachedExtensibility canUnderstand: #add:).

self assert: (CachedExtensibility canUnderstand: #last).

CachedExtensibility removeSelector: #isBlack.

CachedExtensibility removeSelector: #add:.

CachedExtensibility removeSelector: #last.

25.9 References

Joel77 Billy Joel, "The Stranger", (from the song, "Get It Right The First Time"), Columbia Records, 1977.

Foote88 Brian Foote, "Designing to Facilitate Change with Object-Oriented Frameworks", Masters Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1988 (<http://www.laputan.org/dfc/discussion.html>).

Yoder97 Joseph Yoder, "A Framework to build Financial Models"

Beck96 Kent Beck, "Smalltalk Best Practice Patterns", Prentice-Hall, 1996

Auer95 Ken Auer, "Reusability through Self Encapsulation", Chapter 27 in "Pattern Languages of Program Design", edited by Coplien & Schmidt, Addison-Wesley, 1995.

GOF94 E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994 (<http://hillside.net/patterns/books/#Gamma>).

Palfinger97 Guenther Palfinger, "State Action Mapper", PLoP97 Proceedings, Washington University Technical Report #wucs-97-34.

Generating Bytecodes

Main Author(s): Marcus Denker

Introduction

Generating bytecode can be done with *IRBuilder*. The following exercise uses the *IRBuilder* of Squeak 3.7.

Here is the the example from the lecture again:

```
irMethod:= IRBuilder new
  rargs: #(self);    "receiver and args"
  pushTemp: #self;
  send: #truncated;
  returnTop;
  ir.
```

```
aCompiledMethod := irMethod compiledMethod.
```

The variable *aCompiledMethod* now contains the generated compiled method. This method can be executed:

```
aCompiledMethod valueWithReceiver:3.5 arguments: #()
```

26.1 Expressions

With the help of *IRBuilder*, generate a method that calculates the expression `(3 + 4) factorial` and returns the result.

26.2 Parameters

Change the code that you wrote for the first exercise to use parameters instead of hard coded numbers. So in the end you will have a method that requires two arguments. If executed with

```
aCompiledMethod valueWithReceiver: nil arguments: #(3 4)
```

the result should be 7.

26.3 Loops

The Squeak bytecode has support for jumps. Jumps are used to implement conditionals and loops in an efficient way.

Generate a *compiledMethod* with *IRBuilder* that outputs the numbers 1 to 10 on the Transcript window.

26.4 Instance Variables

Generate a method that adds two instance variables and returns the result. Test the code by running it on a Point, e.g., 3@4.

26.5 Installing a Method in a Class

Find a way to add the method from Exercise 4 to the class Point with the name `returnSum`. After that, the following test should be green:

```
testReturnAdd
  self assert: (1@2) returnSum = 3.
  self assert: (3@4) returnSum = 7.
```

Bytecode Analysis

27.1 Counting Number of Executed Bytecodes

Look at the method tallyInstructions: in the class ContextPart (class-side):

"This method uses the simulator to count the number of occurrences of each of the Smalltalk instructions executed during evaluation of aBlock. Results appear in order of the byteCode set."

— tallies —

tallies := Bag new.

thisContext sender

runSimulated: aBlock

contextAtEachStep:

[:current — tallies add: current nextByte].

^tallies sortedElements

"ContextPart tallyInstructions: [3.14159 printString]"

^anArray at: 2

The method runSimulated: aBlock contextAtEachStep: [:current— ...] execute aBlock and for each bytecode executed in this block or in called methods, the second argument is evaluated with an instance of one of the subclasses of ContextPart as the argument.

Write a similar method named numberOfBytecodeExecuted: aBlock that returns the number of bytecode executed when evaluating the provided block. For instance:

ContextPart numberOfBytecodeExecuted: [3.14159 printString]

==> 1029

In total, the expression 3.14159 printString is evaluated by executing 1029 bytecodes.

27.2 Methods Coverage Analysis

Getting information about methods that are currently needed to perform a computation is often difficult to obtain with languages like Java. However this information can easily be retrieved in Smalltalk.

Number of Methods Invoked

Create a method numberOfInvokedMethods: aBlock that return the number of all the methods invoked when aBlock is evaluated.

ContextPart numberOfInvokedMethods: [3.14159 printString]

==> 38

Set of Methods Covered

We are now interested in the methods name.

```
ContextPart methodCovered: [3.14159 printString]
==> #('ContextPart class<<Dolt' 'Object<<printString'
      'Object<<printStringLimitedTo:' ... )
```

Bytecode Covered

Let's focus on bytecode. When a method is invoked, not all the bytecode contained in this method are executed. For instance, when executing 3.14159 printString the method on: defined in the class WriteStream is executed, but only 90% of its bytecode are executed.

```
ContextPart bytecodeCovered: [3.14159 printString]
==> #(#('WriteStream<<on:' 90) #('LimitedWriteStream<<nextPut:' 69)
      #('Object<<species' 100) ...)
```

Part VIII

SmallWiki

SmallWiki Introduction

28.1 Start Smallwiki

Exercise 131 Start the SmallWiki-Server on your machine (use the SmallWiki workspace) and explore all its possibilities from the user-point of view. If you are asked to login, give 'admin' as username and 'smallwiki' as password. Note: The authentication is still experimental and it is known not to work with the Mac OS-X Web browser *Safari*.

28.2 Play with Glossary

Exercise 132 Load the package *SmallWiki Glossary* from Store. To do this, go to the **Store** menu in the **VisualWorks** main window. Choose the **Published Items** menu item. Once the window with all the published items in Store appears, look for the package **SmallWiki Glossary** in the left pane of the window (you might have to scroll past the *bundles* that are listed first). Click in the package and now choose the last version of the package that appears in the right pane of the window. With the right-button of the mouse, choose the **Load** option. Once the process is completed, the package is loaded in your system.

Then go back to your browser and add a new glossary-component to the root of your wiki.

SmallWiki Glossary is a small component to create keyword-indexes of a whole wiki. In that package there are only 3 classes. Explore them and try to answer the following questions:

1. Glossary: What is the super-class of Glossary. Why is the instance side empty? What is done on the class-side? What would you have to do to implement a caching mechanism?
2. GlossaryView: What is the super-class of GlossaryView? What is the responsibility of GlossaryView? What piece of code would you have to change to reverse the sort-order of the keywords?
3. VisitorGlossary: From where is this visitor called? Why does the visitor override the message `#defaultCollection`. Why is the message `#acceptCode:` empty? Should this message be removed? What would you have to change, if you wanted to filter-out common words as *the*, *and*, *or*, *not* ?

28.3 Diving into SmallWiki Tests

Exercise 133 Go to the package *SmallWiki Tests* in the *SmallWiki* bundle. Select all the test-classes and hit on the run-button (assuming that *RBSUnitExtensions* has been loaded). Do you get errors?

1. Now have a look at *StructureTests*. First have a look at *StructureTests>>setUp*¹ and try to understand how a wiki is built from code.
2. Now go to the 'testing-testing' protocol. By looking at those tests only, guess what the messages *Structure>>isEmpty* and *Structure>>isRoot* are used for.

¹ the notation *AClass>>aMethodName* refers to the method named *aMethodName* defined in the class *AClass*. Method defined in a metaclass can be referenced such as *AClass class>>aMethodName*

3. Do the same in the 'testing-navigation' protocol for the messages `Structure>>next` and `Structure>>first`.

28.4 Understanding the Structure

Exercise 134 Now change to the package *SmallWiki Structure* and select the class `Structure`. What are its subclasses? Have you already seen those classes anywhere?

1. Have a look at the messages `Structure>>isEmpty`, `Structure>>isRoot`, `Structure>>next` and `Structure>>first`. Do they do the same as you thought while studying the tests?
2. Now go to the *serving* protocol. Have a close look at all the messages defined in there. What are they used for? What is the starting point?. Why is `Structure>>processChild` overridden by the `Folder` class? What is its default implementation?