CHAPTER 17

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.

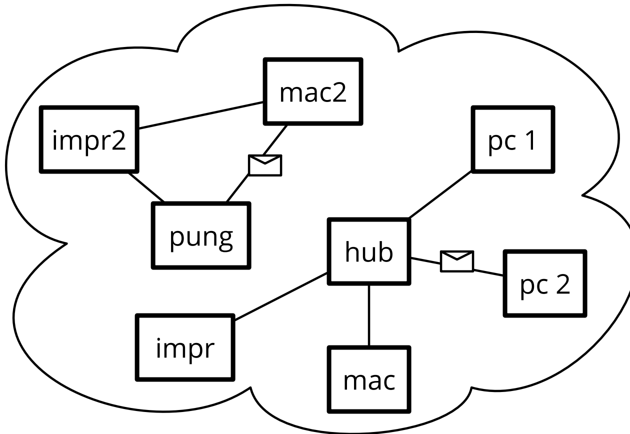


Figure 17-1 Two little networks composed of nodes and sending packets over links.

17.1 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.

17.2 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.

17.3 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
```

NetworkNode
address
withAddress:
attach: aLink
hasLinkTo: aNode

NetworkPacket
sourceAddress
destinationAddress
payload
from:ad1 to: ad2 payload: any

NetworkLink
source
destination
from: asNode to: dNode
attach

Figure 17-2 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.

17.4 Making our objects more understandable

When programming we often make mistakes and it is important to help developer to address them. Le 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 17-3. 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 , ')'
```

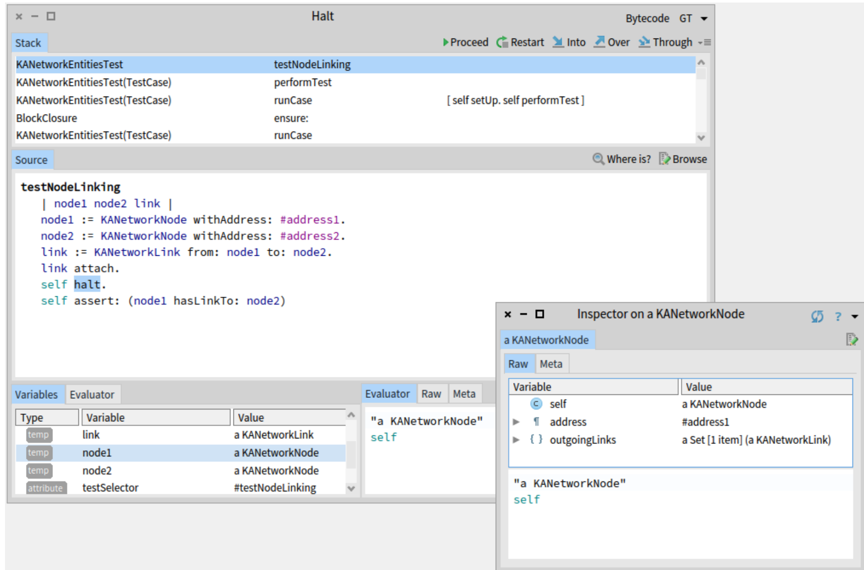


Figure 17-3 Navigating specific objects having a generic presentation.

```
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 17-4: we can see the address of a node and the source and destination of a link.

17.5 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.
```

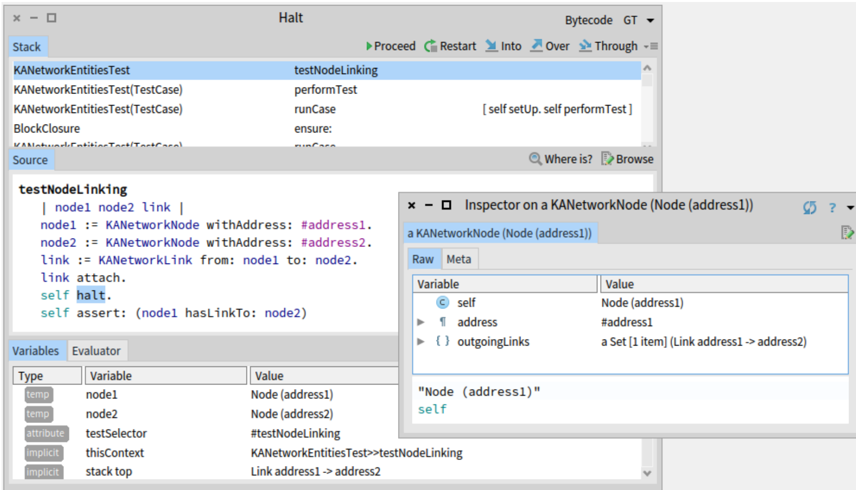


Figure 17-4 Navigating objects offering a customized presentation.

```
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.

17.6 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 ...
```

17.7 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 ]
```

NetworkNode
address
withAddress:
attach: aLink
consume: aPacket
receive: aPacket from: aLink
send: aPacket via: aLink
hasLinkTo: aNode
hasReceived: aPacket

NetworkPacket
sourceAddress
destinationAddress
payload
from:ad1 to: ad2 payload: any

NetworkLink
source
destination
from: asNode to: dNode
attach
transmit: aPacket
isTransmitting: aPacket

Figure 17-5 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 17-5 shows that the API of our classes is getting richer than before.

17.8 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 alone. The logic of `linksTowards:do:` is then to check is the address we want to send the packet is the one of the node. In that case we execute the block using the loopback link, else we simple 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.

17.9 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
```

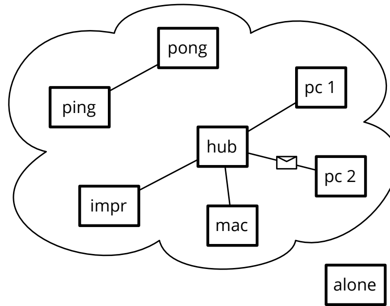


Figure 17-6 A hub.

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 17-6.

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.

17.10 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 ]
```

17.11 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.

17.12 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."
```

17.13 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.

17.14 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 method 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.

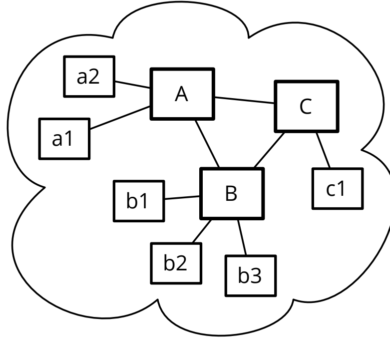


Figure 17-7 A possible extension: a more realistic network with a cycle between three router nodes.

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.

17.15 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 17-7. 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)
```

CHAPTER 18

Snakes and ladders

Snakes and Ladders is a simple game suitable for teaching children how to apply rules (http://en.wikipedia.org/wiki/Snakes_and_ladders). It is dull for adults because there is absolutely no strategy involved, but this makes it easy to implement! In this chapter you will implement SnakesAndLadders and we use it as a pretext to explore design questions.

18.1 Game rules

Snakes and Ladders originated in India as part of a family of die games. The game was introduced in England as "Snakes and Ladders" (see Figure 18-1), then the basic concept was introduced in the United States as *Chutes and Ladders*. Here is a brief description of the rules:

- **Players:** Snakes and Ladders is played by two to four players, each with her/his own token to move around the board.

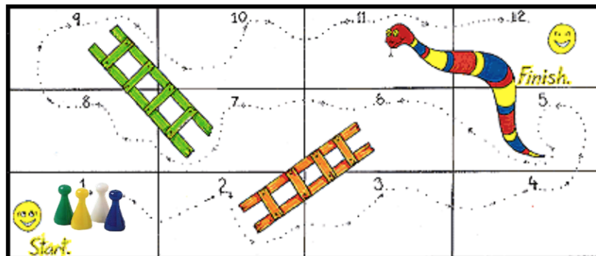


Figure 18-1 An example Snakes and Ladders board with two ladders and a snake.

- **Moving Player**: a player rolls a die, then moves the designated number of tiles, between one and six. Once he lands on a tile, she/he has to perform any action designated by the tile. (Since the rules are fuzzy we decided that we can have multiple players in the same tile).
- **Ladders:** If the tile a player lands on is at the bottom of a ladder, she/he should climb the ladder, which brings him to a tile higher on the board.
- **Snakes:** If the tile a player lands on is a head snake, she/he must slide down the snake, landing on a tile closer to the beginning.
- **Winning:** the winner is the player who gets to the last tile first, whether by landing on it from a roll, or by reaching it with a ladder. We decided that when the player does not move if he does not land directly on the last tile, it does not move.

18.2 Game possible run

The code snippet below is a possible way to program this game. We take as a board configuration the one depicted in Figure 18-1. It defines a board game composed of 12 tiles with two ladders and one snake. We add two players and then start the game.

```
| jill jack game |
game := SLGame new tileNumber: 12.
game
  setLadderFrom: 2 to: 6;
  setLadderFrom: 7 to: 9;
  setSnakeFrom: 11 to: 5.
game
  addPlayer: (SLPlayer new name: 'Jill');
  addPlayer: (SLPlayer new name: 'Jack').
game play
```

Since we want to focus on the game logic, you will develop a textual version of the game and avoid any lengthy user interface descriptions.

The following is an example game execution: Two players are on the first tile. The board contains two ladders, [2->6] and [7->9], and one snake [5<-11].

Jill rolls a die and throws a 3 and moves to the corresponding tile. Jack rolls a die and throws a 6 and moves to the corresponding tile and follow its effect, climbing the ladder at tile 7 up to tile 9. Jack and Jill continue to alternate taking turns until Jill ends up on the last tile.

```
[1<Jill><Jack>][2->6][3][4][5][6][7->9][8][9][10][5<-11][12]
<Jill>throws 3:
    [1<Jack>][2->6][3][4<Jill>][5][6][7->9][8][9][10][5<-11][12]
<Jack>throws 6:
    [1][2->6][3][4<Jill>][5][6][7->9][8][9<Jack>][10][5<-11][12]
```

```
<Jill>throws 5:
    [1][2->6][3][4][5][6][7->9][8][9<Jack><Jill>][10][5<-11][12]
<Jack>throws 1:
    [1][2->6][3][4][5][6][7->9][8][9<Jill>][10<Jack>][5<-11][12]
<Jill>throws 3:
    [1][2->6][3][4][5][6][7->9][8][9][10<Jack>][5<-11][12<Jill>]
```

18.3 Potential objects and responsibilities

Take a piece of paper, study the game rules and list any potential objects and their behavior. This is an important exercise to practice, training yourself to discover potential objects and classes.

Techniques such as *Responsibility Driven Design* exist to help programmers during this phase of object discovery. Responsibility Driven Design suggests analysing the documents describing a project, and turning the subjects of sentences into candidate objects and grouping verbs as the behavior of these objects. Any synonyms are identifed and used to reduce and gather together similar objects or behavior. Then later objects are grouped into classes. Some alternate approaches look for relationship patterns between objects such as part-whole, locations, entity-owner... This could be the topic of a full book.

Here we follow another path: sketching scenarios. We describe several scenarios and from such scenario we identify key playing objects.

- Scenario 1. The game is created with a number of tiles. The game must have an end and start tiles. Ladders and snakes should be declared.
- Scenario 2. Players are declared. They start on the first tiles.
- Scenario 3. When player rolls a die, he should move the number of tiles given by the die.
- Scenario 4. After moving the first player a given number of tiles based on the result of die roll, this is the turn of the second player.
- Scenario 5. When a player arrives to a ladder start, it should be moved to the ladder end.
- Scenario 6. When a player should move further than the end tile, he does not move.
- Scenario 7. When a player ends its course on the end tile, he wins and the game is finished.

Such scenarios are interesting because they are a good basis for tests.

Possible class candidates

When reading the rules and the scenario, here is a list of possible classes that we could use. We will refine it later and remove double or overlapping concepts.

- Game: keeps track of the game state, the players, and whose turn it is.
- Board: keeps the tile configuration.
- Player: keeps track of location on the board and moving over tiles.
- Tile: keeps track of any player on it.
- Snake: is a special tile which sends a player back to an earlier tile.
- Ladder: is a special tile which sends a player ahead to a later tile.
- First Tile: holds multiple players at the beginning of the game.
- Last Tile: players must land exactly on this tile, or else they do not move.
- Die: rolls and indicates the number of tiles that a player must move over.

It is not clear if all the objects we identify by looking at the problem and its scenario should be really turned into real objects. Also sometimes it is useful to get more classes to capture behavior and state variations. We should look to have an exact mapping between concepts identified in the problem scenario or description and the implementation.

From analysing this list we can draw some observations:

- Game and Board are probably the same concept and we can merge them.
- Die may be overkill. Having a full object just to produce a random number may not be worth, especially since we do not have a super fancy user interface showing the die rolling and other effect.
- Tile, Snake, Ladder, Last and First Tile all look like tiles with some variations or specific actions. We suspect that we can reuse some logic by creating an inheritance hierarchy around the concept of Tile.

About representation

We can implement the same system using different implementation choices. For example we could have only one class implementing all the game logic and it would work. Some people may also argue that this is not a bad solution.

Object-oriented design favors the distribution of the state of the system to different objects. It is often better to have objects with clear responsibilities.

Why? Because you should consider that you will have to rethink, modify or extend your system. We should be able to understand and extend easily a system to be able to reply to new requirements.

Not having a nice object-oriented decomposition for a simple game may not be a problem, as soon as you will start to model a more complex system not having a good decomposition will hurt you. Real life applications often have a lifetime up to 25 years.

In addition, imagine that we are a game designer and we want to experiment with different variations and tiles with new properties such as one super special tile changing other tiles, adding snakes before the current player to slow other participants.

18.4 About object-oriented design

When designing a system, you will often have questions that cannot be blindly and automatically answered. Often there is no definite answer. This is what is difficult with object-oriented design and this is why practicing is important.

What composes the state of an object? The state of object should characterize the object over its lifetime. For example the name of player identifies the player.

Now it may happen that some objects just because they are instances of different classes do not need the same state but still offer the same set of messages. For example the tiles and the ladder/snake tiles have probably a similar API but snake and ladder should hold information of their target tile.

We can also distinguish between the intrinsic state of an object (e.g., name of player) and the state we use to represent the collaborators of an object.

The other important and difficult question is about the relationships between the objects. For example imagine that we model a tile as an object, should this object points to the players it contains. Similarly, should a tile knows its position or just the game should know the position of each tile.

Should the game object keep the position of the players or just the player. The game should keep the players list since it should compute who is the next player.

CRC cards

Some designers use CRC (for Class Responsibility Collaborators) cards: the idea is to take the list of classes we identified above. For each of them, they write on a little card: the class name, its responsibility in one or two sentences and list its collaborators. Once this is done, they take a scenario and

see how the objects can play such a scenario. Doing so they refine their design by adding more information (collaborators) to a class or merging two classes or splitting a class into multiple ones when they fill that a class has too many responsibilities.

To improve such process, some designers consider implementation concerns or alternatives and may create objects to represent such variations.

Some heuristics

To help us taking decision, that are some heuristics:

- One object should have one main responsibility.
- Move behavior close to data. If a class defines the behavior of another object, there is a good chance that other clients of this object are doing the same and create duplicated and complex logic. If an object defines a clear behavior, clients just invoke it without duplicating it.
- Prefer domain object over literal objects. As a general principle it is better to get a reference to a more general objects than a simple number. Because we can then invoke a larger set of behavior.

Kind of data passed around

Even if in Pharo, everything is an object, storing a mere integer object instead of a full tile can lead to different solutions. There is no perfect solution mainly consequences of choices and you should learn how to assess a situation to see which one has better characteristics for your problem.

Here is a question illustrating the problem: Should a ladder know the tile it forwards the player to or is the index of a tile enough?

When designing the ladder tile behavior, we should understand how we can access the target tile where the player should be moved to. If we just give the index of the target to a ladder, the tile has to be able to access the board containing the tiles else it will be impossible to access to the target tile of the ladder. The alternative, i.e., passing the tile looks nicer because it represents a natural relation and there is no need to ask the board.

Agility to adapt

In addition it is important not to get stressed, writing tests that represent parts or scenario we want to implement is a good way to make sure that we can adapt in case we discover that we missed a point.

Now this game is interesting also from a test point of view because it may be difficult to test the parts in isolation (i.e., without requiring to have a game object).

18.5 Let us get started

You will follow an iterative process and test first approach. You will take scenario implement a test and define the corresponding classes.

This game implementation raises an interesting question which is how do we test the game state without hardcoding too much implementation details in the tests themselves. Indeed tests that validate scenario only involving public messages and high-level interfaces are more likely to be stable over time and do not require modifications. Indeed if we check the exact class of certain objects you will have to change the implementation as well as the tests when modifying the implementation. In addition, since in Pharo the tests are normal clients of the objects they test, writing some tests may force us to define extra methods to access to private data.

But enough talking! Let us start by defining a test class named SLGameTest. We will see in the course of development if we define other test classes. Our feeling is that the tiles and players are objects with limited responsibility and their responsibility is best illustrated (and then tested) when they interact with each other in the context of a given game. Therefore the class SLGameTest describes the place in which relevant scenario will occur.

Define the class SLGameTest.

```
TestCase subclass: #SLGameTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'SnakesAndLadders'
```

One of the first scenario is that a game is composed of a certain number of tiles.

We can write a test as follows but it does not have a lot of value. At the beginning of the development, this is normal to have limited tests because we do not have enough objects to interact with.

```
SLGameTest >> testCheckingSimpleGame

  | game |
  game := SLGame new tileNumber: 12.
  self assert: game tileNumber equals: 12
```

Now we should make this test pass. Some strong advocates of TDD say that we should code the first simplest method that would make the test pass and go to the next one. Let us see what it would be (of course this method will be changed later).

First you should define the class SLGame.

```
Object subclass: #SLGame
  instanceVariableNames: 'tiles'
  classVariableNames: ''
  package: 'SnakesAndLadders'
```

Now you can define the methods `tileNumber:` and `tileNumber`. This is not really nice because we should get a collection of tiles and now we put a number.

```
SLGame >> tileNumber: aNumber
  tiles := aNumber
```

```
SLGame >> tileNumber
  ^ tiles
```

These method definitions are enough to make our test pass. It means that our test was not really good because tiles should hold a collection containing the tiles and not just a number. We will address this point later.

18.6 A first real test

Since we would like to be able to check that our game is correct we can use its textual representation and test it as a way to check the game state. The following test should what we want.

```
SLGameTest >> testPrintingSimpleGame

  | game |
  game := SLGame new tileNumber: 12.
  self
    assert: game printString
    equals: '[1][2][3][4][5][6][7][8][9][10][11][12]'
```

What we would like is that the printing of the game asks the tiles to print themselves this way we will be able to take advantage that there will be different tiles in a modular way: i.e. we will not change the game to display the ladder and snake just have different tiles with different behavior.

The first step is then to define a class named `SLTile` as follows:

```
Object subclass: #SLTile
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'SnakesAndLadders'
```

Now we would like to test the printing of a single tile. So let us define a test case named `SLTileTest`. This test case will test some basic behavior but it is nice to decompose our implementation process. We are trying to minimize the gap between one functionality and one test.

```
TestCase subclass: #SLTileTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'SnakesAndLadders-Test'
```

Now we can write a simple test to make sure that we can print a tile.

```
SLTileTest >> testPrinting

  | tile |
  tile := SLTile new position: 6.
  self assert: tile printString equals: '[6]'
```

Tile position could have been managed by the game itself. But it means that we would have to ask the game for the position of a given tile and while it would work, it does not feel good. In Object-Oriented Design, we should distribute responsibilities to objects and their state is their first responsibility. Since the position is an attribute of a tile, better define it there.

This is where you see that the fact that the code is running is not a quality test for good Object-Oriented Design.

In particular it means that we should add an accessor to set the position and to add an instance variable `position` to the class `SLile`. Execute the test. You should get a debugger and use it to create a method `position:` as well as the instance variable.

Now we can define the `printOn:` method for tiles as follows. We add a `[` into the stream, then we asked the position to print itself in the stream by sending it the message `printOn:` and we add `]` in the stream. Since the position is a simple integer, the result of the `position printOn: aStream` expression is just to add a string representing the number in the stream.

```
SLTile >> printOn: aStream

  aStream << '['.
  position printOn: aStream.
  aStream << ']'
```

Your tile test should pass now. When we read the definition of the method `printOn:` above we see that it also sends the message `printOn:` here to the number used for the position. Indeed, we can send messages with the same name to different objects and each object may react differently to these messages. We can also send a message with the same name than the method to the receiver to perform a recursive call, but as with any recursive call we should have a non recursive branch.

We are ready to finish the printing of the game itself. Now we can define the method `printOn:` of the game to print all its tiles. Note that this will not work since so far we did not create tiles.

```
SLGame >> printOn: aStream

  tiles do: [ :aTile |
    aTile printOn: aStream ]
```

We modify the method `tileNumber:` to create an array of the given size and store it inside the `tiles` instance variable and to put a new tile for each position. Pay attention the tile should have the correct position.

```
SLGame >> tileNumber: aNumber
  ... Your code ...
```

Now your printing tests should be working both for the tile and the game. But wait if we run the test `testCheckingSimpleGame` it fails. Indeed we did not change the definition `tileNumber`. Do it and make sure that your tests all pass. And save your code.

18.7 Accessing one tile

Now we will need to be able to ask the game for a given tile, for example with the message `tileAt:`. Let us add a test for it.

```
SLGameTest >> testTileAt

  | game |
  game := SLGame new tileNumber: 12.
  self assert: (game tileAt: 6) printString equals: '[6]'
```

Define the method `tileAt:`.

```
SLGame >> tileAt: aNumber
  ... Your code ...
```

18.8 Adding players

Now we should add players. The first scenario to test is that when we add a player to game, it should be on the first tile.

Let us write a test: we create a game and a player. Then we add the player to the game and the player should be part of the players of the first tile.

```
SLGameTest >> testPlayerAtStart

  | game jill |
  game := SLGame new tileNumber: 12.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  self assert: ((game tileAt: 1) players includes: jill).
```

```
Object subclass: #SLPlayer
  instanceVariableNames: 'name'
  classVariableNames: ''
  package: 'SnakesAndLadders'
```

Define the method `name:` in the class `SLPlayer`. Now we should think a bit how we should manage the players. We suspect that the game itself should get a list of players so that in the future it can ask each player to play its turn. Notice the previous sentence: we say each player to play and not the game to play the next turn - again this is Object-Oriented Design in action.

Now our test does not really cover the point that the game should keep track of the players so we will not do it. Similarly we may wonder if a player should know its position. At this point we do not know and we postpone this decision for another scenario.

```
SLGame >> addPlayer: aPlayer
  (tiles at: 1) addPlayer: aPlayer
```

Now what is clear is that a tile should keep a player list. Add an instance variable `players` to the `SLTile` class and initialize it to be an OrderedCollection.

```
SLTile >> initialize
  ...  Your code ...
```

Then implement the method `addPlayer:`

```
SLTile >> addPlayer: aPlayer
  ... Your code ...
```

Now all your tests should pass.

Let us the opportunity to write better tests. We should check that we can add two players and that both are on the starting tile.

```
SLGameTest >> testSeveralPlayersAtStart

  | game jill jack |
  game := SLGame new tileNumber: 12.
  jill := SLPlayer new name: 'Jill'.
  jack := SLPlayer new name: 'Jack'.
  game addPlayer: jill.
  game addPlayer: jack.
  self assert: ((game tileAt: 1) players includes: jill).
  self assert: ((game tileAt: 1) players includes: jack).
```

All the tests should pass. This is the time to save and take a break.

Figure 18-2 Playground in action. Use Do it and go - to get an embedded inspector.

18.9 Avoid leaking implementation information

We are not really happy with the previous tests for example `testPlayerAtStart`.

```
SLGameTest >> testPlayerAtStart

  | game jill |
  game := SLGame new tileNumber: 12.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  self assert: ((game tileAt: 1) players includes: jill).
```

Indeed a test is a first client of our code. Here we see in the expression `players includes: jill` that we have to know that players are held in a collection and that this collection includes such a player.

It can be a real problem if later we decide to change how we manage players, since we will have to change all the places using the result of the `players` message.

Let us address this issue: define a method `includesPlayer:` that returns whether a tile has the given player.

```
SLTile >> includesPlayer: aPlayer
  ... Your code ...
```

Now we can rewrite the two tests `testPlayerAtStart` and `testSeveralPlayersAtStart` to use this new message.

```
SLGameTest >> testPlayerAtStart

  | game jill |
  game := SLGame new tileNumber: 12.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  self assert: ((game tileAt: 1) includesPlayer: jill).
```

```
SLGameTest >> testSeveralPlayersAtStart

  | game jill jack |
  game := SLGame new tileNumber: 12.
  jill := SLPlayer new name: 'Jill'.
  jack := SLPlayer new name: 'Jack'.
  game addPlayer: jill.
  game addPlayer: jack.
  self assert: ((game tileAt: 1) includesPlayer: jill).
  self assert: ((game tileAt: 1) includesPlayer: jack).
```

18.10 About tools

Pharo is a living environment in which we can interact with the objects. Let us see a bit of that in action now.

Type the following game creation in a playground (as shown in Figure 18-2).

```
| game jill jack |
game := SLGame new tileNumber: 12.
jill := SLPlayer new name: 'Jill'.
jack := SLPlayer new name: 'Jack'.
game addPlayer: jill.
game addPlayer: jack.
game
```

Now you can inspect the game either using the inspect command-i or sending the message `inspect` to the game as in `game inspect`. You can also use the *do it and go* menu item of a playground window. You should get a picture similar to the one 18-3.

We see that the object is a `SLGame` instance and it has an instance variable named `tiles`. You can navigate on the instance variables as shown in Figure 18-4. Figure 18-5 shows that we can navigate the object structure: here we start from the game, go to the first tile and see the two players. At any moment you can interact with the selected object sending it messages.

18.11 Displaying players

Navigating the structure of the game is nice when we want to debug and interact with the game entities. Now we propose to display the player objects

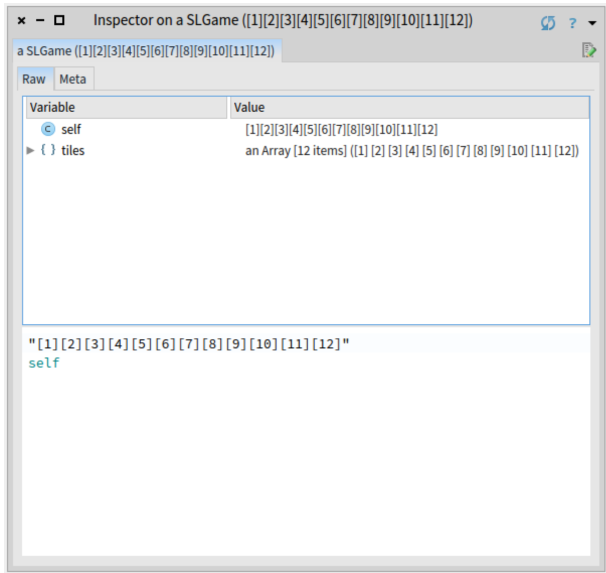


Figure 18-3 Inspecting the game: a game instance and its instance variable `tiles`.

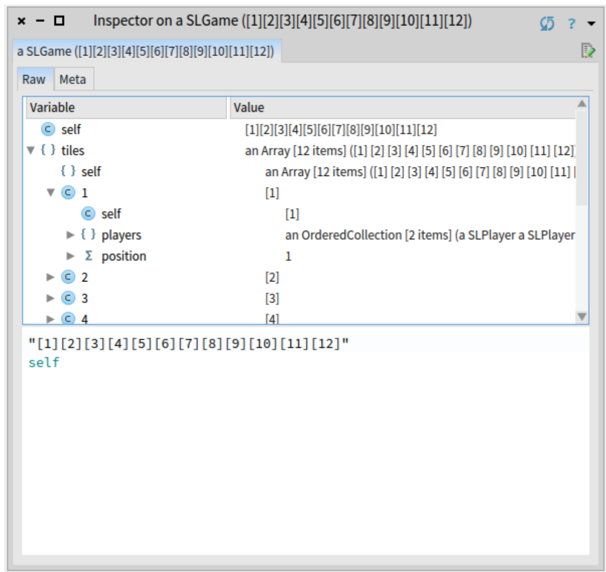


Figure 18-4 Navigating inside the game: getting inside the tiles and checking the players.

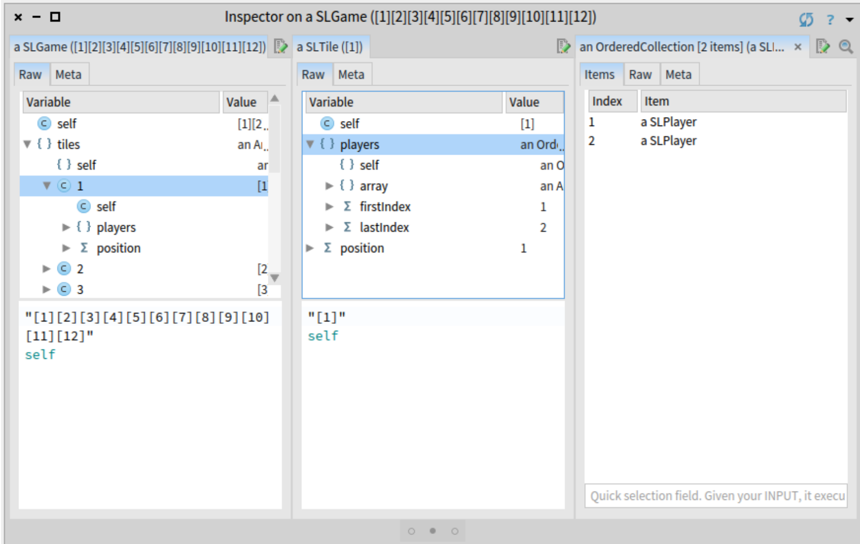


Figure 18-5 Navigating the objects using the navigation facilities of the inspector.

in a nicer way. We will reuse such behavior when printing the game to follow the movement of the player on the board.

Since we love testing, let us write a test describing what we expect when displaying a game.

```
SLGameTest >> testPrintingSimpleGameWithPlayers

  | game jill jack |
  game := SLGame new tileNumber: 12.
  jack := SLPlayer new name: 'Jack'.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill. "first player"
  game addPlayer: jack.
  self
    assert: game printString
    equals: '[1<Jill><Jack>][2][3][4][5][6][7][8][9][10][11][12]'
```

To make this test pass, you must define a `printOn:` on `SLPlayer`. Make sure that the `printOn:` of `SLTile` also invokes this new method.

```
SLPlayer >> printOn: aStream
  ... Your code ...
```

Here is a possible implementation for the tile logic.

```
SLTile >> printOn: aStream
  aStream << '['.
  position printOn: aStream.
  players do: [ :aPlayer | aPlayer printOn: aStream ].
  aStream << ']'
```

Run your tests, they should pass.

18.12 Preparing to move players

To move the player we need to know the tile on which it will arrive. We want to ask the game: what is the target tile if this player (for example, jill) is moving a given distance. Let us write a test for the message `tileFor: aPlayer atDistance: aNumber`.

```
SLGameTest >> testTileForAtDistance

  | jill game |
  game := SLGame new tileNumber: 12.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  self assert: (game tileFor: jill atDistance: 4) position equals: 5.
```

What is implied is that a player should know its location or that the game should start to look from the beginning to find what is the current position of a player. The first option looks more reasonable in terms of efficiency and this is the one we will implement.

Let us write a simpler test for the introduction of the position in a player.

```
SLGameTest >> testPlayerAtStartIsAtPosition1

  | game jill |
  game := SLGame new tileNumber: 12.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  self assert: jill position equals: 1.
```

Define the methods `position` and `position:` in the class `SLPlayer` and add an instance variable `position` to the class. If you run the test it should fail saying that it got nil instead of one. This is normal because we never set the position of a player. Modify the `addPlayer:` to handle this case.

```
SLGame >> addPlayer: aPlayer
  ... Your code ...
```

The test `testPlayerAtStartIsAtPosition1` should now pass and we can return to the `testTileForAtDistance`. Since we lost a bit track, the best thing to do is to run our tests and check why they are failing. We get an error

saying that a game instance does not understand the message `tileFor:atDistance:` this is normal since we never implemented it. For now we do not consider that a roll can bring the player further than the last tile.

Let us fix that now. Define the method `tileFor:atDistance:`

```
SLGame >> tileFor: aPlayer atDistance: aNumber
  ... Your code ...
```

Now all your test should pass and this is a good time to save your code.

18.13 Finding the tile of a player

We can start to move a player from a tile to another one. We should get the tile destination using the message `tileFor:atDistance:` and add the player there. Of course we should not forget that the tile where the player is currently positioned should be updated. So we need to know what is the tile of the player.

Now once a player has position it is then easy to find the tile on top of which it is. Let us write a test for it.

```
SLGameTest >> testTileOfPlayer

  | jill game |
  game := SLGame new tileNumber: 12.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  self assert: (game tileOfPlayer: jill) position equals: 1.
```

Implement the method `tileOfPlayer:`.

```
SLGame >> tileOfPlayer: aSLPlayer
  ... Your code ...
```

18.14 Moving to another tile

Now we are ready to work on moving a player from one tile to the other. Let us express a test: we create only one player. We test that after the move, the new position is the one of the target tile, that the original tile does not have player and the target tile has effectively the player.

```
SLGameTest >> testMovePlayerADistance

  | jill game |
  game := SLGame new tileNumber: 12.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  game movePlayer: jill distance: 4.
  self assert: jill position equals: 5.
```

```
  self assert: (game tileAt: 1) players isEmpty.
  self assert: ((game tileAt: 5) includesPlayer: jill).
```

What is hidden in this test is that we should be able to remove a player from a tile.

Since we should remove the player of a tile when it moves, implement the method

```
SLTile >> removePlayer: aPlayer
  ... Your code ...
```

Now propose an implementation of the method `movePlayer: aPlayer distance: anInteger`. You should get the destination tile for the player, remove the player from its current tile, add it to the destination tile and change the position of the player to reflect its new position.

```
SLGame >> movePlayer: aPlayer distance: anInteger
  ... Your code ...
```

We suspect that when we will introduce ladder and snake tiles, we will have to revisit this method because snakes and ladders do not store players just move them around.

About our implementation

The implementation that we propose below for the method `movePlayer: aPlayer distance: anInteger` is not as nice as we would like it to be. Why? Because it does not give a chance to the tiles to extend this behavior and our experience tells us that we will need it when we will introduce the snake and ladder. We will discuss that when we will arrive there.

```
SLGame >> movePlayer: aPlayer distance: anInteger
  | targetTile |
  targetTile := self tileFor: aPlayer atDistance: anInteger.
  (self tileOfPlayer: aPlayer) removePlayer: aPlayer.
  targetTile addPlayer: aPlayer.
  aPlayer position: targetTile position.
```

18.15 Snakes and ladders

Now we can introduce the two special tiles: the snakes and ladders. Let us analyse a bit their behavior: when a player lands on such a tile, it is automatically moved to another tile. As such, snake and ladder tiles do not need to keep references to players because players never stay on them.

Snakes is really similar to ladders: we could just have a special kind of tiles to manage them. Now we will define two separate classes so that we can add extra behavior. Remember creating a class is cheap. One behavior we will

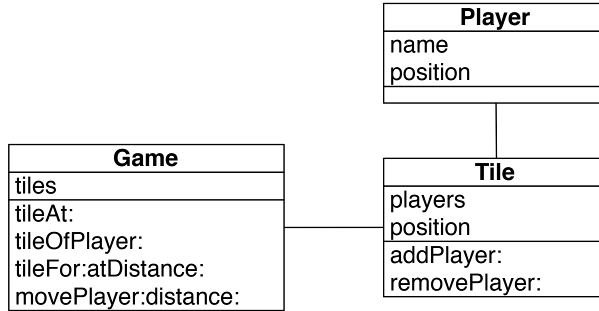


Figure 18-6 Current simple design: three classes with a player acting a simple object.

implement is a different printed version so that we can identify the kind of tile we have.

At the beginning of the chapter we used `->` for ladders and `<-` for snakes.

```
[1<Jill><Jack>][2->6][3][4][5][6][7->9][8][9][10][5<-11][12]
```

18.16 A hierarchy of tiles

We have now our default tile and two kinds of different *active* tiles. Now we will split our current tile class to be able to reuse a bit of its state and behavior with the new tiles. Our current tile class will then be one of the leaves of our hierarchy tree.

To factor the behavior of the active tiles, we will introduce a new class named `ActiveTile`. Once we will be done, we should have a hierarchy as the one presented in the Figure 18-7.

Let us start create the hierarchy.

Split Tile class in two

Let us do the following actions:

- Using the class refactoring "insert superclass" (click on the `SLTile` and check the class refactoring menu), introduce a new superclass to `SLTile`. Name it `SLAbstractTile`.
- Run the tests and they should pass.
- Using the class instance variable refactoring "pull up", push the position instance variable
- Run the tests and they should pass.

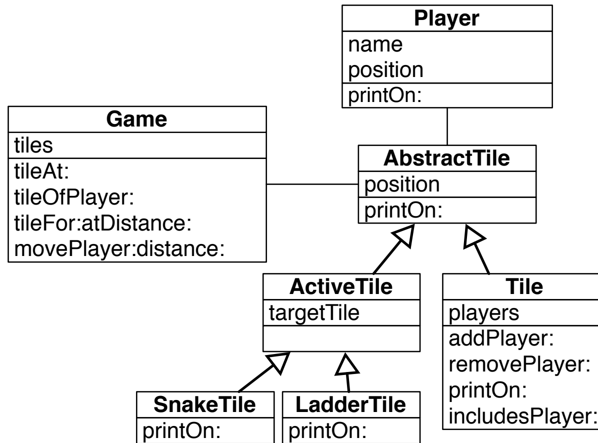


Figure 18-7 A hierarchy of tiles.

- Using the method refactoring ”push up”, push the methods `position` and `position:`.
- Run the tests and they should pass.

What you see is that we did not execute the actions randomly but we want to control that each step is under control using the tests.

Here are the classes and methods `printOn:`.

```
Object subclass: #SLAbstractTile
  instanceVariableNames: 'position'
  classVariableNames: ''
  package: 'SnakesAndLadders'
```

Define a `printOn:` method so that all the subclasses can be displayed in the board by their position.

```
SLAbstractTile >> printOn: aStream
  aStream << '['.
  position printOn: aStream.
  aStream << ']'
```

```
SLAbstractTile subclass: #SLTile
  instanceVariableNames: 'players'
  classVariableNames: ''
  package: 'SnakesAndLadders'
```

Adding snake and ladder tiles

Now we can add a new subclass to `SLAbstractTile`.

```
SLAbstractTile subclass: #SLActiveTile
  instanceVariableNames: 'targetTile'
  classVariableNames: ''
  package: 'SnakesAndLadders'
```

We add a method `to:` to set the destination tile.

```
SLActiveTile >> to: aTile
  targetTile := aTile
```

Then we add the two new subclasses of `SLActiveTile`

```
SLActiveTile subclass: #SLSnakeTile
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'SnakesAndLadders'
```

```
SLSnakeTile >> printOn: aStream

  aStream << '['.
  targetTile position printOn: aStream.
  aStream << '<-'.
  position printOn: aStream.
  aStream << ']'
```

```
SLActiveTile subclass: #SLLadderTile
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'SnakesAndLadders'
```

This is fun to see that the order when to print the position of the tile is different between the snakes and ladders.

```
SLLadderTile >> printOn: aStream

  aStream << '['.
  position printOn: aStream.
  aStream << '->'.
  targetTile position printOn: aStream.
  aStream << ']'
```

We did on purpose not to ask you to define tests to cover the changes. This exercise should show you how long sequence of programming without adding new tests expose us to potential bugs. They are often more stressful.

So let us add some tests to make sure that our code is correct.

```
SLTileTest >> testPrintingLadder

  | tile |
  tile := SLLadderTile new position: 2; to: (SLTile new position: 6).
  self assert: tile printString equals: '[2->6]'
```

```
SLTileTest >> testPrintingSnake

  | tile |
  tile := SLSnakeTile new position: 11; to: (SLTile new position: 5).
  self assert: tile printString equals: '[5<-11]'
```

Run the tests and they should pass. Save your code. Take a rest!

18.17 New printing hook

When we look at the printing situation we see code duplication logic. For example, we always see at least the repetition of the first and last expression.

```
SLTile >> printOn: aStream

  aStream << '['.
  position printOn: aStream.
  players do: [ :aPlayer | aPlayer printOn: aStream ].
  aStream << ']'
```

```
SLLadderTile >> printOn: aStream

  aStream << '['.
  position printOn: aStream.
  aStream << '->'.
  targetTile position printOn: aStream.
  aStream << ']'
```

Do you think that we can do better? What would be the solution?

In fact what we would like is to have a method that we can reuse and that handles the output of `'[ ]'`. And in addition we would like to have another method for the contents between the parentheses and that we can specialize it. This way each class can define its own behavior for the inside part and reuse the parenthesis part.

This is what you will do now. Let us split the `printOn:` method of the class `SLAbstractTile` in two methods:

- a new method named `printInsideOn:` just printing the position, and
- the `printOn:` method using this new method.

```
SLAbstractTile >> printInsideOn: aStream

  position printOn: aStream
```

Now define the method `printOn:` to produce the same behavior as before but calling the message `printInsideOn:`.

```
SLAbstractTile >> printOn: aStream
  ... Your code ...
```

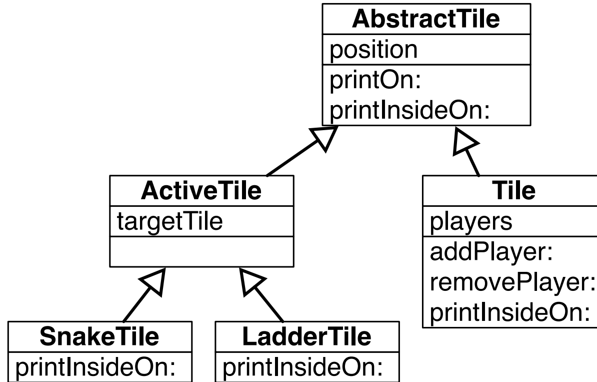


Figure 18-8 Introducing `printInsideOn:` as a new hook.

Run your tests and they should pass. You may have noticed that this is normal because none of them is covering the abstract tile. We should have been more picky on our tests.

What you should see is that we will have only one method defining the behavior of representing the surrounding of a tile and this is much better if one day we want to change it.

18.18 Using the new hook

Now you are ready to express the printing behavior of `SLTile`, `SLSnake` and `SLLadder` in a much more compact fashion. Do not forget to remove the `printOn:` methods in such classes, else they will hide the new behavior (If you do not get why you should read again the chapter on inheritance). You should get the situation depicted as in Figure 18-8.

Here is our definition for the `printInsideOn:` method of the class `SLTile`.

```
SLTile >> printInsideOn: aStream

  super printInsideOn: aStream.
  players do: [ :aPlayer | aPlayer printOn: aStream ].
```

What you should see is that we are invoking the default behavior (from the class `SLAbstractTile`) using the `super` pseudo-variable and we enrich it with the information of the players.

Define the one for the `SLLadderTile` class and the one for `SLSnakeTile`.

```
SLLadderTile >> printInsideOn: aStream
  ... Your code ...
```

```
SLSnakeTile >> printInsideOn: aStream
  ... Your code ...
```

super does not have to be the first expression

Now we show you our definition of `printInsideOn:` for the class `SLSnakeTile`. Why do we show it? Because it shows you that an expression invoking an overriden method can be placed anywhere. It does not have to be the first expression of a method. Here it is the last one.

```
SLSnakeTile >> printInsideOn: aStream

  targetTile position printOn: aStream.
  aStream << '<-'.
  super printInsideOn: aStream
```

Do not forget to run your tests. And they should all pass.

18.19 About hooks and templates

If we look at what we did. We created what is called a Hook/Template.

- The template method is the `printOn:` method. It defines a context of the execution of the hook methods.
- The `printInsideOn:` message is the hook that get specialized for each subclass. It happens in the context of a template method.

What you should see is that the `printOn:` message is also a hook of the `printString` message. There the `printString` method is creating a context and send the message `printOn:` which gets specialized.

The second point that we want to stress is that we turned expressions into a self-message. We transformed the expressions `position printOn: aStream` into `self printInsideOn: aStream` and such simple transformation created a point of variation extensible using inheritance. Note that the expression could have been a lot more complex.

Finally what is important to realize is that even `position printOn: aStream` creates a variation point. Imagine that we have multiple kind of positions, this expression will invoke the corresponding method on the object that is currently referred to by `position`. Such position objects could or not be organized in a hierarchy as soon as they offer a similar interface. So each message is in fact a variation point in a program.

18.20 Snake and ladder declaration

Now we should add to the game some messages to declare snake and ladder tiles. We propose to name the messages `setLadderFrom:to:` and `setSnake-`

From:to:. Now let us write a test and make sure that it fails before starting.

```
SLGameTest >> testFullGamePrintString

  | game |
  game := SLGame new tileNumber: 12.
  game
    setLadderFrom: 2 to: 6;
    setLadderFrom: 7 to: 9;
    setSnakeFrom: 11 to: 5.
  self
    assert: game printString
    equals: '[1][2->6][3][4][5][6][7->9][8][9][10][5<-11][12]'
```

Define the method setSnakerFrom:to: that takes two positions, the first one is the position of the tile and the second one is the position of the target. Pay attention that the message to: of the active tiles expects a tile and not a position.

```
SLGame >> setSnakeFrom: aSourcePosition to: aTargetPosition
  ... Your code ...
```

```
SLGame >> setLadderFrom: aSourcePosition to: aTargetPosition
  ... Your code ...
```

Run your tests! And save your code.

18.21 Better tile protocol

Now we should define what should happen when a player lands on an active tiles (snake or ladder). Indeed for the normal tiles, we implemented that the player change its position, then the origin tile loses the player and the receiving tile gains the player.

We implemented such behavior in the method movePlayer: aPlayer distance: anInteger shown below. We paid attention that a player cannot be in two places at the same time: we remove it from its tile, then move it to its destination.

```
SLGame >> movePlayer: aPlayer distance: anInteger
  | targetTile |
  targetTile := self tileFor: aPlayer atDistance: anInteger.
  (self tileOfPlayer: aPlayer) removePlayer: aPlayer.
  targetTile addPlayer: aPlayer.
  aPlayer position: targetTile position.
```

At that moment we said that we did not like too much this implementation. And now this is the time to understand why and do improve the situation.

First it would be good that the behavior to manage the entering and leaving of a tile would be closer to the objects performing it. We have two solutions:

we could move it to the tile or to the player class. Second we should take another factor into play: different tiles have different behavior; normal tiles manage players and active tiles are placing players on their target tile and they do not manage players. Therefore it is more interesting to define a variation point on the tile because we will be able to exploit it for normal and active tiles.

We propose to define two methods on the tile: one to accept a new player named `acceptPlayer:` and to release a player named `releasePlayer:`. Let us rewrite `movePlayer: aPlayer distance: anInteger` with such methods.

```
SLTile >> acceptPlayer: aPlayer
  self addPlayer: aPlayer.
  aPlayer position: position.
```

The use in this definition of self messages or direct instance variable access is an indication that definition belongs to this class. Now we define the method `releasePlayer:` as follows:

```
SLTile >> releasePlayer: aPlayer
  self removePlayer: aPlayer
```

Defining the method `releasePlayer:` was not necessary but we did it because it is more symmetrical.

Now we can redefine `movePlayer: aPlayer distance: anInteger`.

```
SLGame >> movePlayer: aPlayer distance: anInteger
  | targetTile |
  targetTile := self tileFor: aPlayer atDistance: anInteger.
  (self tileOfPlayer: aPlayer) releasePlayer: aPlayer.
  targetTile acceptPlayer: aPlayer.
```

All the tests should pass. And this is the power of test driven development, we change the implementation of our game and we can verify that we did not change its behavior.

Another little improvement

Now we can improve the definition of `acceptPlayer:`. We can implement its behavior partly on `SLAbstractTile` and partly on `SLTile`. This way the definition of the methods are closer to the definition of the instance variables and the state of the objects.

```
SLAbstractTile >> acceptPlayer: aPlayer
  aPlayer position: position
```

```
SLTile >> acceptPlayer: aPlayer
  super acceptPlayer: aPlayer.
  self addPlayer: aPlayer
```

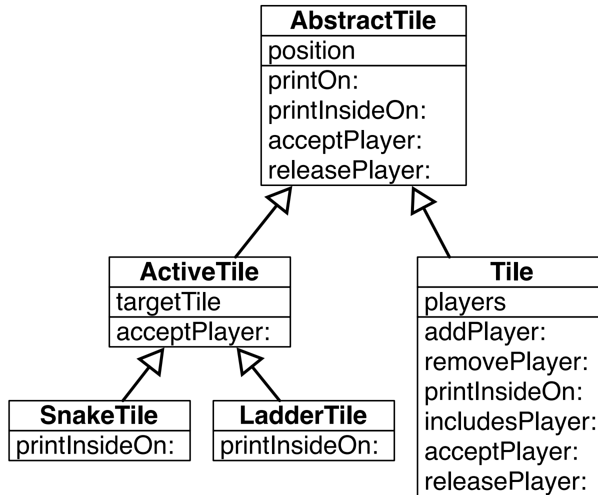


Figure 18-9 acceptPlayer: and releasePlayer: new message.

Note that we change the order of execution by invoking the superclass behavior first (using `super acceptPlayer: aPlayer`) because we prefer to invoke first the superclass method, because we prefer to think that a subclass is extending an existing behavior.

To be complete, we define that `releasePlayer:` does nothing on `SLAbstractTile`. We define it to document the two faces of the protocol. Figure 18-9 shows the situation.

```
SLAbstractTile >> releasePlayer: aPlayer
  "Do nothing by default, subclasses may modify this behavior."
```

18.22 Active tile actions

Now we are ready to implement the behavior of the active tiles. But.... yes we will write a test first. What we want to test is that when a player lands on a snake it falls back on the target and that the original tile does not have this player anymore. This is what this test expresses.

```
SLGameTest >> testPlayerStepOnASnake

  | jill game |
  game := SLGame new
      tileNumber: 12;
      setLadderFrom: 2 to: 6;
      setLadderFrom: 7 to: 9;
      setSnakeFrom: 11 to: 5.
```

```
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  game movePlayer: jill distance: 10.
  self assert: jill position equals: 5.
  self assert: (game tileAt: 1) players isEmpty.
  self assert: ((game tileAt: 5) includesPlayer: jill).
```

Now we just have to implement it!

```
SLActiveTile >> acceptPlayer: aPlayer
  ... Your code ...
```

There is nothing to do for the message `releasePlayer:`, because the player is never added to the active tile. Once you are done run the tests and save.

18.23 Alternating players

We are nearly finished with the game. First we should manage that each turn a different player is playing and that the game finishes when the current player lands on the final tile.

We would like to be able to:

- make the game play in automatic mode
- make the game one step at the time so that humans can play.

The logic for the automatic play can be expressed as follows:

```
play
  [ self isNotOver ] whileTrue: [
    self playPlayer: (self currentPlayer) roll: 6 atRandom ]
```

Until the game is finished, the game identifies the current player and plays this player for a given number given by a die of six faces. The expression `6 atRandom` selects randomly a number between 1 and 6.

18.24 Player turns and current player

The game does not keep track of the players and their order. We will have to support it so that each player can play in alternance. It will also help us to compute the end of the game. Given a turn, we should identify the current player.

The following test verifies that we obtain the correct player for a given turn.

```
SLGameTest >> testCurrentPlayer

  | jack game jill |
  game := SLGame new tileNumber: 12.
  jack := SLPlayer new name: 'Jack'.
```

```
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jack; addPlayer: jill.
  game turn: 1.
  self assert: game currentPlayer equals: jack.
  game turn: 2.
  self assert: game currentPlayer equals: jill.
  game turn: 3.
  self assert: game currentPlayer equals: jack.
```

You should add two instance variables `players` and `turn` to the `SLGame` class.

Then you should initialize the two new instance variables adequately: the `players` instance variable to an OrderedCollection and the `turn` instance variable to zero.

```
SLGame >> initialize
  ... Your code ...
```

You should modify the method `addPlayer:` to add the player to the list of players as shown by the method below.

```
SLGame >> addPlayer: aPlayer
  aPlayer position: 1.
  players add: aPlayer.
  (tiles at: 1) addPlayer: aPlayer
```

We define also the setter method `turn:` to help us for the test. This is where you see that it would be good in Pharo to have the possibility to write tests inside the class and not to be forced to add a method definition just for a test but SUnit does not allow such behavior. One approach to resolve this, and ensuring only test code makes use of `turn:`, is to use class extensions. We make `turn:` belong to the `*SnakesAndLadders-Test` protocol. In this way if we only load the `SnakesAndLadders` package then it will not include any test specific methods.

```
SLGame >> turn: aNumber
  turn := aNumber
```

18.25 How to find the logic of currentPlayer?

Now we should define the method `currentPlayer`. We will try to show you how we brainstorm and experiment when we are looking for an algorithm or even the logic of a simple method.

Imagine a moment that we have two players Jack-Jill. The turns are the following ones: Jack 1, Jill 2, Jack 3, Jill 4, Jack 5.....

Now we know that we have two players. So using this information, at turn 5, the rest of the division of 5 by 2, gives us 1 so this is the turn of the first

player. At turn 4, the rest of the division of 5 by 2 is zero so we take the latest player: Jill.

Here is an expression that shows the result when we have two players and we use the division.

```
(1 to: 10) collect: [ :each | each -> (each \\ 2) ]
> {1->1. 2->0. 3->1. 4->0. 5->1. 6->0. 7->1. 8->0. 9->1. 10->0}
```

Here is an expression that shows the result when we have three players and we use the division.

```
(1 to: 10) collect: [ :each | each -> (each \\ 3) ]
> {1->1. 2->2. 3->0. 4->1. 5->2. 6->0. 7->1. 8->2. 9->0. 10->1}
```

What you see is that each time we get 0, it means that this is the last player (second in the first case and third in the second).

This is what we do with the following method. We compute the rest of the division. We obtain a number between 0 and the player number minus one. This number indicates the index of the number in the `players` ordered collection. When it is zero it means that we should take the latest player.

```
SLGame >> currentPlayer

  | rest playerIndex |
  rest := (turn \\ players size).
  playerIndex := (rest isZero
          ifTrue: [ players size ]
          ifFalse: [ rest ]).
  ^ players at:   playerIndex
```

Run your tests and make sure that they all pass and save.

18.26 Game end

Checking for the end of the game can be implemented in at least two ways:

- the game can check if any of the player is on the last tile.
- or when a player lands on the last tile, its effect is to end the game.

We will implement the first solution but let us write a test first.

```
SLGameTest >> testIsOver

  | jack game |
  game := SLGame new tileNumber: 12.
  jack := SLPlayer new name: 'Jack'.
  game addPlayer: jack.
  self assert: jack position equals: 1.
  game movePlayer: jack distance: 11.
  self assert: jack position equals: 12.
```

```
  self assert: game isOver.
```

Now define the method `isOver`. You can use the `anySatisfy:` message which returns true if one of the elements of a collection (the receiver) satisfies a condition. The condition is that a player's position is the number of tiles (since the last tile position is equal to the number of tiles).

```
SLGame >> isOver
  ... Your code ...
```

Alternate solution

To implement the second version, we can introduce a new tile `SLEndTile`. Here is the list of what should be done:

- define a new class.
- redefine the `acceptPlayer:` to stop the game. Note that it means that the tile should have a reference to the game. This should be added to this special tile.
- initialize the last tile of the game to be an instance of such a class.

18.27 Playing one move

Before automating the play of the game we should make sure that a die roll will not bring our player outside the board.

Here is a simple test covering the situations.

```
SLGameTest >> testCanMoveToPosition

  | game |
  game := SLGame new tileNumber: 12.
  self assert: (game canMoveToPosition: 8).
  self assert: (game canMoveToPosition: 12).
  self deny: (game canMoveToPosition: 13).
```

Define the method `canMoveToPosition:`. It takes as input the position of the potential move.

```
SLGame >> canMoveToPosition: aNumber
  ... Your code ...
```

Playing one game step

Now we are finally ready to finish the implementation of the game. Here are two tests that check that the game can play a step correctly, i.e., picking the correct player and moving it in the correct place.

```
SLGameTest >> testPlayOneStep

  | jill jack game |
  game := SLGame new tileNumber: 12.
  jack := SLPlayer new name: 'Jack'.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  game addPlayer: jack.
  self assert: jill position equals: 1.
  game playOneStepWithRoll: 3.
  self assert: jill position equals: 4.
  self assert: (game tileAt: 1) players size equals: 1.
  self assert: ((game tileAt: 4) includesPlayer: jill)
```

```
SLGameTest >> testPlayTwoSteps

  | jill jack game |
  game := SLGame new tileNumber: 12.
  jack := SLPlayer new name: 'Jack'.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  game addPlayer: jack.
  game playOneStepWithRoll: 3.
  game playOneStepWithRoll: 2.
  "nothing changes for jill"
  self assert: jill position equals: 4.
  self assert: ((game tileAt: 4) includesPlayer: jill).
  "now let us verify that jack moved correctly to tile 3"
  self assert: (game tileAt: 1) players size equals: 0.
  self assert: jack position equals: 3.
  self assert: ((game tileAt: 3) includesPlayer: jack)
```

Here is a possible implementation of the method `playOneStepWithRoll:`.

```
SLGame >> playOneStepWithRoll: aNumber

  | currentPlayer |
  turn := turn + 1.
  currentPlayer := self currentPlayer.
  Transcript show: currentPlayer printString, 'drew ', aNumber
    printString, ': '.
  (self canMoveToPosition: currentPlayer position + aNumber)
    ifTrue: [ self movePlayer: currentPlayer distance: aNumber ].
  Transcript show: self; cr.
```

Now we can verify that when a player lands on a ladder it is getting up.

```
SLGameTest >> testPlayOneStepOnALadder

  | jill jack game |
  game := SLGame new
```

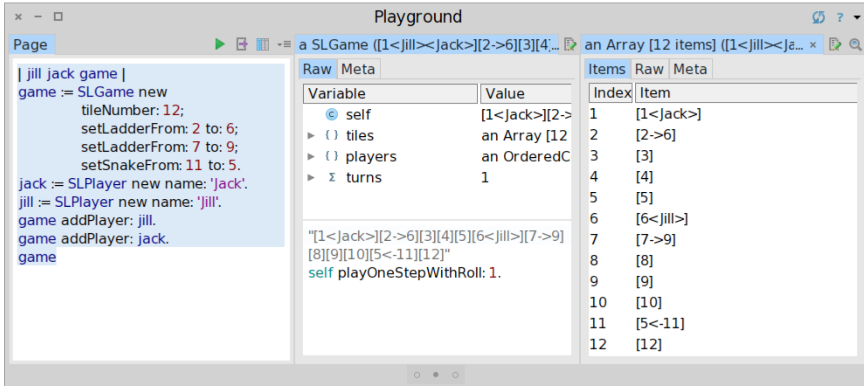


Figure 18-10 Playing step by step inside the inspector.

```
        tileNumber: 12;
        setLadderFrom: 2 to: 6;
        setLadderFrom: 7 to: 9;
        setSnakeFrom: 11 to: 5.
  jack := SLPlayer new name: 'Jack'.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  game addPlayer: jack.
  game playOneStepWithRoll: 1.
  self assert: jill position equals: 6.
  self assert: (game tileAt: 1) players size equals: 1.
  self assert: ((game tileAt: 6) includesPlayer: jill).
```

You can try this method inside an inspector and see the result of the moves displayed in the transcript as shown in Figure 18-10.

```
| jill jack game |
game := SLGame new
      tileNumber: 12;
      setLadderFrom: 2 to: 6;
      setLadderFrom: 7 to: 9;
      setSnakeFrom: 11 to: 5.
jack := SLPlayer new name: 'Jack'.
jill := SLPlayer new name: 'Jill'.
game addPlayer: jill.
game addPlayer: jack.
game inspect
```

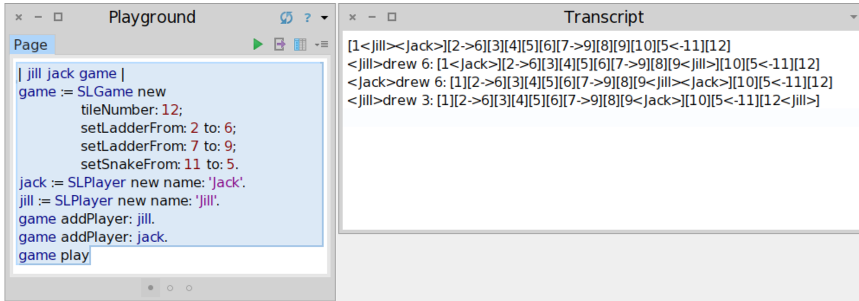


Figure 18-11 Automated play.

18.28 Automated play

Now we can can define the `play` method as follows and use it as shown in Figure 18-11.

```
SLGame >> play

  Transcript show: self; cr.
  [ self isOver not ] whileTrue: [
    self playOneStepWithRoll: 6 atRandom ]
```

Some final tests

We would like to make sure that the player is not moved when it does not land on the last tile or that the game is finished when one player lands on the last tile. Here are two tests covering such behavior.

```
SLGameTest >> testPlayOneStepOnExactFinish

  | jill jack game |
  game := SLGame new
      tileNumber: 12;
      setLadderFrom: 2 to: 6;
      setLadderFrom: 7 to: 9;
      setSnakeFrom: 11 to: 5.
  jack := SLPlayer new name: 'Jack'.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  game addPlayer: jack.

  game playOneStepWithRoll: 11.
  "jill lands on the finish tile!"
  self assert: jill position equals: 12.
  self assert: (game tileAt: 1) players size equals: 1.
  self assert: ((game tileAt: 12) includesPlayer: jill).
```

```
  self assert: game isOver.
```

```
SLGameTest >> testPlayOneStepOnInexactFinish

  | jill jack game |
  game := SLGame new
      tileNumber: 12;
      setLadderFrom: 2 to: 6;
      setLadderFrom: 7 to: 9;
      setSnakeFrom: 11 to: 5.
  jack := SLPlayer new name: 'Jack'.
  jill := SLPlayer new name: 'Jill'.
  game addPlayer: jill.
  game addPlayer: jack.
    "jill moves"
  game playOneStepWithRoll: 9.
  self assert: jill position equals: 10.
  self assert: ((game tileAt: 10) includesPlayers: jill).
  "jack moves"
  game playOneStepWithRoll: 2.
  "jill tries to move but in fact stays at her place"
  game playOneStepWithRoll: 5.
  self assert: jill position equals: 10.
  self assert: ((game tileAt: 10) includesPlayer: jill).
  self deny: game isOver.
```

18.29 Variations

As you see this single game has multiple variations. Here are some of the ones you may want to implement:

- A player who lands on an occupied tile must go back to its originating tile.
- If you roll a number higher than the number of tiles needed to reach the last square, you must continue moving backwards from the end.

You will see that such extensions can be implemented in different manner. We suggest to avoid conditions but define objects responsible for this behavior and its variations.

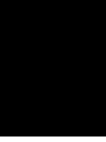
18.30 Conclusion

This chapter went step by step to the process of getting from requirements to an actual implementation covered by tests.

This chapter shows that design is an iterative process. What is also important is that without tests we would be a lot more worried about breaking something without be warned immediately. With tests we were able to change

some parts of the design and rapidly make sure that the previous specification still hold.

This chapter shows that identifying objects and their interactions is not always straightforward and multiple designs are often valid.



CHAPTER 19

TinyChat: a fun and small chat client/server

Pharo allows the definition of a REST server in a couple of lines of code thanks to the Teapot package by zeroflag, which extends the superb HTTP client/server Zinc developed by BetaNine and was given to the community. The goal of this chapter is to make you develop, in five small classes, a client/server chat application with a graphical client. This little adventure will familiarize you with Pharo and show the ease with which Pharo lets you define a REST server. Developed in a couple of hours, TinyChat has been designed as a pedagogical application. At the end of the chapter, we propose a list of possible improvements.

TinyChat has been developed by O. Auverlot and S. Ducasse with a lot of fun.

19.1 Objectives and architecture

We are going to build a chat server and one graphical client as shown in Figure 19-1.

The communication between the client and the server will be based on HTTP and REST. In addition to the classes `TCServer` and `TinyChat` (the client), we will define three other classes: `TCMessage` which represents exchanged messages (as a future exercise you could extend TinyChat to use more structured elements such as JSON or STON (the Pharo object format), `TCMessageQueue` which stores messages, and `TCConsole` the graphical interface.

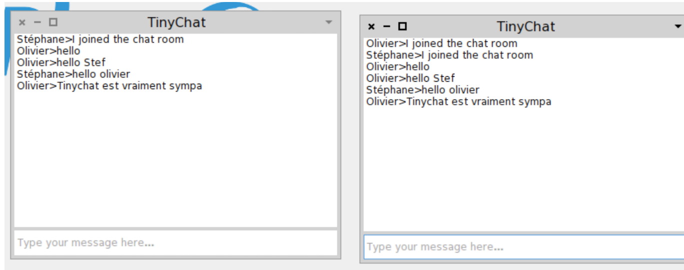


Figure 19-1 Chatting with TinyChat.

19.2 Loading Teapot

We can load Teapot using the Configuration Browser, which you can find in the Tools menu item of the main menu. Select Teapot and click "Install Stable". Another solution is to use the following script:

```
Gofer it
    smalltalkhubUser: 'zeroflag' project: 'Teapot';
    configuration;
    loadStable.
```

Now we are ready to start.

19.3 Message representation

A message is a really simple object with a text and sender identifier.

Class TCMessage

We define the class `TCMessage` in the package `TinyChat`.

```
Object subclass: #TCMessage
  instanceVariableNames: 'sender text separator'
  classVariableNames: ''
  category: 'TinyChat'
```

The instance variables are as follows:

- `sender`: the identifier of the sender,
- `text`: the message text, and
- `separator`: a character to separate the sender and the text.

Accessor creation

We create the following accessors:

```
TCMessage >> sender
  ^ sender

TCMessage >> sender: anObject
  sender := anObject

TCMessage >> text
  ^ text

TCMessage >> text: anObject
  text := anObject
```

19.4 Instance initialisation

Each time an instance is created, its `initialize` method is invoked. We redefine this method to set the separator value to the string >.

```
TCMessage >> initialize
  super initialize.
  separator := '>'.
```

Now we create a class method named `from:text:` to instantiate a message (a class method is a method that will be executed on a class and not on an instance of this class):

```
TCMessage class >> from: aSender text: aText
  ^ self new sender: aSender; text: aText; yourself
```

The message `yourself` returns the message receiver: this way we ensure that the returned object is the new instance and not the value returned by the `text:` message. This definition is equivalent to the following:

```
TCMessage class >> from: aSender text: aText
  | instance |
  instance := self new.
  instance sender: aSender; text: aText.
  ^ instance
```

19.5 Converting a message object into a string

We add the method `printOn:` to transform a message object into a character string. The model we use is sender-separator-text-crlf. Example: 'john>hello !!!'. The method `printOn:` is automatically invoked by the method `printString`. This method is invoked by tools such as the debugger or object inspector.

```
TCMessage >> printOn: aStream

  aStream
    << self sender; << separator;
    << self text; << String crlf
```

19.6 Building a message from a string

We also define two methods to create a message object from a plain string of the form: `'olivier>tinychat is cool'`.

First we create the method `fromString:` filling up the instance variables of an instance. It will be invoked like this: `TCMessage new fromString: 'olivier>tinychat is cool'`, then the class method `fromString:` which will first create the instance.

```
TCMessage >> fromString: aString
  "Compose a message from a string of this form 'sender>message'."
  | items |
  items := aString subStrings: separator.
  self sender: items first.
  self text: items second.
```

You can test the instance method with the following expression `TCMessage new fromString: 'olivier>tinychat is cool'`.

```
TCMessage class >> fromString: aString
  ^ self new
    fromString: aString;
    yourself
```

When you execute the following expression `TCMessage fromString: 'olivier>tinychat is cool'` you should get a message. We are now ready to work on the server.

19.7 Starting with the server

For the server, we are going to define a class to manage a message queue. This is not really mandatory but it allows us to separate responsibilities.

Storing messages

Create the class `TCMessageQueue` in the package *TinyChat-Server*.

```
Object subclass: #TCMessageQueue
  instanceVariableNames: 'messages'
  classVariableNames: ''
  category: 'TinyChat-server'
```

The messages instance variable is an ordered collection whose elements are instances TCMessage. An OrderedCollection is a collection which dynamically grows when elements are added to it. We add an instance initialize method so that each new instance gets a proper messages collection.

```
TCMessageQueue >> initialize
  super initialize.
  messages := OrderedCollection new.
```

Basic operations on message list

We should be able to add, clear the list, and count the number of messages, so we define three methods: add:, reset, and size.

```
TCMessageQueue >> add: aMessage
  messages add: aMessage

TCMessageQueue >> reset
  messages removeAll

TCMessageQueue >> size
  ^ messages size
```

List of messages from a position

When a client asks the server about the list of the last exchanged messages, it mentions the index of the last message it knows. The server then answers the list of messages received since this index.

```
TCMessageQueue >> listFrom: aIndex
  ^ (aIndex > 0 and: [ aIndex <= messages size])
    ifTrue: [ messages copyFrom: aIndex to: messages size ]
    ifFalse: [ #() ]
```

Message formatting

The server should be able to transfer a list of messages to its client given an index. We add the possibility to format a list of messages (given an index). We define the method formattedMessagesFrom: using the formatting of a single message as follows:

```
TCMessageQueue >> formattedMessagesFrom: aMessageNumber

  ^ String streamContents: [ :formattedMessagesStream |
    (self listFrom: aMessageNumber)
      do: [ :m | formattedMessagesStream << m printString ]
    ]
```

Note how the streamContents: lets us manipulate a stream of characters.

19.8 The Chat server

The core of the server is based on the Teapot REST framework. It supports the sending and receiving of messages. In addition this server keeps a list of messages that it communicates to clients.

TCServer class creation

We create the class `TCServer` in the *TinyChat-Server* package.

```
Object subclass: #TCServer
  instanceVariableNames: 'teapotServer messagesQueue'
  classVariableNames: ''
  category: 'TinyChat-Server'
```

The instance variable `messagesQueue` represents the list of received and sent messages. We initialize it like this:

```
TCServer >> initialize
  super initialize.
  messagesQueue := TCMessageQueue new.
```

The instance variable `teapotServer` refers to an instance of the Teapot server that we will create using the method `initializePort:`

```
TCServer >> initializePort: anInteger
  teapotServer := Teapot configure: {
    #defaultOutput -> #text.
    #port -> anInteger.
    #debugMode -> true
  }.
  teapotServer start.
```

The HTTP routes are defined in the method `registerRoutes`. Three operations are defined:

- GET `messages/count`: returns to the client the number of messages received by the server,
- GET `messages/<id:IsInteger>`: the server returns messages from an index, and
- POST `/message/add`: the client sends a new message to the server.

```
TCServer >> registerRoutes
  teapotServer
    GET: '/messages/count' -> (Send message: #messageCount to: self);
    GET: '/messages/<id:IsInteger>' -> (Send message: #messagesFrom:
     to: self);
    POST: '/messages/add' -> (Send message: #addMessage: to: self)
```

Here we express that the path `message/count` will execute the message `messageCount` on the server itself. The pattern `<id:IsInteger>` indicates that the argument should be expressed as number and that it will be converted into an integer.

Error handling is managed in the method `registerErrorHandlers`. Here we see how we can get an instance of the class `TeaResponse`.

```
TCServer >> registerErrorHandlers
  teapotServer
    exception: KeyNotFound -> (TeaResponse notFound body: 'No such
     message')
```

Starting the server is done in the class method `TCServer class>>startOn:` that gets the TCP port as argument.

```
TCServer class >> startOn: aPortNumber
  ^self new
    initializePort: aPortNumber;
    registerRoutes;
    registerErrorHandlers;
    yourself
```

We should also offer the possibility to stop the server. The method `stop` stops the teapot server and empties the message list.

```
TCServer >> stop
  teapotServer stop.
  messagesQueue reset.
```

Since there is a chance that you may execute the expression `TCServer startOn:` multiple times, we define the class method `stopAll` which stops all the servers by iterating over all the instances of the class `TCServer`. The method `TCServer class>>stopAll` stops each server.

```
TCServer class >> stopAll
  self allInstancesDo: #stop
```

19.9 Server logic

Now we should define the logic of the server. We define a method `addMessage` that extracts the message from the request. It adds a newly created message (instance of class `TCMessage`) to the list of messages.

```
TCServer >> addMessage: aRequest
  messagesQueue add: (TCMessage from: (aRequest at: #sender) text:
    (aRequest at: #text)).
```

The method `messageCount` gives the number of received messages.

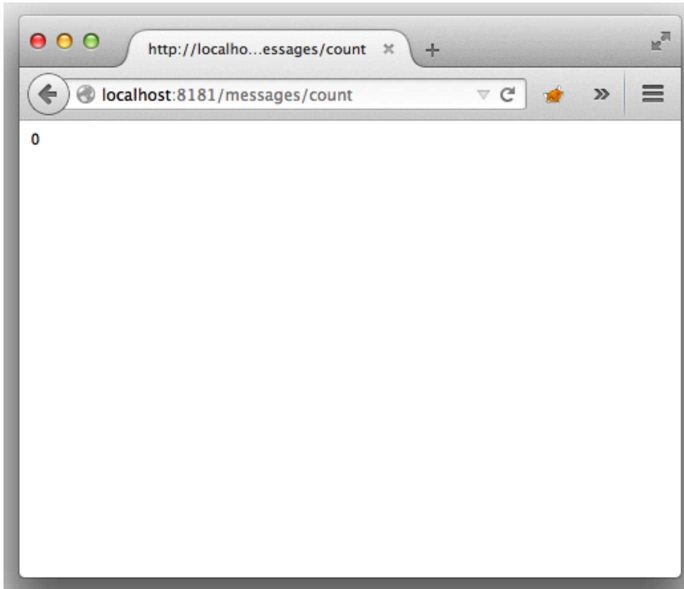


Figure 19-2 Testing the server.

```
TCServer >> messageCount
  ^ messagesQueue size
```

The method `messageFrom:` gives the list of messages received by the server since a given index (specified by the client). The messages returned to the client are a string of characters. This is definitively a point to improve - using string is a poor choice here.

```
TCServer >> messagesFrom: request
  ^ messagesQueue formattedMessagesFrom: (request at: #id)
```

Now the server is finished and we can test it. First let us begin by starting it:

```
  TCServer startOn: 8181
```

Now we can verify that it is running either with a web browser (Figure 19-2), or with a Zinc expression as follows:

```
ZnClient new url: 'http://localhost:8181/messages/count' ; get
```

Shell lovers can also use the curl command:

```
  curl http://localhost:8181/messages/count
```

We can also add a message the following way:

```
ZnClient new
  url: 'http://localhost:8181/messages/add';
  formAt: 'sender' put: 'olivier';
  formAt: 'text' put: 'Super cool ce tinychat' ; post
```

19.10 The client

Now we can concentrate on the client part of TinyChat. We decomposed the client into two classes:

- `TinyChat` is the class that defines the connection logic (connection, send, and message reception),
- `TCConsole` is a class defining the user interface.

The logic of the client is:

- During client startup, it asks the server the index of the last received message,
- Every two seconds, it requests from the server the messages exchanged since its last connection. To do so, it passes to the server the index of the last message it got.

TinyChat class

We now define the class `TinyChat` in the package `TinyChat-client`.

```
Object subclass: #TinyChat
  instanceVariableNames: 'url login exit messages console
    lastMessageIndex'
  classVariableNames: ''
  category: 'TinyChat-client'
```

This class defines the following instance variables:

- url that contains the server url,
- login a string identifying the client,
- messages is an ordered collection containing the messages read by the client,
- lastMessageIndex is the index of the last message read by the client,
- exit controls the connection. While exit is false, the client regularly connects to the server to get the unread messages
- console refers to the graphical console that allows the user to enter and read messages.

We initialize these variables in the following instance `initialize` method.

```
TinyChat >> initialize
  super initialize.
  exit := false.
  lastMessageIndex := 0.
  messages := OrderedCollection new.
```

HTTP commands

Now, we define methods to communicate with the server. They are based on the HTTP protocol. Two methods will format the request. One, which does not take an argument, builds the requests `/messages/add` and `/messages/-count`. The other has an argument used to get the message given a position.

```
TinyChat >> command: aPath
  ^'{1}{2}' format: { url . aPath }

TinyChat >> command: aPath argument: anArgument
  ^'{1}{2}/{3}' format: { url . aPath . anArgument asString }
```

Now that we have these low-level operations we can define the three HTTP commands of the client as follows:

```
TinyChat >> cmdLastMessageID
  ^ self command: '/messages/count'

TinyChat >> cmdNewMessage
  ^self command: '/messages/add'

TinyChat >> cmdMessagesFromLastIndexToEnd
  "Returns the server messages from my current last index to the
    last one on the server."
  ^ self command: '/messages' argument: lastMessageIndex
```

Now we can create commands but we need to emit them. This is what we look at now.

19.11 Client operations

We need to send the commands to the server and to get back information from the server. We define two methods. The method `readLastMessageID` returns the index of the last message received from the server.

```
TinyChat >> readLastMessageID
  | id |
  id := (ZnClient new url: self cmdLastMessageID; get) asInteger.
  id = 0 ifTrue: [ id := 1 ].
  ^ id
```

The method readMissingMessages adds the last messages received from the server to the list of messages known by the client. This method returns the number of received messages.

```
TinyChat >> readMissingMessages
  "Gets the new messages that have been posted since the last
    request."
  | response receivedMessages |
  response := (ZnClient new url: self cmdMessagesFromLastIndexToEnd;
    get).
  ^ response
    ifNil: [ 0 ]
    ifNotNil: [
      receivedMessages := response subStrings: (String crlf).
      receivedMessages do: [ :msg | messages add: (TCMessage
    fromString: msg) ].
      receivedMessages size.
    ].
```

We are now ready to define the refresh behavior of the client via the method refreshMessages. It uses a light process to read the messages received from the server at a regular interval. The delay is set to 2 seconds. (The message fork sent to a block (a lexical closure in Pharo) executes this block in a light process). The logic of this method is to loop as long as the client does not specify to stop via the state of the exit variable.

The expression (Delay forSeconds: 2) wait suspends the execution of the process in which it is executed for a given number of seconds.

```
TinyChat >> refreshMessages
  [
    [ exit ] whileFalse: [
      (Delay forSeconds: 2) wait.
      lastMessageIndex := lastMessageIndex + (self
    readMissingMessages).
      console print: messages.
    ]
  ] fork
```

The method sendNewMessage: posts the message written by the client to the server.

```
TinyChat >> sendNewMessage: aMessage
  ^ ZnClient new
    url: self cmdNewMessage;
    formAt: 'sender' put:  (aMessage sender);
    formAt: 'text' put: (aMessage text);
    post
```

This method is used by the method send: that gets the text written by the user. The string is converted into an instance of TCMessage. The message

is sent and the client updates the index of the last message it knows, then it prints the message in the graphical interface.

```
TinyChat >> send: aString
  "When we send a message, we push it to the server and in addition
    we update the local list of posted messages."

  | msg |
  msg := TCMessage from: login text: aString.
  self sendNewMessage: msg.
  lastMessageIndex := lastMessageIndex + (self readMissingMessages).
  console print: messages.
```

We should also handle the server disconnection. We define the method `disconnect` that sends a message to the client indicating that it is disconnecting and also stops the connecting loop of the server by putting `exit` to true.

```
TinyChat >> disconnect
  self sendNewMessage: (TCMessage from: login text: 'I exited from
    the chat room.').
  exit := true
```

19.12 Client connection parameters

Since the client should contact the server on specific ports, we define a method to initialize the connection parameters. We define the class method `TinyChat class>>connect:port:login:` so that we can connect the following way to the server: `TinyChat connect: 'localhost' port: 8080 login: 'username'`

```
TinyChat class >> connect: aHost port: aPort login: aLogin

  ^ self new
    host: aHost port: aPort login: aLogin;
    start
```

`TinyChat class>>connect:port:login:` uses the method `host:port:login:`. This method just updates the `url` instance variable and sets the `login` as specified.

```
TinyChat >> host: aHost port: aPort login: aLogin
  url := 'http://' , aHost , ':' , aPort asString.
  login := aLogin
```

Finally we define a method `start:` which creates a graphical console (that we will define later), tells the server that there is a new client, and gets the last message received by the server. Note that a good evolution would be to decouple the model from its user interface by using notifications.

```
TinyChat >> start
  console := TCConsole attach: self.
  self sendNewMessage: (TCMessage from: login text: 'I joined the
    chat room').
  lastMessageIndex := self readLastMessageID.
  self refreshMessages.
```

19.13 User interface

The user interface is composed of a window with a list and an input field as shown in Figure 19-1.

```
ComposableModel subclass: #TCConsole
  instanceVariableNames: 'chat list input'
  classVariableNames: ''
  category: 'TinyChat-client'
```

Note that the class `TCConsole` inherits from `ComposableModel`. This class is the root of the user interface logic classes. `TCConsole` defines the logic of the client interface (i.e. what happens when we enter text in the input field...). Based on the information given in this class, the Spec user interface builder automatically builds the visual representation. The `chat` instance variable is a reference to an instance of the client model `TinyChat` and requires a setter method (`chat:`). The `list` and `input` instance variables both require an accessor. This is required by the User Interface builder.

```
TCConsole >> input
  ^ input

TCConsole >> list
  ^ list

TCConsole >> chat: anObject
  chat := anObject
```

We set the title of the window by defining the method `title`.

```
TCConsole >> title
  ^ 'TinyChat'
```

Now we should specify the layout of the graphical elements that compose the client. To do so we define the class method `TCConsole class>>defaultSpec`. Here we need a column with a list and an input field placed right below.

```
TCConsole class >> defaultSpec
  <spec: #default>

  ^ SpecLayout composed
    newColumn: [ :c |
```

```
      c add: #list; add: #input height: 30 ]; yourself
```

We should now initialize the widgets that we will use. The method `initializeWidgets` specifies the nature and behavior of the graphical components. The message `acceptBlock:` defines the action to be executed then the text is entered in the input field. Here we send it to the chat model and empty it.

```
TCConsole >> initializeWidgets

  list := ListModel new.
  input := TextInputFieldModel new
    ghostText: 'Type your message here...';
    enabled: true;
    acceptBlock: [ :string |
      chat send: string.
      input text: '' ].
  self focusOrder add: input.
```

The method `print` displays the messages received by the client and assigns them to the list contents.

```
TCConsole >> print: aCollectionOfMessages
  list items: (aCollectionOfMessages collect: [  :m |  m printString
    ])
```

Note that this method is invoked by the method `refreshMessages` and that changing all the list elements when we add just one element is rather ugly but ok for now.

Finally we need to define the class method `TCConsole class>>attach:` that gets the client model as argument. This method opens the graphical elements and puts in place a mechanism that will close the connection as soon as the client closes the window.

```
TCConsole class >> attach: aTinyChat
  | window |
  window := self new chat: aTinyChat.
  window openWithSpec whenClosedDo: [ aTinyChat disconnect ].
  ^ window
```

19.14 Now chatting

Now you can chat with your server. The example resets the server and opens two clients.

```
| tco tcs |
TCServer stopAll.
TCServer startOn: 8080.
tco := TinyChat connect: 'localhost' port: 8080 login: 'olivier'.
tco send: 'hello'.
tcs := TinyChat connect: 'localhost' port: 8080 login: 'Stef'.
```

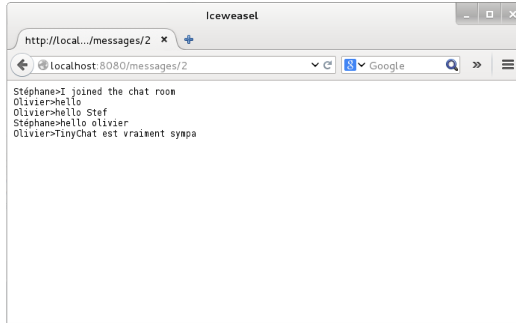


Figure 19-3 Server access.

```
tcs send: 'salut olivier'
```

19.15 Conclusion and ideas for future extensions

We show that creating a REST server is really simple with Teapot. TinyChat provides a fun context to explore programming in Pharo and we hope that you like it. We designed TinyChat so that it favors extensions and exploration. Here is a list of possible extensions.

- Using JSON or STON to exchange information and not plain strings.
- Making sure that the clients can handle a failure of the server.
- Adding only the necessary messages to the list in the graphical client.
- Managing concurrent access in the server message collection (if the server should handle concurrent requests the current implementation is not correct).
- Managing connection errors.
- Getting the list of connected users.
- Editing the delay to check for new messages.

There are probably more extensions and we hope that you will have fun exploring some. The code of the project is available at http://www.smalltalkhub.com/#!/~olivierauverlot/TinyChat.