4 A Simple Application: A LAN simulation

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, Logger`, and `Printer`. 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.

Use a fresh pharo1.0-10451-BETAweb09.09.3 image to answer all questions.

4.1 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 1 Create a new category `LAN`, and create a subclass of `Object` called `Node`, with two instance variables: `name` and `nextNode`.

Exercise 2 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.

Exercise 3 Define a method called `hasNextNode` that returns whether the node has a next node or not.

Exercise 4 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 test:

```
testPrintOn
| printed |
```

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

self assert: printed = 'Node named: Node1 connected to: PC1'.
```

Exercise 5 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.
...
```

4.2 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.
```

Exercise 6 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.

Exercise 7 Write a test for the method `printOn: aStream` that puts a textual representation of a packet on its argument `aStream`. Now implement the method `printOn:` to make that test green.

4.3 Creating the Class `Workstation`

A workstation is the entry point for new packets onto the LAN network. It can originate packet to other workstations, printers 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 8 In the category `LAN` create a subclass of `Node` called `Workstation` without instance variables.

Exercise 9 Define the method `accept: aPacket` so that if the workstation is the destination of the packet, a 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. If you evaluate this code in the workspace:

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

The following method should be printed to the transcript:

```
A packet is accepted by the Workstation Mac
```

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

Exercise 10 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."
 ...
```

4.4 Creating output Nodes

Exercise 11 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 traveling around the LAN. Therefore, you will now create a class `Printer`, a special node that receives packets addressed to it and prints them (on the Transcript). Implement the class Printer.

Exercise 12 For testing, it would be nice to have a node that saves a log of all packages it forwards and those that where addressed to it. Implement this Logger. Provide an interface so that it is easy to use for SUnit testing the LAN.

4.5 Simulating the LAN

Exercise 13 Now write a method that sets up a simple LAN with the following configuration:

```
mac -> node1 -> node2 ->
logger -> node3 -> pc -> printer -> mac
```

- Write a test where a packet is send from mac that is addressed to the printer. Make sure to test that it passed the logger.
- Write a test where the packet is adressed to the logger.

4.6 Handling Loops

When a packet is sent to an unknown node (try it, but save the image before...) it loops endlessly around the LAN. You will implement two solutions for this problem.

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

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

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

1. Each node keeps track of the packets it already received. When a packet already received is asked to be accepted again by the node, the packet is not sent again in the LAN. This solution implies that packet can be uniquely identified. Their current representation does not allow that. We could imagine to tag the packet with a unique generated identifier. Moreover, each node would have to remember the identity of all the packets and there is no simple way to know when the identity of treated node can be removed from the nodes.

2. Each packet keeps track of the node it visited. Every time a packet arrived at a node, it is asked if it has already been here. This solution implies a modification of the communication between the nodes and the packet: the node must ask the status of the packet. This solution allows the construction of different packet semantics (one could imagine that packets are broadcasted to all the nodes, or have to be accepted twice). Moreover once a packet is accepted, the references to the visited nodes are simply destroyed with the packet so there is no need to propagate this information among the nodes.

Exercise 14 Please implement Solution 1. Provide an SUnit test.

Exercise 15 From Solution 2, please implement the second version (where the packet keeps track of the nodes it visited). Provide an SUnit test.

Please save the Monticello package LAN and send it by mail to st-staff@iam.unibe.ch. Attach your written solutions that are not part of the source-code to this mail and hand them in as hardcopy at the beginning of the next exercise session. Your mail and solutions should be clearly marked with names and matrikel numbers of the solution authors.