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

October 4, 2005

Contents

Part I

First Contact

1

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 0 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 1 What kind of object does the literal expression 'Hello, Dave' describe?

Exercise 2 What kind of object does the literal expression #Node1 describe?

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

Exercise 4 What can one assume about a variable named Transcript?

Exercise 5 What can one assume about a variable named rectangle?

Exercise 6 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 7   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 8   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 9   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 10   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 11   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 12   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 13   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 14   Examine the block expression:

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

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

---

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

2

Counter Example

Main Author(s): Bergel, Ducasse, Wuyts

A Simple Counter

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

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

Creating your own class

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

When we are using Store, categories are replaced by packages. Therefore in VisualWorks with Store you define a package and define your class within this package. The steps we will do are the same ones every time you create a class, so memorize them well. We are going to create a class SimpleCounter in a package called DemoCounter. Figure 2.1 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.

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 2.1).

With the package selected, as shown in Figure 2.1, 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.

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 2.1)
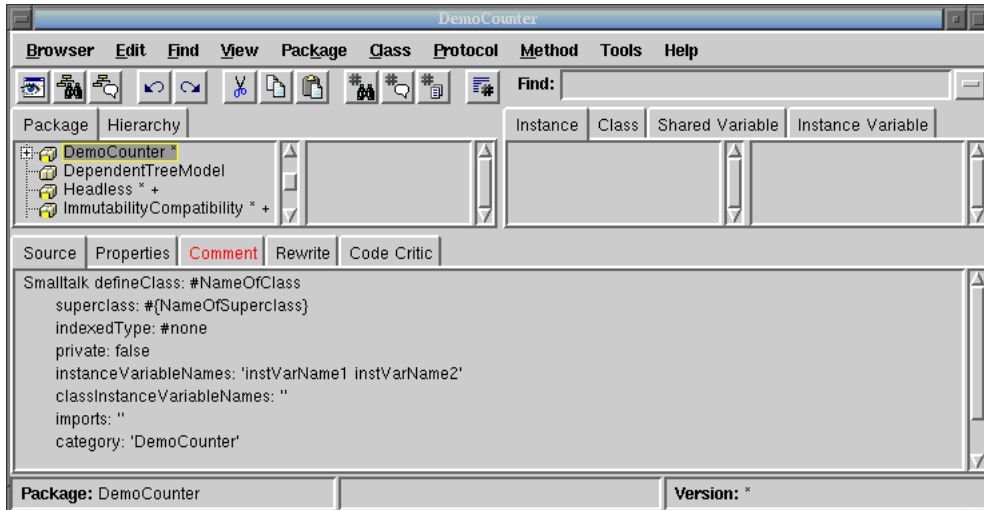
Figure 2.1: Your package is created.

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 2.1 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.

Defining protocols and methods

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

Creating and Testing Methods

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

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

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

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

Exercise 15 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 2.1, 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”

    ˆvalue

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 2.2).

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 16 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:
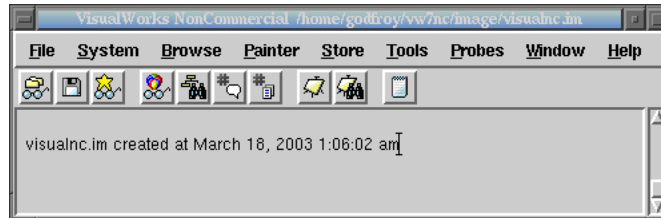
Figure 2.2: The Launcher of VisualWorks.

SimpleCounter new value: 7

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

Exercise 17   Implement the following methods in the protocol operations.

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

Exercise 18   Implement the following methods in the protocol printing

---

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

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

SimpleCounter new value: 0; increment ; value.

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

Adding an instance creation method

When we create a new instance of the class SimpleCounter using the message new, we would like to obtain a well initialized instance. To do so, we need to override the method new to add a call to an initialization method (invoking an initialize method is a very common practice! Ask for the senders of initialize). Notice that new is always sent to a class. This means that we have to define the new method on the *class side*, not on the *instance side*. To define an instance creation method like the method new you should be on the class side, so you click on the **Class** tab (See in the figure 2.1, the **Class** is situated in the same level as the **Instance** tab).

Exercise 19   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"
ˆsuper new initialize
```

Adding an instance initialization method

Now we have to write an initialization method that sets a default value to the value instance variable. However, as we mentioned the initialize message is sent to the newly created instance. This means that the initialize method should be defined at the instance side as any method that is sent to an instance of SimpleCounter like increment and decrement. The initialize method does not have specific and predefined semantics; it is just a convention to name the method that is responsible to set up the instance variable default values.

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

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

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

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

```
SimpleCounter new increment
```

Another instance creation method

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

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

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

Saving your Work

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

3

Set, Dictionary et Bag

Main Author(s): Bernard Pottier

3.1 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 **??**).

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')"
```

3.2 Set

3.2.1 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 connait 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.

3.2.2 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.

3.3 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 3.5).

- Possibilité d'utiliser les dictionnaires comme des tables de hachage, et pour des clés tout à fait quelconques.

- Dynamicité de la collection qui grandit avec les besoins.

3.3.1 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 chaine. On remplit le premier dictionnaire en notant les index des blancs et les index des ponctuations. Le dictionnaire apparait 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.

3.3.2 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 occurencesOf: 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. . .

3.3.3 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 chaine 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 eclats de glace de ton rire etoile'.
(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 )"
```

3.4   Bag

Cette classe s'appuie sur Dictionary ou IdentityDictionary pour compter le nombre d'occurences 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.

3.4.1 Ajouts et suppressions

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

- unSac add: unObjet
  Le compteur des occurences de unObjet est incrémenté.

- unSac remove: unObjet
  unSac remove: unObjet ifAbsent: aBlock
  Le compteur des occurences de unObjet est décrémenté. Si ce compteur tombe à 0, l'objet disparait 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 occurences du même objet, ou enlève toutes les occurences d'un objet.

3.4.2 Énumérations

- do: collect: inject: . . . : Le principe utilisé est d'évaluer les blocs paramêtres pour chaque élément de l'ensemble. Par exemple, si unObjet apparait 10 fois dans le sac, il y aura 10 évaluations du bloc paramêtre.

  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 chaine

```
| 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 paramêtres (élément et son compteur).

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

```
| stream sort |
stream := WriteStream on: (String new:200).
"preparation d'une collection comportant des Associations triees 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 - '"
```

3.5 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.

3.5.1 Boucle externe du test et formatage

On y reconnait un stream en écriture sur une chaine de caractères, qui sera utilisé pour produire une table au format LaTeX (voir la table 3.1).

Le bloc eval possède deux paramêtres, 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 Ordered-Collection 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
```

3.5.2 Boucle interne du test

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

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 abcisse 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. "Generateur 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
```

3.5.3 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 3.1: 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'apparait pas ici.

4

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.

4.1   Set

Exercise 20   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.

4.2   Dictionary

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

4.3   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

5

Some Useful Tools in Squeak

Main Author(s): Bergel, Denker, Ducasse

5.1   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 5.1: SqueakMap Package Loader on Shout
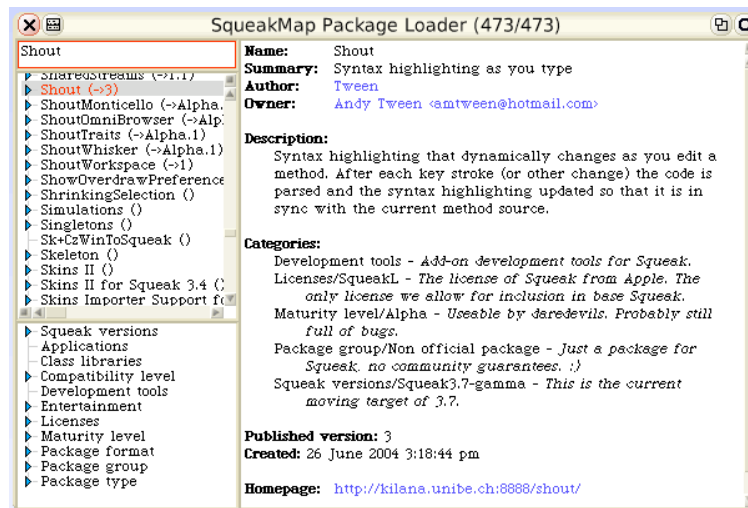
1. Monticello: Monticello is a package support for Squeak (normally already included in 3.7 full release).

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.

5.2 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 5.3, the method named stylerAbout-ToStyle: 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.
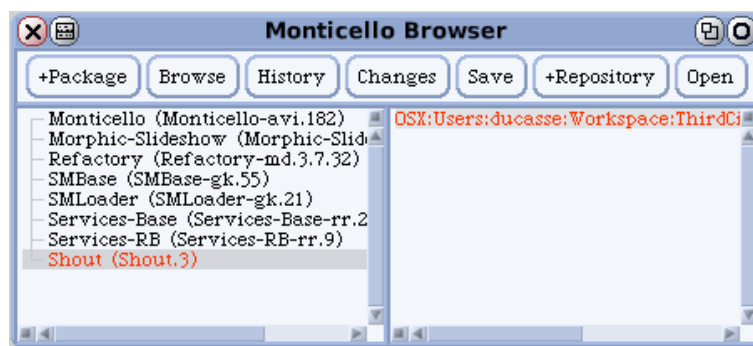
Figure 5.2: Monticello
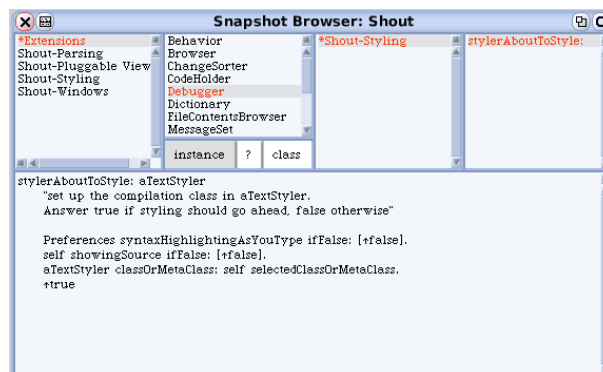
Figure 5.3: Browsing the changes associated to a package.

5.3 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 5.5

Figure 5.4: SqueakSource is a source forge like server for Squeak.

Figure 5.5: Adding a repository to your monticello repository list.

6
Monticello

6.1 Packages in Monticello: PackageInfo

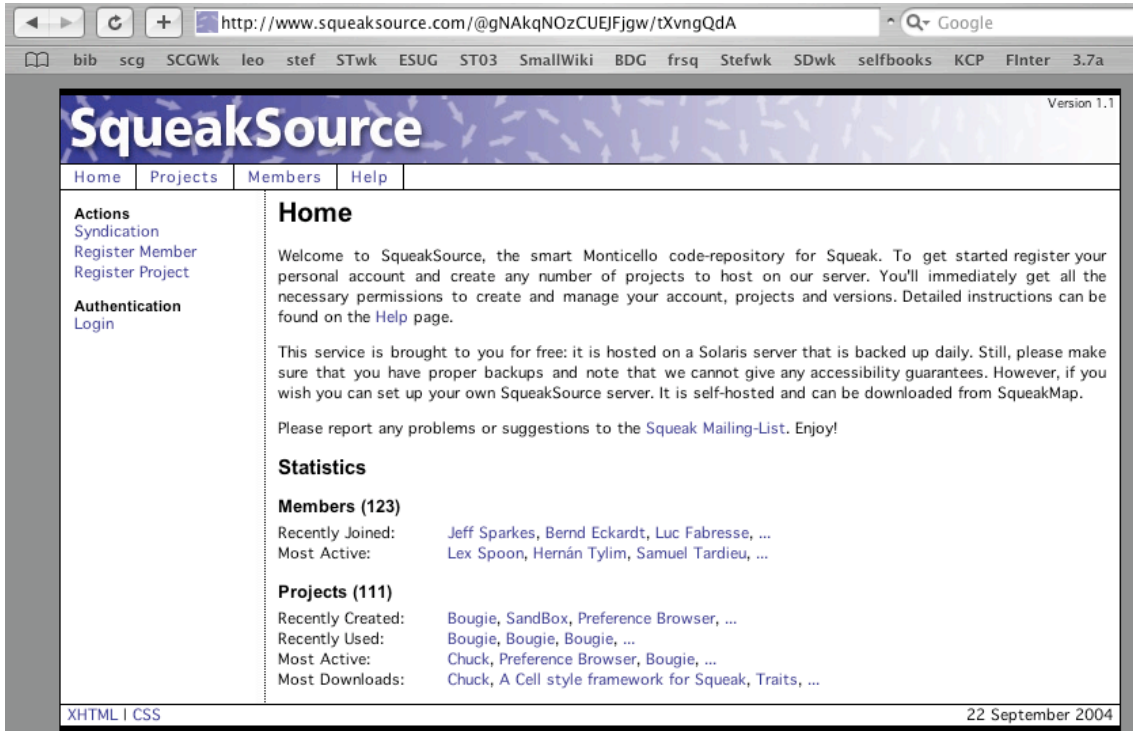The PackageInfo system is a simple, lightweight way of organizing Smalltalk source: it is nothing more than a naming convention, which uses (or abuses) the existing categorization mechanisms to group related code. Let me give you an example: say that you are developing a framework named SqueakLink to 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).

6.2 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 verison 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.

6.3   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.

6.4   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, wich 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 retreived. 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 a 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.

6.5   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 memebers 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 explaination:
    (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.

6.6   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 pacakge.

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 with 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. The is useful when you need to find a specific Version to load, merge, browse etc.

6.7   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 editiing.

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.

6.8 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 II

Seaside

7

Web dynamique avec Seaside

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

7.1   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: 'chaine de caracteres' affiche simplement la chane 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: ["definition 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. Reoit 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 clic sur un bouton du formulaire dans lequel se trouve la zone de saisie.

- submitButtonWitAction: ["traitements"] text: 'titre du bouton' ajoute un bouton qui a pour titre la chane de caractères passée comme deuxième argument. Un clic sur le bouton provoque l'exécution des traitements définis dans le bloc passé comme premier paramètre.

7.2   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),

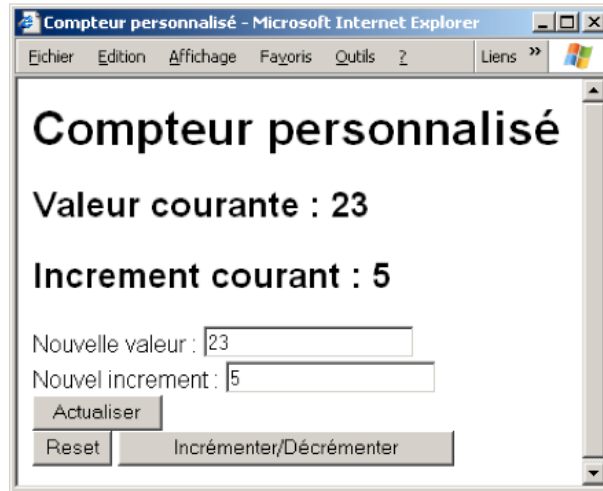- ainsi que la méthode de génération du code html (renderOn:).

Figure 7.1: L'interface du compteur personnalisé

L'interface utilisateur doit être analogue à celle de la figure 7.1. 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 increment 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.

7.3   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 7.2.

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.
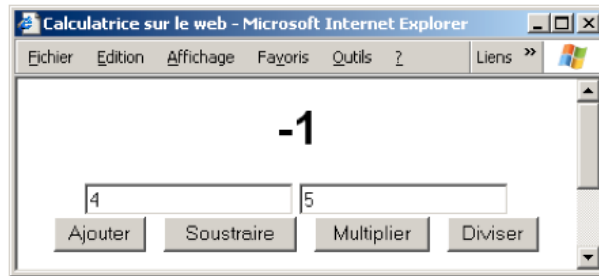
Figure 7.2: L'interface de la calculatrice.

7.4   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 identifiants 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.

8

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.

8.1 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.
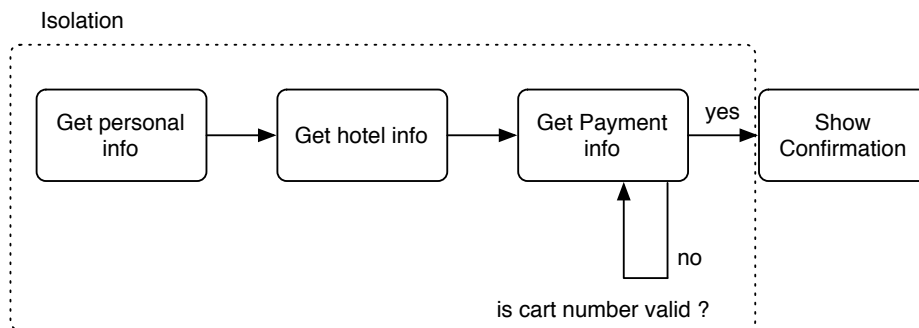

Figure 8.1:

8.2 Application Building Blocks

8.2.1 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.

8.2.2 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 WAComponent). 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:
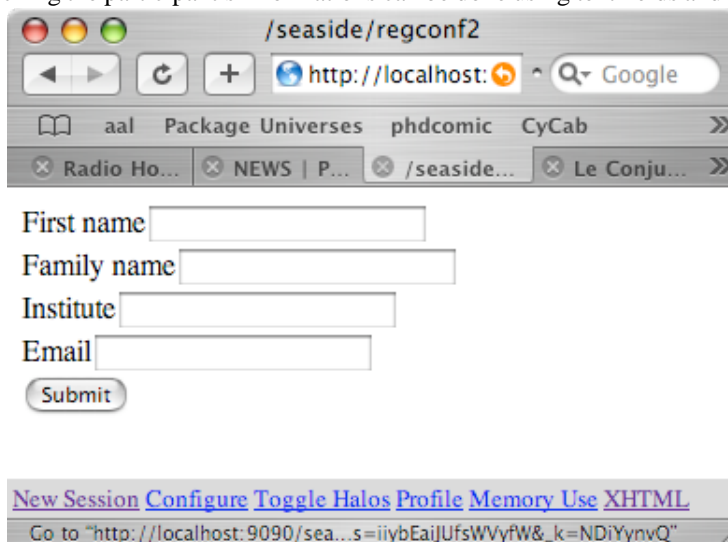
Your job: Write the method renderContentOn: in RCGetUserInfo.

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.

8.2.3 Getting Hotel Information: RCGetHotelInfo

A list of choices is pleasant to fetch informations of the hotel.
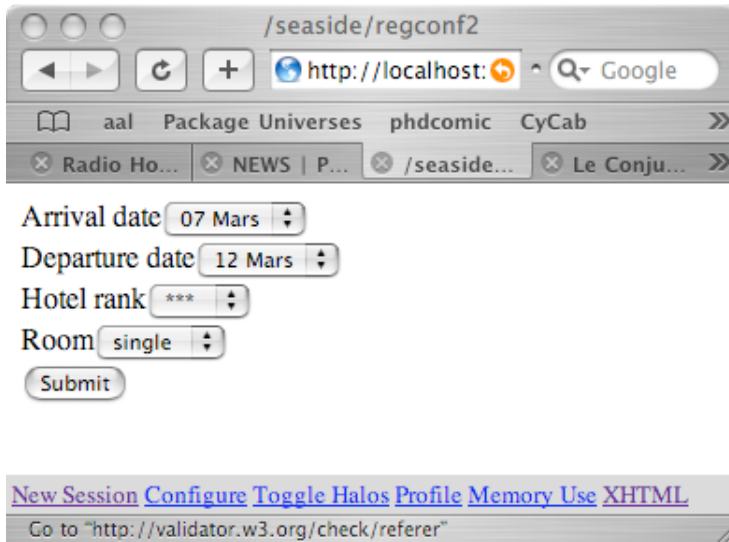
Your job: Write the class RCGetHotelInfo

8.2.4 Payment: **RCPayment**

The payment is valid only if 16 number was provided and if the issue date is not over.
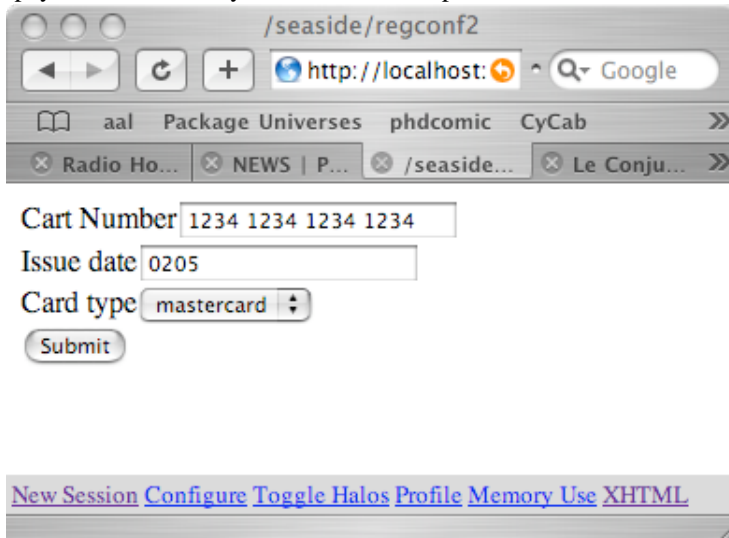
Your job: Write the class RCPayment

8.2.5 Confimation: **RCConfirmation**

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

Your job: Write the class RCConfirmation

8.3 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.

Part III

Object-Oriented Design

9

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

Exercise 22 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 23 Define a method called hasNextNode that returns whether the node has a next node or not.

Exercise 24 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

39

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 25 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 26 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 responsi-
bilities 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.
```
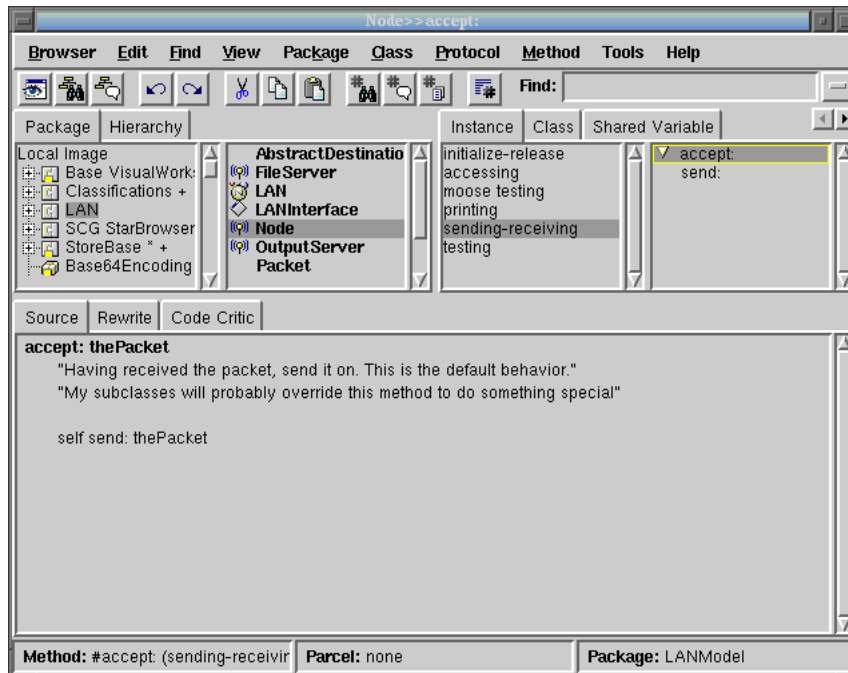
Figure 9.1: Definition of accept: method

Exercise 27   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 28   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 29   In the package LAN create a subclass of Node called Workstation without instance variables.

Exercise 30 Define the method `accept: aPacket` so that if the workstation is the destination of the packet, the following message is written into the Transcript. Note that if the packets are not addressed to the workstation they are sent to the next node of the current one.

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

A packet is accepted by the Workstation Mac

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

Exercise 31 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 32 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.

10

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.

10.1 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:.

10.2 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 whichor 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 33 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.

11

Object Responsibility and Better Encapsulation

11.1 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.

11.1.1 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.

11.1.2 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.

11.2 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 34 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 35 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 36 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 ourslef from possible unexpected extension we add yourself that returns the receiver a the first cascaded message (here name:), here the newly created instance.

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

---

11.3 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.

11.3.1 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.

11.3.2 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.

11.4 A Question of Creation Responsibility

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

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

- When possible, providing default values for instance variable is a good way to provide well-defined instances.

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

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

Packet class>>new

. . .

Packet>>initialize
 . . .

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

---

contents = 'no contents'

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

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

Instead of writing:

Packet>>hasDefaultContents

 ˆ 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

  ˆ 'no contents'

Packet>>initialize
  . . .

  contents := self defaultContent
  . . .

Packet>>hasDefaultContent
  ˆcontents = 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.

11.5   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.

11.6   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.

11.6.1 Remarks and Analysis.

A first solution could be the following code:

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

    ˆ 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

ˆ 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.

11.7 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

ˆ 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.
ˆ 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

 ˆ 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.

12

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.

12.1 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 a all the different kinds of nodes. The method simplePrintString, even if in our case is rather simple, is called a template method. This name reflects the fact that the method specifies the context in which hook methods will be called and how they will fit into the template method to produce the expected result.

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

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

Exercise 37   Study the method printOn: on the class Object. Check its implementors and senders.

Exercise 38   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.

missing compiler td of bernard, actalk, objVlisp implementations