1

Object Responsibility and Better Encapsulation

1.1 Reducing the coupling between classes

To be a good citizen you as an object should follow as much as possible the following rules:

- Be private. Never let somebody else play with your data.
- Be lazy. Let do other objects your job.
- Be focused. Do only one main task.

While these guidelines are not really formal, one of the main consequences is that this is the responsibility of an object to provide a well defined interface protecting itself from its clients. The other consequence is that by delegating to other objects an object concentrates on a single task and responsibility. We now look how such guidelines can help us to provide better objects in our example.

1.1.1 Current situation

The interface of the packet class is really weak. It just provides free access to its data. The main impact of this weakness is the fact that the clients of the class Packet like Workstation relies on the internal coding of the Packet as shown in the first line of the following method.

```
Workstation>>accept: aPacket

  aPacket addressee = self name
    ifTrue: [  Transcript show: 'A packet is accepted by the Workstation ', self name asString ]
    ifFalse:  [ super accept: aPacket ]
```

As a consequence, if the structure of the class Packet would change, the code of its clients would have to change too. Generalizing such a bad practice would lead to system that are badly coupled and being really difficult to change to meet new requirements.

1.1.2 Solution.

This is the responsibility of a packet to say if the packet is addressed to a particular node or if it was sent by a particular node.

- Define a method named isAddressedTo: aNode in 'testing' protocol that answers if a given packet is addressed to the specified node.
- Define a method named isOriginatedFrom: aNode in 'testing' protocol that answers if a given packet is originated from the specified node.

Once these methods are defined, change the code of all the clients of the class Packet to call them.

1.2 A Question of Creation Responsibility

One of the problem with the previous approach for creating the nodes and the packets is the following: it is the responsibility of the client of the objects to create them well-formed. For example, it is possible to create a node without specifying a name! This is a disaster for our LAN system (create an example method 3, and try it out). The same problem occurs with the packet: it is possible to create a packet without address nor contents.

We will find a solution to these problems.

Exercise: 1. Define a class method named withName: in the class Node (protocol 'instance creation') that creates a new node and assign its name.

```
withName: aSymbol
....
```

Define a class method named withName:nextNode: in the class Node (protocol 'instance creation') that creates a new node and assign its name and the next node in the LAN

```
withName: aSymbol nextNode: aNode
....
```

Note that the first method can simply invoke the second one.

Define a class method named send:to: in the class Packet (protocol 'instance creation') that creates a new Packet with a contents and an address.

```
send: aString to: aSymbol
....
```

Now the problem is that we want to forbid the creation of non-well formed instances of these classes. To do so, we will simply redefine the creation method new so that it will raise an error.

Exercise: 2. Rewrite the new method of the class Node and Packet as the following:

```
new

  self error: 'you should invoke the method... to create a...'
```

However, you have just introduced a problem: the instance creation methods you just wrote in exercise 11 will not work anymore, because they call *new*, and that calling results in an error ! The solution is to rewrite them such as

```
Node class>>withName: aSymbol nextNode: aNode
    ˆ self basicNew initialize name: aSymbol ; nextNode: aNode
```

Do the same for the instance creation methods in class Packet.

Exercise: 3. Update and rerun your tests to make sure that your changes were correct.

Note that the previous code may break if a subclass specialize the nextNode: method does not return the instance. To protect ourslef from possible unexpected extension we add yourself that returns the receiver a the first cascaded message (here name:), here the newly created instance.

```
Node class>>withName: aSymbol nextNode: aNode
    ˆ self basicNew initialize name: aSymbol ; nextNode: aNode ; yourself
```

1.3 Reducing the coupling between classes

To be a good citizen you as an object should follow as much as possible the following rules:

- Be private. Never let somebody else play with your private data.
- Be lazy. Let do other objects your job.
- Be focused. Do only one main task.

While these guidelines are not really formal, one of the main consequences is that this is the responsibility of an object to provide a well defined interface protecting itself from its clients. The other consequence is that by delegating to other objects an object concentrates on a single task and responsibility. We now look how such guidelines can help us to provide better objects in our example.

1.3.1 Current situation

The interface of the Packet class is really weak. It just provides free access to its data. The main impact of this weakness is the fact that the clients of the class Packet like Workstation relies on the internal coding of the Packet as shown in the first line of the following method.

```
Workstation>>accept: aPacket

   aPacket addressee = self name
     ifTrue: [  Transcript show: 'A packet is accepted by the Workstation ', self name asString ]
     ifFalse:  [ super accept: aPacket ]
```

As a consequence, if the structure of the class Packet would change, the code of its clients would have to change too. Generalizing such a bad practice would lead to system that are badly coupled and being really difficult to change to meet new requirements.

1.3.2 Solution.

This is the responsibility of a packet to say if the packet is addressed to a particular node or if it was sent by a particular node.

- Define a method named isAddressedTo: aNode in 'testing' protocol that answers if a given packet is addressed to the specified node.
- Define a method named isOriginatedFrom: aNode in 'testing' protocol that answers if a given packet is originated from the specified node.

Once these methods are defined, change the code of all the clients of the class Packet to call them. You should note that a better interface encapsulates better the private data and the way they are represented. This allows one to locate the change in case of evolution.

1.4 A Question of Creation Responsibility

One of the problems with the first approach for creating the nodes and the packets is the following: it is the responsibility of the client of the objects to create them well-formed. For example, it is possible to create a node without specifying a name! This is a disaster for our LAN system, the node would never reachable, and worse the system would breaks because the assumptions that the name of a node is specified would not hold anymore (insert an anonymous node in Lan and try it out). The same problem occurs with the packet: it is possible to create a packet without address nor contents.

The solution to these problems is to give the responsibility to the objects to create well-formed instances. Several variations are possible:

- When possible, providing default values for instance variable is a good way to provide well-defined instances.
- It is also a good solution to propose a consistent and well-defined creation interface. For example one can only provide an instance creation method that requires the mandatory value for the instance and forbid the creation of other instances.

The class Packet. We investigate the two solutions for the Packet class. For the first solution, the principle is that the creation method (new) should invoke an initialize method. Implement this solution. Just remember that new is sent to classes (a class method) and that initialize is sent to instances (instance method). Implement the method new in a 'instance creation' protocol and initialize in a 'initialize-release' protocol.

```
Packet class>>new

...

Packet>>initialize
  ...
```

The only default value that can have a default value is contents, choose

```
contents = 'no contents'
```

Ideally if each LAN would contain a default trash node, the default address and originator would point to it. We will implement this functionality in a future lesson. Implement first your own solution.

Remarks and Analysis. Note that with this solution it would be convenient to know if a packet contents is the default one or not. For this purpose you could provide the method hasDefaultContents that tests that. You can implement it in a clever way as shown below:

Instead of writing:

```
Packet>>hasDefaultContents

 ^ contents = 'no contents'

Packet>>initialize
 ...

contents := 'no contents'
...
```

You should apply the rule: 'Say only once' and define a new method that returns the default content and use it as shown below:

```
Packet>>defaultContents

   ^ 'no contents'

Packet>>initialize
   ...

   contents := self defaultContent
   ...

Packet>>hasDefaultContent
   ^contents = self defaultContents
```

With this solution, we limit the knowledge to the internal coding of the default contents value to only one method. This way changing it does not affect the clients nor the other part of the class.

1.5 Proposing a creational interface

Packet. We now apply the second approach by providing a better interface for creating packet. For this purpose we define a new creation method that requires a contents and an address.

Define a **class** methods named `send:to:` and `to:` in the class `Packet` (protocol 'instance creation') that creates a new `Packet` with a contents and an address.

```
Packet class>>send: aString to: aSymbol

....

Packet class>>to: aSymbol

....
```

The class Node. Now apply the same techniques to the class `Node`. Note that you already implemented a similar schema that the default value in the previous lessons. Indeed by default instance variable value is `nil` and you already implemented the method `hasNextNode` that to provide a good interface.
Define a **class** method named `withName:` in the class `Node` (protocol 'instance creation') that creates a new node and assign its name.

```
Node class>>withName: aSymbol

....
```

Define a **class** method named `withName:connectedTo:` in the class `Node` (protocol 'instance creation') that creates a new node and assign its name and the next node in the LAN.

```
Node class>>withName: aSymbol connectedTo: aNode

....
```

Note that if to avoid to duplicate information, the first method can simply invoke the second one.

1.6 Forbidding the Basic Instance Creation

One the last question that should be discussed is the following one: should we or not let a client create an instance without using the constrained interface? There is no general answer, it really depends on what we want to express. Sometimes it could be convenient to create an uncompleted instance for debugging or user interface interaction purpose.

Let us imagine that we want to ensure that no instance can be created without calling the methods we specified. We simply redefine the creation method new so that it will raise an error. Rewrite the `new` method of the class `Node` and `Packet` as the following:

```
Node class>>new

  self error: 'you should invoke the method... to create a...'
```

However, you have just introduced a problem: the instance creation methods you just wrote in the previous exercise will not work anymore, because they call `new`, and that calling results in an error! Propose a solution to this problem.

1.6.1 Remarks and Analysis.

A first solution could be the following code:

```
Node class>>withName: aSymbol connectedTo: aNode

    ˆ super new initialize name: aSymbol ; nextNode: aNode
```

However, even if the semantics permits such a call using super with a different method selector than the containing method one, it is a bad practice. In fact it implies an implicit dependency between two different methods in different classes, whereas the super normal use links two methods with the same name in two different classes. It is always a good practice to invoke the own methods of an object by using self. This conceptually avoids to link the class and its superclass and we can continue to consider the class as self contained.

The solution is to rewrite the method such as:

```
Node class>>withName: aSymbol connectedTo: aNode

ˆ  self basicNew initialize name: aSymbol ; nextNode: aNode
```

In Smalltalk there is a convention that all the methods starting with 'basic' should not be overridden. basicNew is the method responsible for always providing an newly created instance. You can for example browse all the methods starting with 'basic*' and limit yourself to Object and Behavior.

You can do the same for the instance creation methods in class Packet.

1.7 Protecting yourself from your children

The following code is a possible way to define an instance creation method for the class Node.

```
Node class>>withName: aSymbol

 ˆ self new name: aSymbol
```

We create a new instance by invoking new, we assign the name of the node and then we return it. One possible problem with such a code is that a subclass of the class Node may redefine the method name: (for example to have a persistent object) and return another value than the receiver (here the newly created instance). In such a case invoking the method withName: on such a class would not return the new instance. One way to solve this problem is the following:

```
Node class>>withName: aSymbol

 | newInstance |
 newInstance := self new.
 NewInstance name: aSymbol.
 ˆ newInstance
```

This is a good solution but it is a bit too much verbose. It introduces extra complexity by the the extra temporary variable definition and assignment. A good Smalltalk solution for this problem is illustrated by the following code and relies on the use of the yourself message.

```
Node class>>withName: aSymbol

  ˆ self new name: aSymbol ; yourself
```

yourself specifies that the receiver of the first message involved into the cascade (name: here and not new) is return. Guess what is the code of the yourself method is and check by looking in the library if your guess is right.