Exercise 1

A Simple Application: A LAN simulation

Basic LAN Application

The purpose of this exercise is to create a basis on 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 **LAN** network. 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 basic behavior for all nodes. As a network is defined as basically a linked list of nodes, a Node should 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 treat it.
Per default sends it to the following node.
send: aPacket - sends a packet to the following 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.

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 exercise, 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 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).

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:

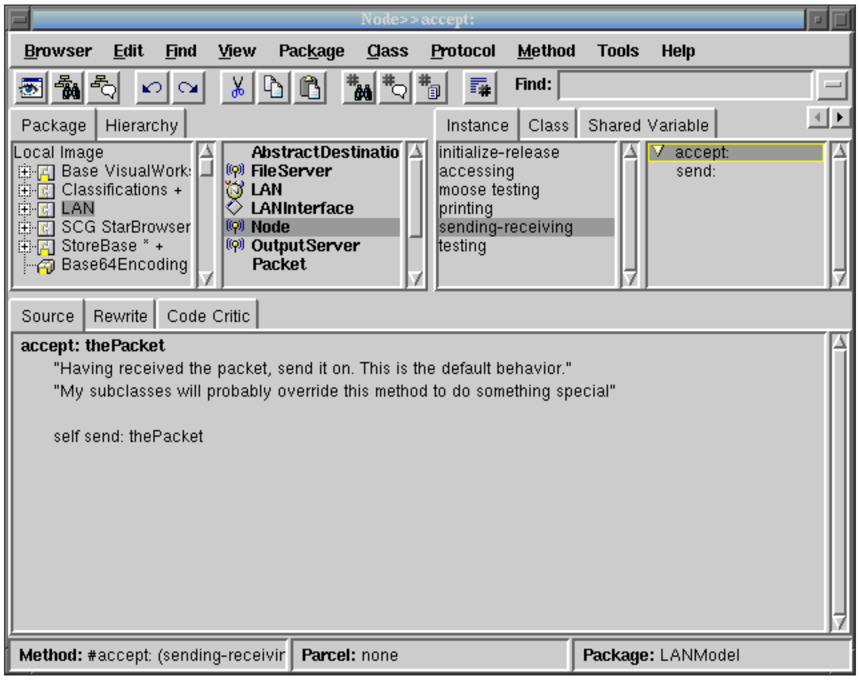
```
accept: thePacket
 "Having received the packet, send it on. This is the default
behavior My subclasses will probably override me to do
something special"

    ...

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 in 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.
```

Exercise: In the package 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



```
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 1.1: nlesson3Fig1.png about here.

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

```
Workstation inherits from Node
Collaborators: Node, Workstation
and Packet
Responsibility: (the ones of node)
originate: aPacket - sends a packet.
accept: aPacket - does some actions 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 acceptation of packet addressed to other node, you could copy and paste the code of the `Node` class. However this is a bad practice, decreasing the reuse of code and the "Say it only once" rules. It is better to invoke the default code that is currently overriden by using `super`.

Exercise: 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 create a class `LanPrinter` here, a special node that receive packets addressed to it and print them (on the Transcript). Note that we name it like this because Printer already exists in the system. Write this class.

```
LanPrinter inherits from Node
Collaborators: Node and Packet
Responsibility:
accept: aPacket - if the packet is addressed to the
printer, prints the packet contents else sends the packet
to the following node.
print: aPacket - prints the contents of the packet
(into the Transcript 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 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, we will propose a solution to this problem in the future. To break the loop, use or **Ctrl-Y** depending of the VisualWorks versions.

Creating of 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.
```