Learning Object-Oriented Programming, Design and TDD with Pharo

Stéphane Ducasse

March 11, 2019

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

A simple network simulator

In this chapter, we develop a simulator for a computer network, step by step from scratch. The program starts with a simplistic model of a computer network, made of objects that represent different parts of a local network such as packets, nodes, workstations, routers and hubs.

At first, we will just simulate the different steps of packet delivery and have fun with the system. In a second step we will extend the basic functionalities by adding extensions such as a hub and different packet routing strategies. Doing so, we will revisit many object-oriented concepts such as polymorphism, encapsulation, hooks and templates. Finally this system could be refined to become an experiment platform to explore and understand distributed algorithms.

Basic definitions and a starting point

We need to establish the basic model; what does the description above tell us? A network is a number of interconnected nodes, which exchange data packets. We will therefore probably need to model the nodes, the connection links, and the packets:

- Nodes have addresses, can send and receive packets;
- Links connect two nodes together, and transmit packets between them;
- A packet transports a payload and has the address of the node to which it should be delivered; if we want nodes to be able to answer (after reception), packets should also have the address of the node which originally sent it.

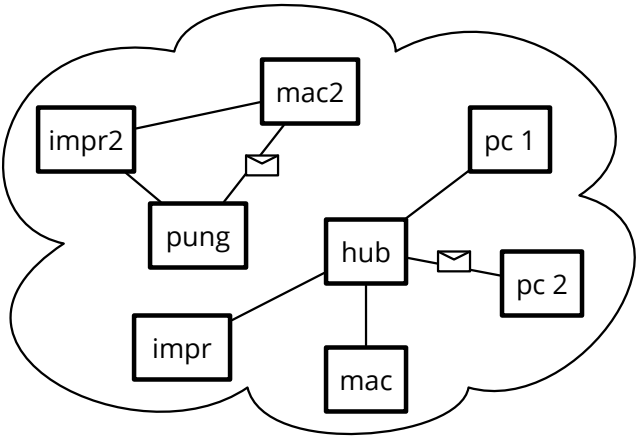

Figure 1-1 Two little networks composed of nodes and sending packets over links.

1.1 Packets are simple value objects

Packets seem to be the simplest objects in our model: we need to create them, and ask them about the data they contain, and that's about it. Once created, a packet object is merely a passive data structure: it will not change its data, knows nothing of the surrounding network, and has no behavior that we can really talk about.

Let's start by defining a test class and a first test sketching what creating and looking at packets would look like:

```
TestCase subclass: #KANetworkEntitiesTest
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'NetworkSimulator-Tests'
```

```
KANetworkEntitiesTest >> testPacketCreation
    | src dest payload packet |
    src := Object new.
    dest := Object new.
    payload := Object new.

    packet := KANetworkPacket from: src to: dest payload: payload.

    self assert: packet sourceAddress equals: src.
    self assert: packet destinationAddress equals: dest.
    self assert: packet payload equals: payload
```

By writing this unit test, we described how we think packets should be created, using a `from:to:payload:` constructor message, and how it should be

accessed, using three messages `sourceAddress`, `destinationAddress`, and `payload`. Since we have not yet decided what addresses and payloads should look like, we just pass arbitrary objects as parameters; all that matters is that when we ask the packet, it returns the correct object back.

Of course, if we now compile and run this test method, it will fail, because the class `KANetworkPacket` has not been created yet, nor any of the four above messages. You can either execute and let the system prompt you when needed or we can define the class:

```
Object subclass: #KANetworkPacket
    instanceVariableNames: 'sourceAddress destinationAddress payload'
    classVariableNames: ''
    category: 'NetworkSimulator-Core'
```

The class-side constructor method creates an instance, which it returns after sending it an initialization message; nothing original as far as constructors go:

```
KANetworkPacket class >> from: sourceAddress to: destinationAddress
    payload: anObject
    ... Your code ...
```

That constructor will need to pass the initialization parameters to the new instance. It's preferable to define a single initialization method that takes all needed parameters at once, since it is only supposed to be called when creating packets and should not be confused with a setter:

```
KANetworkPacket >> initializeSource: source destination: destination
    payload: anObject
    ... Your code ...
```

Once a packet is created, all we need to do with it is to obtain its payload, or the addresses of its source or destination nodes. Define the following getters:

```
KANetworkPacket >> sourceAddress
  ... Your code ...
KANetworkPacket >> destinationAddress
  ... Your code ...
KANetworkPacket >> payload
  ... Your code ...
```

Now our test should be running and passing. That's enough for our admittedly simplistic model of packets; we completely ignore the layers of the OSI model, but it could be an interesting exercise to model them more precisely.

1.2 Nodes are known by their address

The first obvious thing we can say about a network node is that if we want to be able to send packets to it, then it should have an address; let's translate

that into a test:

```
KANetworkEntitiesTest >> testNodeCreation
    | address node |
    address := Object new.
    node := KANetworkNode withAddress: address.
    self assert: node address equals: address
```

Like before, to run this test to completion, we will have to define the `KANetworkNode` class:

```
Object subclass: #KANetworkNode
    instanceVariableNames: 'address'
    classVariableNames: ''
    category: 'NetworkSimulator-Core'
```

Then a class-side constructor method taking the address of the new node as parameter:

```
KANetworkNode class >> withAddress: aNetworkAddress
    ^ self new
        initializeAddress: aNetworkAddress;
        yourself
```

The constructor relies on an instance-side initialization method, and the test asserts that the `address` accessor works; define them:

```
KANetworkNode >> initializeAddress: aNetworkAddress
  ... Your code ...
KANetworkNode >> address
  ... Your code ...
```

Again, our simplistic tests should now pass.

1.3 Links are one-way connections between nodes

After nodes and packets, what about looking at links? In the real world, network cables are bidirectional, but that's because they have wires going both ways. Here, we're going to keep it simple and define links as simple one-way connections; to make a two-way connection, we will just use two links, one in each direction.

However, creating links that know their source and destination nodes is not sufficient: *nodes* also need to know about their outgoing links, otherwise they cannot send packets. Let us write a test to cover this.

```
KANetworkEntitiesTest >> testNodeLinking
    | node1 node2 link |
    node1 := KANetworkNode withAddress: #address1.
    node2 := KANetworkNode withAddress: #address2.
    link := KANetworkLink from: node1 to: node2.
```

```
    link attach.

    self assert: (node1 hasLinkTo: node2)
```

This test creates two nodes and a link; after telling the link to *attach* itself, we check that it did so: the source node should confirm that it has an outgoing link to the destination node. Note that the constructor could have registered the link with `node1`, but we opted for a separate message `attach` instead, because it's bad form to have a constructor change other objets; this way we can build links between arbitrary nodes and still have control of when the connection really becomes part of the network model. For symmetry, we could have specified that `node2` has an incoming link from `node1`, but that ends up not being necessary, so we leave that out for now.

Again, we need to define the class of links:

```
Object subclass: #KANetworkLink
    instanceVariableNames: 'source destination'
    classVariableNames: ''
    category: 'NetworkSimulator-Core'
```

A constructor that passes the two required parameters to an instance-side initialization message:

```
KANetworkLink class >> from: sourceNode to: destinationNode
    ^ self new
        initializeFrom: sourceNode to: destinationNode
```

As well as the initialization method and accessors:

```
KANetworkLink >> initializeFrom: sourceNode to: destinationNode
  ... Your code ...
KANetworkLink >> source
  ... Your code ...
KANetworkLink >> destination
  ... Your code ...
```

The `attach` method of a link should not (and cannot) directly modify the source node, so it must delegate to it instead.

```
KANetworkLink >> attach
    source attach: self
```

This is an example of separation of concerns: the link knows which node has to do what, but only the node itself knows precisely how to do that. Here, if a node knows about all its outgoing links, it means it has a collection of those, and attaching a link adds it to that collection:

```
KANetworkNode >> attach: anOutgoingLink
    outgoingLinks add: anOutgoingLink
```

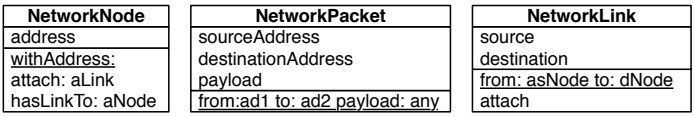

Figure 1-2 Current API of our three main classes.

For this method to compile correctly, we will need to extend `KANetworkNode` with the new instance variable `outgoingLinks`, and with the corresponding initialization code:

```
KANetworkNode >> initialize
    outgoingLinks := Set new.
```

And finally the unit test relied on a predicate method to define in `KANetworkNode`:

```
KANetworkNode >> hasLinkTo: anotherNode
    ... Your code ...
```

The method `hasLinkTo:` should verify that there is at least one outgoing links whose destination is the node passed as argument. We suggest to have a look at the iterator `anySatisfy:` to express this logic.

Again, all the tests should now pass.

1.4 Making our objects more understandable

When programming we often make mistakes and it is important to help developer to address them. Let us put a breakpoint and try to understand the objects.

```
KANetworkEntitiesTest >> testNodeLinking
  | node1 node2 link |
  node1 := KANetworkNode withAddress: #address1.
  node2 := KANetworkNode withAddress: #address2.
  link := KANetworkLink from: node1 to: node2.
  link attach.
  self halt.
  self assert: (node1 hasLinkTo: node2)
```

Running the test will open a debugger as the one shown in Figure 1-3. We get object but their textual representation is too generic to really help us.

The method `printOn:` is responsible to the printing of the object representation. We will then redefine this method for the different objects we have.

```
KANetworkNode >> printOn: aStream
  aStream nextPutAll: 'Node ('.
  aStream nextPutAll: address , ')'
```

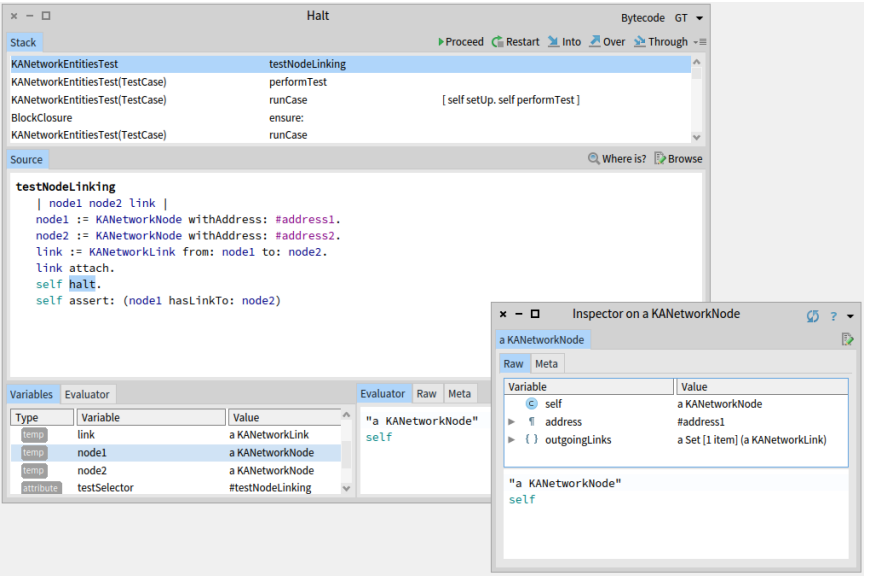

Figure 1-3 Navigating specific objects having a generic presentation.

```
KANetworkLink >> printOn: aStream
  aStream nextPutAll: 'Link'.
  source
    ifNotNil: [ aStream
        nextPutAll: ' ';
        nextPutAll: source address ].
  destination
    ifNotNil: [ aStream
        nextPutAll: ' -> ';
        nextPutAll: destination address ]
```

Now if we rerun the test we obtain a better user experience as shown in Figure 1-4: we can see the address of a node and the source and destination of a link.

1.5 Simulating the steps of packet delivery

The next big feature is that nodes should be able to send and receive packets, and links to transmit them.

```
KANetworkEntitiesTest >> testSendAndTransmit
    | srcNode destNode link packet |
    srcNode := KANetworkNode withAddress: #src.
    destNode := KANetworkNode withAddress: #dest.
```

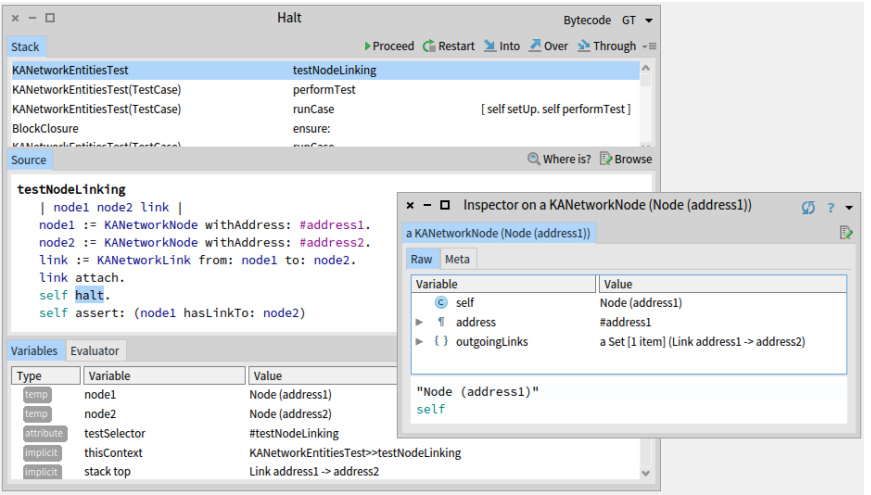

Figure 1-4 Navigating objects offering a customized presentation.

```
    link := (KANetworkLink from: srcNode to: destNode) attach;
    yourself.
    packet := KANetworkPacket from: #address to: #dest payload:
    #payload.

    srcNode send: packet via: link.
    self assert: (link isTransmitting: packet).
    self deny: (destNode hasReceived: packet).

    link transmit: packet.
    self deny: (link isTransmitting: packet).
    self assert: (destNode hasReceived: packet)
```

We create and setup two nodes, a link between them, and a packet. Now, to control which packets get delivered in which order, we specify that it happens in separate, controlled steps. This will allow us to model packet delivery precisely, to simulate latency, out-of-order reception, etc.:

- First, we tell the node to send the packet using the message `send:via:`. At that point, the packet should be passed to the link for transmission, but not completely delivered yet.
- Then, we tell the link to actually transmit the packet along using the message `transmit:`, and thus the packet should be received by the destination node.

1.6 Sending a packet

To send a packet, the node emits it on the link:

```
KANetworkNode >> send: aPacket via: aLink
    aLink emit: aPacket
```

For the simulation to be realistic, we do not want the packet to be delivered right away; instead, emitting a packet really just stores it in the link, until the user elects this packet to proceed using the `transmit:` message. Storing packets requires adding an instance variable to `KANetworkLink`, as well as specifying how this instance variable should be initialized.

```
Object subclass: #KANetworkLink
    instanceVariableNames: 'source destination packetsToTransmit'
    classVariableNames: ''
    category: 'NetworkSimulator-Core'
```

```
KANetworkLink >> initialize
    packetsToTransmit := OrderedCollection new
```

```
KANetworkLink >> emit: aPacket
    "Packets are not transmitted right away, but stored.
    Transmission is explicitly triggered later, by sending
    #transmit:."

    packetsToTransmit add: aPacket
```

We also add a testing method to check whether a given packet is currently being transmitted by a link:

```
KANetworkLink >> isTransmitting: aPacket
    ... Your code ...
```

1.7 Transmitting across a link

Transmitting a packet means telling the link's destination node to receive it. Nodes only consume packets addressed to them; fortunately this is what will happen in our test, so we can worry about the alternative case later (`notYetImplemented` is a special message that we can use in place of code that we will have to write eventually, but prefer to ignore for now).

```
KANetworkNode >> receive: aPacket from: aLink
    aPacket destinationAddress = address
        ifTrue: [
            self consume: aPacket.
            arrivedPackets add: aPacket ]
        ifFalse: [ self notYetImplemented ]
```

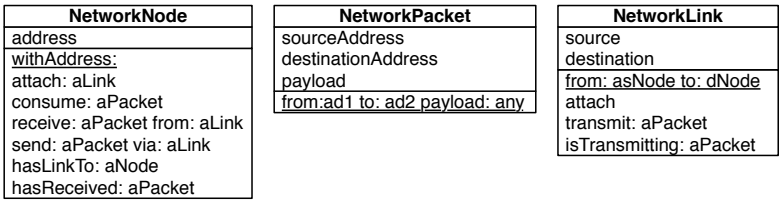

Figure 1-5 Richer API.

Consuming a packet represents what the node will do with it at the application level; for now let's just define an empty `consume:` method, as a placeholder:

```
KANetworkNode >> consume: aPacket
    "Default handling is to do nothing."
```

After consuming the packet, we remember it did arrive; this is mostly for testing and debugging, but someday we might want to simulate packet losses and re-emissions. Don't forget to declare and initialize the `arrivedPackets` instance variable, along with its accessor:

```
KANetworkNode >> hasReceived: aPacket
  ... Your code ...
```

Now we can implement the `transmit:` message. A link can not transmit packets that have not been sent via it, and once transmitted, the packet should not be on the link anymore. We should remove it from the link list of package to be transmitted and tell the destination to receive it using the message `receive:from:`.

```
KANetworkLink >> transmit: aPacket
    "Transmit aPacket to the destination node of the receiver link."
    ... Your code ...
```

At that point all our tests should pass. Note that the message `notYetImplemented` is not called, since our tests do not yet require routing. Figure 1-5 shows that the API of our classes is getting richer than before.

1.8 The loopback link

On a real network, when a node wants to send a packet to itself, it does not need any connection to do so. In real-world networking stacks, loopback routing shortcuts the lower networking layers; however, this is finer detail than we are modeling here.

Still, we want to model the fact that the loopback link is a little special, so each node will store its own loopback link, separately from the outgoing

links. We start to define a test.

```
KANetworkEntitiesTest >> testLoopback
    | node packet |
    node := KANetworkNode withAddress: #address.
    packet := KANetworkPacket from: #address to: #address payload:
    #payload.

    node send: packet.
    node loopback transmit: packet.

    self assert: (node hasReceived: packet).
    self deny: (node loopback isTransmitting: packet)
```

The loopback link is implicitely created as part of the node itself. We also introduce a new `send:` message, which takes the responsibility of selecting the link to emit the packet. For triggering packet transmission, we have to use a specific accessor to find the loopback link of the node.

First, we have to add yet another instance variable in nodes:

```
Object subclass: #KANetworkNode
    instanceVariableNames: 'address outgoingLinks loopback
    arrivedPackets'
    classVariableNames: ''
    category: 'NetworkSimulator-Core'
```

As with all instance variables, we have to remember to make sure it is correctly initialized; we thus modify `initialize`:

```
KANetworkNode >> initialize
    ... Your code ...
```

The accessor has nothing special:

```
KANetworkNode >> loopback
    ^ loopback
```

And finally we can focus on the `send:` method and automatic link selection. The method `send:` should be more generic than the method `send:via:` and will be one exposed as a public entry point.

This method has to rely on some routing algorithm to identify which links will transmit the packet closer to its destination. Since some routing algorithms select more than one link, we will implement routing as an *iteration* method, which evaluates the given block for each selected link.

```
KANetworkNode >> send: aPacket
    "Send aPacket, leaving the responsibility of routing to the
    node."
    self
        linksTowards: aPacket destinationAddress
        do: [ :link | self send: aPacket via: link ]
```

One of the simplest routing algorithm is *flooding*: just send the packet via every outgoing link. Obviously, this is a waste of bandwidth, but it works without any knowledge of the network topology beyond the list of outgoing links.

However, there is one case where we know how to route the packet: if the destination address matches the one of the current node, we can select the loopback link. The logic of `linksTowards:do:` is then: compare the packet's destination address with the one of the node, if it is the same, we execute the block using the loopback link, else we simply iterate on the outgoing links of the receiver.

```
KANetworkNode >> linksTowards: anAddress do: aBlock
    "Simple flood algorithm: route via all outgoing links.
    However, just loopback if the receiver node is the routing
    destination."
    ... Your code ...
```

Now we have the basic model working, and we can try more realistic examples.

1.9 Modeling the network itself

More realistic tests will require non-trivial networks. We thus need an object that represents the network as a whole, to avoid keeping many nodes and links in individual variables. We will introduce a new class `KANetwork`, whose responsibility is to help us build, assemble then find the nodes and links involved in a network.

Let's start by creating another test class, to keep things in order:

```
TestCase subclass: #KANetworkTest
    instanceVariableNames: 'net hub alone'
    classVariableNames: ''
    category: 'NetworkSimulator-Tests'
```

Since every test needs to rebuild the whole example network from scratch, we specify so in the `setUp` method:

```
KANetworkTest >> setUp
    self buildNetwork
```

Before anything else, let's write a test that will pass once we've made progress; we want to access network nodes given only their addresses. Here we check that we get a hub node based on its address:

```
KANetworkTest >> testNetworkFindsNodesByAddress
    self
        assert: (net nodeAt: hub address ifNone: [ self fail ])
        equals: hub
```

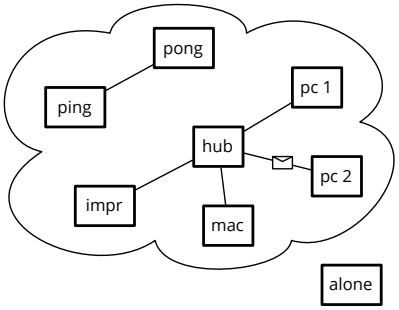

Figure 1-6 A hub.

We will have to implement this `nodeAt:ifNone:` on our `KANetwork` class; but first we need to decide how its instances are built. Let's build network `net`, with the main part connected in a star shape around a `hub` node; a pair of nodes `ping` and `pong` are part of the network but not connected to `hub`, and the `alone` node is just by itself, not even added to the network as shown in Figure 1-6.

Expanding a network implies adding new connections and possibly new nodes to it. If the `net` object understands a `connect: aNode to: anotherNode` message, you should be able to build nodes and connect them into a network that matches the figure.

```
KANetworkTest >> buildNetwork
  alone := KANetworkNode withAddress: #alone.
  net := KANetwork new.
  hub := KANetworkNode withAddress: #hub.
  #(mac pc1 pc2 prn)
    do: [ :addr |
      | node |
      node := KANetworkNode withAddress: addr.
      net connect: node to: hub ].
  net connect: (KANetworkNode withAddress: #ping) to: (KANetworkNode
    withAddress: #pong)
```

The name of the `connect:to:` message suggests that establishing the bidirectional links is the responsibility of the `net` object. It also has to remember enough info so we can inspect the network topology; we can simply store nodes and links in a couple of sets, even though that representation is a little redundant. Let's define the class with two instance variables:

```
Object subclass: #KANetwork
    instanceVariableNames: 'nodes links'
    classVariableNames: ''
    category: 'NetworkSimulator-Core'
```

Whenever we define an instance variable, initialization comes next:

```
KANetwork >> initialize
    ... Your code ...
```

Now we can give the network the possibility to create links. This method we will use to add links to the network link collection.

```
KANetwork >> makeLinkFrom: aNode to: anotherNode
  ^ KANetworkLink from: aNode to: anotherNode
```

We add a low level method `add:` to add a node in a network.

```
KANetwork >> add: aNode
  nodes add: aNode
```

To be able to test the network construction we add a little test message;

```
KANetwork >> doesRecordNode: aNode
  ^ nodes includes: aNode
```

Now, we can add isolated nodes to the network, even if it does not seem very useful.

Connecting nodes.

Connecting nodes without ensuring that they are part of the network really does not make sense. Therefore, when connecting nodes, we will first ensure the nodes are added (by simply adding them in the node Set of the network), then we create and attach links in *both* directions; finally we store both links.

Here is a test covering this aspect.

```
KANetworkTest >> testConnect
  | netw hubb mac pc1 |
  netw := KANetwork new.
  hubb := KANetworkNode withAddress: #hub.
  mac := KANetworkNode withAddress: #mac.
  pc1 := KANetworkNode withAddress: #pc1.

  netw connect: hubb to: mac.
  self assert: (hubb hasLinkTo: mac).
  self assert: (mac hasLinkTo: hubb).
  self assert: (netw doesRecordNode: hubb).
  self assert: (netw doesRecordNode: mac).

  netw connect: hubb to: pc1.
  self assert: (hubb hasLinkTo: pc1).
  self assert: (mac hasLinkTo: hubb)
```

Now implement the `connect:to:` method; for concision, note that the `attach` method we defined previously effectively returns the link.

```
KANetwork >> connect: aNode to: anotherNode
    ... Your code ...
```

The test `testConnect` should be green.

1.10 Looking up nodes

At this point, the test `testNetworkFindsNodesByAddress` should run through `setUp` but fail in the unit test itself, because we still need to implement node lookup. The base lookup should find the first node that has the requested address, or evaluate a fall-back block (a perfect case for the `detect:ifNone:` message):

```
KANetwork >> nodeAt: anAddress ifNone: noneBlock
    ... Your code ...
```

We can also make a convenience `nodeAt:` method for node lookup, that will raise the predefined `NotFound` exception if it does not find the node. Let's first write a test which validates this behavior:

```
KANetworkTest >> testNetworkOnlyFindsAddedNodes
    self
        should: [ net nodeAt: alone address ]
        raise: NotFound
```

Then we can simply express `nodeAt:` by delegating to `nodeAt:ifNone:`. Note that raise an exception, you simply send the message `signal` to the exception class. Here we use the specific class method `signalFor:in:` defined on the `NotFound` class.

```
KANetwork >> nodeAt: anAddress
    ^ self
        nodeAt: anAddress
        ifNone: [ NotFound signalFor: anAddress in: self ]
```

1.11 Looking up links

Next, we want to be able to lookup links between two nodes. Again we define a new test:

```
KANetworkTest >> testNetworkFindsLinks
    | link |
    self
        shouldnt: [ link := net linkFrom: #pong to: #ping ]
        raise: NotFound.
    self
        assert: link source
        equals: (net nodeAt: #pong).
    self
        assert: link destination
```

```
        equals: (net nodeAt: #ping)
```

And we define the method `linkFrom:to:` returning the link between source and destination nodes with matching addresses, and signalling `NotFound` if no such link is found:

```
KANetwork >> linkFrom: sourceAddress to: destinationAddress
    ... Your code ...
```

Final check.

As a final check, let's try some of the previous tests, first on the isolated `alone` node, showing that loopback works even without a network connection:

```
KANetworkTest >> testSelfSend
    | packet |
    packet := KANetworkPacket
        from: alone address
        to: alone address
        payload: #something.
    self assert: (packet isAddressedTo: alone).
    self assert: (packet isOriginatingFrom: alone).

    alone send: packet.
    self deny: (alone hasReceived: packet).
    self assert: (alone loopback isTransmitting: packet).

    alone loopback transmit: packet.
    self deny: (alone loopback isTransmitting: packet).
    self assert: (alone hasReceived: packet)
```

You can see that we used new convenience testing methods `isAddressedTo:` and `isOriginatingFrom:` which help inspect the state of a simulated network without explicitly comparing addresses. However, those methods should not take part in network simulation code, since in the real world nodes can never know their peers other than through their addresses.

```
KANetworkPacket >> isAddressedTo: aNode
    ^ destinationAddress = aNode address
```

```
KANetworkPacket >> isOriginatingFrom: aNode
    ^ sourceAddress = aNode address
```

The second test attempts transmitting a packet in the network, between the directly connected nodes `ping` and `pong`:

```
KANetworkTest >> testDirectSend
    | packet ping pong link |
    packet := KANetworkPacket from: #ping to: #pong payload: #ball.
    ping := net nodeAt: #ping.
    pong := net nodeAt: #pong.
```

```
    link := net linkFrom: #ping to: #pong.

    ping send: packet.
    self assert: (link isTransmitting: packet).
    self deny: (pong hasReceived: packet).

    link transmit: packet.
    self deny: (link isTransmitting: packet).
    self assert: (pong hasReceived: packet)
```

Both tests should pass with no additional work, since they just reproduce what we already tested in `KANetworkEntitiesTest` and adding `KANetwork` did not impact the established behavior of nodes, links, and packets.

1.12 Packet delivery with forwarding

Until now, we only tested packet delivery between directly connected nodes; let's try sending a node so that the packet has to be forwarded through the hub.

```
KANetworkTest >> testSendViaHub
    | hello mac pc1 firstLink secondLink |
    hello := KANetworkPacket from: #mac to: #pc1 payload: 'Hello!'.
    mac := net nodeAt: #mac.
    pc1 := net nodeAt: #pc1.
    firstLink := net linkFrom: #mac to: #hub.
    secondLink := net linkFrom: #hub to: #pc1.

    self assert: (hello isAddressedTo: pc1).
    self assert: (hello isOriginatingFrom: mac).

    mac send: hello.
    self deny: (pc1 hasReceived: hello).
    self assert: (firstLink isTransmitting: hello).

    firstLink transmit: hello.
    self deny: (pc1 hasReceived: hello).
    self assert: (secondLink isTransmitting: hello).

    secondLink transmit: hello.
    self assert: (pc1 hasReceived: hello).
```

If you run this test, you will see that it fails because of the `notYetImplemented` message we left earlier in `receive:from:`; it's time to fix that! When a node receives a packet but is not the recipient, it should forward the packet:

```
KANetworkNode >> receive: aPacket from: aLink
    aPacket destinationAddress = address
        ifTrue: [
            self consume: aPacket.
            arrivedPackets add: aPacket ]
        ifFalse: [ self forward: aPacket from: aLink ]
```

Now we need to implement packet forwarding, but there is a trap. An easy solution would be to simply `send:` the packet again: the hub would send the packet to all its connected nodes, one of which happens to be `pc1`, the recipient, so all is good!

Wrong...

The packet would be also sent to other nodes than the recipient; what would those nodes do when they receive a packet not addressed to them? Forward it. Where? To all their neighbours, which would forward it again... so when would the forwarding stop?

To fix this, we need hubs to behave differently from nodes. In reality, hubs work at the lower layers of the OSI model, but our simplified model does not have that level of detail. We can approximate this by saying that upon reception of a packet addressed to another node, a hub should forward the packet, but a normal node should just ignore it.

Let's first define an empty `forward:from:` method for nodes, then add a new class for hubs, which will be modeled as nodes with an actual implementation of forwarding:

```
KANetworkNode >> forward: aPacket from: arrivalLink
    "Do nothing. Normal nodes do not route packets."
```

1.13 Introducing a new kind of node

Now we define the class `KANetworkHub` that will be the recipient of hub specific behavior.

```
KANetworkNode subclass: #KANetworkHub
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'NetworkSimulator'
```

A hub does not have routing information, so all it can do is flood routing, with a catch: the packet must not be sent back from where it arrived, because if that happens to be another hub the packet would bounce back and forth indefinitely. We suggest to take advantage of the message `linksTowards:do:` that performs an action for all given links to one address.

```
KANetworkHub >> forward: aPacket from: arrivalLink
   ... Your code ...
```

Now we can use a proper hub in our test, replacing the relevant line in `KANetworkTest >> buildNetwork`, and check that the `testSendViaHub` unit test passes.

```
    hub := KANetworkHub withAddress: #hub.
```

You have now a nice basis for network simulation. In the following we will present some possible extensions.

1.14 Other examples of specialized nodes

In this section we will present some extensions of the core to support different scenarios. We will propose some tasks to make sure that the extensions are fully working. In addition in this section we do not define tests and we strongly encourage you to start to write tests. At the moment of the book you should be ready to write your own tests and see their values to improve your development process. So take this opportunity to practice.

Workstations counting received packets

We would like to know how many packets specific nodes are receiving. In particular when a workstation consumes a packet, it simply increments a packet counter.

Let's start by subclassing `KANetworkNode`:

```
KANetworkNode subclass: #KANetworkWorkstation
    instanceVariableNames: 'receivedCount'
    classVariableNames: ''
    category: 'NetworkSimulator-Nodes'
```

We need to initialize the `receivedCount` instance variable. Properly redefining `initialize` is enough, because the address is initialized separately in the constructor method `KANetworkNode >> withAddress:`; however, it's really important not to forget the `super initialize` message, because that method does initialize the default node behavior.

```
KANetworkWorkstation >> initialize
    super initialize.
    receivedCount := 0
```

Now we can redefine `consume:` accordingly:

```
KANetworkWorkstation >> consume: aPacket
    receivedCount := receivedCount + 1
```

Define accessors and the `printOn:` method for debugging. Define a test for the behavior of workstation nodes.

Printers accumulating printouts

When a printer consumes a packet, it prints it; we can model the output tray as a list where packet payloads get queued, and the supply tray as the number of blank sheets it contains.

The implementation is very similar; we subclass `KANetworkNode` to redefine the `consume:` method:

```
KANetworkNode subclass: #KANetworkPrinter
    instanceVariableNames: 'supply tray'
    classVariableNames: ''
    category: 'NetworkSimulator-Nodes'
```

```
KANetworkPrinter >> consume: aPacket
    supply > 0 ifTrue: [ ^ self "no paper, do nothing" ].

    supply := supply - 1.
    tray add: aPacket payload
```

Initialization is a bit different, though; since the standard `initialize` method has no argument, the only sensible initial value for the `supply` instance variable is zero:

```
KANetworkPrinter >> initialize
    super initialize.
    supply := 0.
    tray := OrderedCollection new
```

We therefore need a way to pass the initial supply of paper available to a fresh instance:

```
KANetworkPrinter >> resupply: paperSheets
    supply := supply + paperSheets
```

For convenience, we can provide an extended constructor to create printers with a non-empty supply in one message:

```
KANetworkPrinter class >> withAddress: anAddress initialSupply:
    paperSheets
    ^ (self withAddress: anAddress)
        resupply: paperSheets;
        yourself
```

Define accessors and the `printOn:` method for debugging purpose. Define some test methods for the behavior of printer nodes.

Servers answering requests

When a server node consumes a packet, it converts the payload to uppercase, then sends that back to the sender of the request.

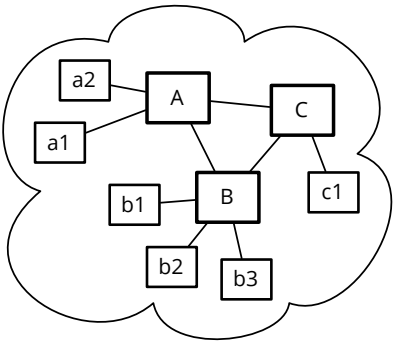

Figure 1-7 A possible extension: a more realistic network with a cycle between three router nodes.

This is yet another subclass which redefines the `consume:` method, but this time the node is stateless, so we have no initialization or accessor methods to write:

```
KANetworkNode subclass: #KANetworkServer
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'NetworkSimulator-Nodes'
```

```
KANetworkServer >> consume: aPacket
    | response |
    response := aPacket payload asUppercase.
    self send: (KANetworkPacket
        from: self address
        to: aPacket sourceAddress
        payload: response)
```

Define a test for the behavior of server nodes.

1.15 Conclusion

In this chapter, we built a little network simulation system, step by step. We showed the benefit of good protocol decompositions.

As a further extension, we suggest modeling a more realistic network with cycles, as shown in Figure 1-7. Making this work properly will require replacing hubs with routers and flood routing with more realistic routing algorithms.

Here is a possible setup for a new family of tests.

```
KARoutingNetworkTest >> buildNetwork
    | routers |
    net := KANetwork new.

    routers := #(A B C) collect:
        [ :each | KANetworkHub withAddress: each ].
    net connect: routers first to: routers second.
    net connect: routers second to: routers third.
    net connect: routers third to: routers first.

    #(a1 a2) do: [ :addr |
        net connect: routers first
            to: (KANetworkNode withAddress: addr) ].
    #(b1 b2 b3) do: [ :addr |
        net connect: routers second
            to: (KANetworkNode withAddress: addr) ].
    net connect: routers third
        to: (KANetworkNode withAddress: #c1)
```

Bibliography