# Advanced Object Oriented Design with Smalltalk

Gabriela Arévalo
Alexandre Bergel
Prof. Dr. Stéphane Ducasse
Michele Lanza
Dr. Roel Wuyts
Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland
{arevalo, bergel, ducasse, lanza, wuyts}@iam.unibe.ch
http://www.iam.unibe.ch/∼scg/Teaching/Smalltalk03/

June 19, 2015

## Useful information

You can find all the information relative to the lecture at University of Bern at http://www.iam.unibe.ch/

**About me**

```
Prof. Dr. Stphane Ducasse
Room 101
Email:  ducasse@iam.unibe.ch
WWW:       http://www.iam.unibe.ch/~ducasse/
Freeonline books:
www.iam.unibe.ch/~ducasse/WebPages/FreeBooks.html
```

**Where to get VisualWorks?** You have received a CD that contains normally everything you need: all the free Smalltalk implementations currently available. If you have any kind of problem for installing VisualWorks, just ask and do not wait not the end of the lecture for that. On solaris define the variable $VisualWorks to refer to the directory of installation.

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

```
3 + 4
```

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

```
Date today
```

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

```
anArray at: 1 put: 'hello'
```

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

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

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

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

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

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

**Exercise:** Examine the following expression:

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

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

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

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

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

**Exercise:** Guess what are the results of the following expressions

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

**Exercise:** Examine the following expression:

```
25@50
```

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

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

```
Date today daysInMonth
```

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

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

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

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

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

**Exercise:** Examine the block expression:

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

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

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

# Exercise 2

# Counter Example

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

## A Simple Counter

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

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

## Creating your own class

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

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

### Creating a Package

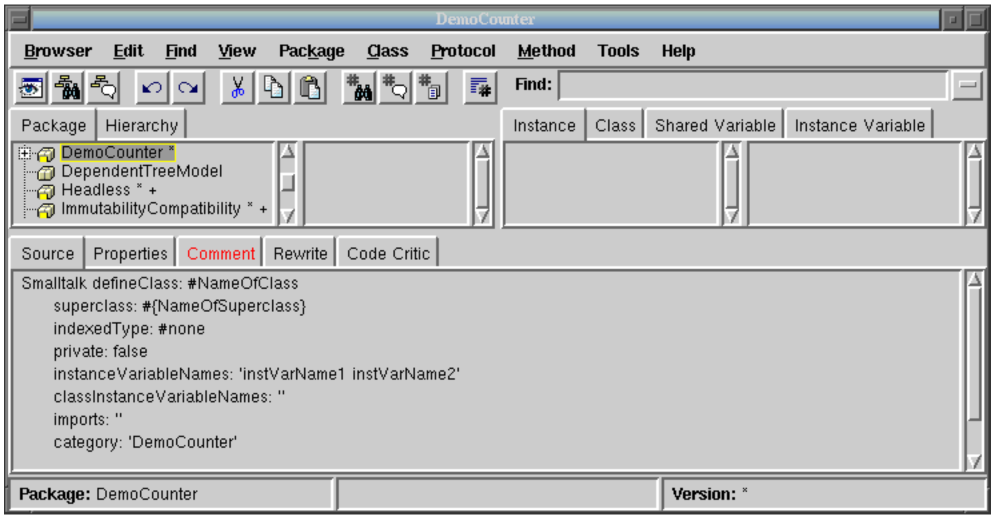
In the System Browser, click on the line `Local Image` located in the left-most upper pane (left button of the mouse) and select `New>>Package`. The system will ask you a name. You should write `DemoCounter`, prefixed with your initials or group name. This new package will be created and added to the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

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

## With Namespace

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

## Creating a Class


DemoCounter
Browser Edit Find View Package Class Protocol Method Tools Help
Find:
Package Hierarchy
Instance Class Shared Variable Instance Variable
DemoCounter *
DependentTreeModel
Headless * +
ImmutabilityCompatibility * +
Source Properties Comment Rewrite Code Critic
Package: DemoCounter
Version: *

```
Smalltalk defineClass: #NameOfClass
      superclass: #{NameOfSuperclass}
      indexedType: #none
      private: false
      instanceVariableNames: 'instVarName1 instVarName2'
      classInstanceVariableNames: ''
      imports: ''
      category: 'DemoCounter'
```

Figure 2.1: Your package is created.

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

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

2. **Class Name**. Next, you should fill in the name of your class by replacing the word `NameOfClass` with the word `SimpleCounter`. Take care that the name of the class starts with a capital letter and that you do not remove the # sign in front of `NameOfClass`.

3. **Instance Variable Specification**. Then, you should fill in the names of the instance variables of this class. We need one instance variable called `value`. You add it by replacing the words *instVarName1* and *instVarName2* with the word `value`.Take care that you leave the string quotes!

4. **Class Variable Specification**. As we do not need any class variable make sure that the argument for the class instance variables is an empty string (`classInstanceVariableNames: ''`).

5. **Compilation**. That's it! We now have a filled-in class definition for the class `SimpleCounter`. To define it, we still have to **compile** it. Therefore, select the **accept** option from the operate menu (right-click button of the mouse). The class `SimpleCounter` is now compiled and immediately added to the system.

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

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

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

Instance Variables:

value        <Integer>
```

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

## Defining protocols and methods

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

### Creating and Testing methods

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

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

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

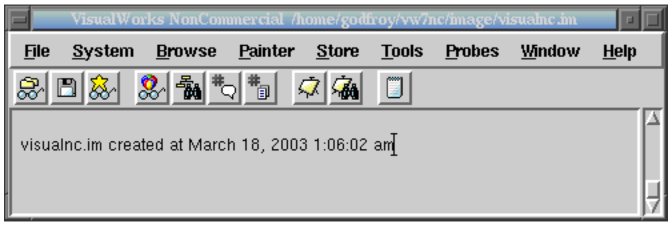
**An important remark:** *Accessors* can be defined in protocols `accessing` or `private`. Use the `accessing` protocol when a client object (like an interface) really needs to access your data. Use `private` to clearly state that no client should use the accessor. This is purely a convention. There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

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

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

  ^value
```

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

To use a workspace, click on the last icon of the launcher as shown in Figure **??**.


VisualWorks NonCommercial /home/godfroy/vw7nc/image/visualnc.im
File System Browse Painter Store Tools Probes Window Help

visualnc.im created at March 18, 2003 1:06:02 am

Figure 2.2: The Launcher of VisualWorks.

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

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

```
SimpleCounter new value: 7
```

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

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

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

**Exercise:** Implement the following methods in the protocol `printing`

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

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

```
SimpleCounter new value: 0; increment ; value.
```

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

## Adding an instance creation method

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

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

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

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

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

```
new
   "Create and return an initialized instance of SimpleCounter"

   ^ super new initialize
```

## Adding an instance initialization method

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

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

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

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

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

```
SimpleCounter new increment
```

### Another instance creation method

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

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

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

## SUnit

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

```
TestRunner open
```

## Saving your Work

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

# Exercise 3

# A Simple Application: A LAN simulation

## Basic LAN Application

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

### Creating the Class `Node`

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

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

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

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

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

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

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

Node named: Node1 connected to: PC1
```

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

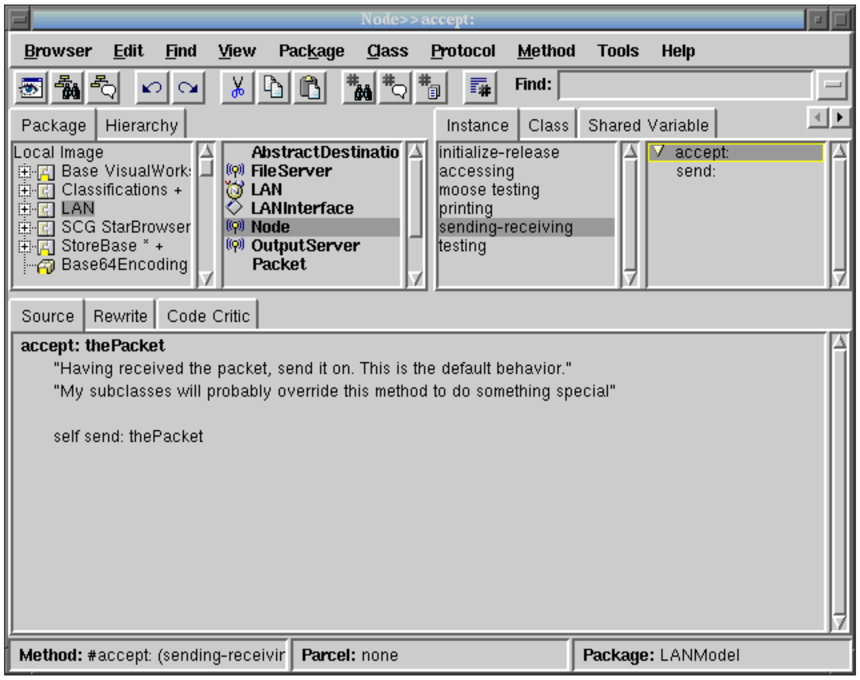
**Exercise:** A node has two basic messages to send and receive packets. When a packet is sent to a node, the node has to accept the packet, and send it on. Note that with this simple behavior the packet can loop infinitely in the LAN. We will propose some solutions to this issue later. To implement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

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

    ...

send: aPacket
"Precondition: self have a nextNode"

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

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

## Creating the Class `Packet`

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

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



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

	self send: thePacket
```

**Method:** #accept: (sending-receivir **Parcel:** none **Package:** LANModel

Figure 3.1: Definition of `accept:` method

---

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

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

## Creating the Class `Workstation`

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

---

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

---

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

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

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

A packet is accepted by the Workstation Mac
```

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

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

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

## Creating the class `LANPrinter`

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

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

## Simulating the LAN

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

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

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

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

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

mac originate: packet.
```

---

**anotherSimpleLan**

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

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

pc := Workstation new name:#pc.

node1 := Node new name: #node1.

node2 := Node new name: #node2.

node3 := Node new name: #node3.

igPrinter := LanPrinter new name: #IGPrinter.

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

node1 nextNode: node2.

node2 nextNode:igPrinter.
```

```
igPrinter nextNode: node3.

node3 nextNode: pc.

pc nextNode: mac.

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

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

## Creating the Class `FileServer`

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

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