The Advanced Object-Oriented Design Mooc Companion

S. Ducasse

April 1, 2024

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

Contents

Illustrations

CHAPTER 1

Introduction

This book accompanies the *'The Advanced Object-Oriented Design and Development with Pharo'* MOOC (AOOD) that is freely available at https://advanced-design-mooc.pharo.org. The AOOD Mooc is not about Pharo but about object-oriented design and presents a large set of topics on this subject from basic to advanced points. From this perspective, AOOD Mooc extends the Pharo Mooc https://mooc.pharo.org. AOOD Mooc uses Pharo and readers may want to watch some videos of the Pharo Mooc to better understand the code. The module 0 of the AOOD Mocc proposes a selection of lectures to get up to speed with Pharo.

The current document contains a collection of design exercises at different levels of guidance and difficulties.

- The first part of the book is dedicated to teachers and presents different setups in which this material has been used.
- The second part proposes several little projects to exercise double dispatch, Command, and Visitor Design Patterns.
- The third part proposes some unguided extensions to the previous projects as well as some exercises that are also loosely described to give space for variations.
- The fourth part presents some game ideas that you are invited to design and build with Bloc the new graphic layer of Pharo and that can also benefit from the micro-framework Myg.

CHAPTER 2

Module Exercises

The module is composed of 10 modules and an optional one around Pharo. There is no direct mapping one to one mapping between the exercises and the modules. For example the Module 2 around tests is the second one so that the learners can try to write tests in any of the exercises. In addition when we could all the guided exercises contained tests.

2.1 Default flow

By default we suggest

- follow the book order and perform the exercises. Note that the Tamagotchi (Chapter 13) and Civilization (Chapter 14) projects are less guided on purpose and this can give you more space to design solutions.
- then follow some unguided projects as described in Chapter 15. Note that each of the unguided projects requires that the extended project has been performed. For the LAN we are giving the possibility to load the code since the exercise is simple.
- implement one of the games proposed in Chapter 16 or a nice extension of the Microdown project proposed in Chapter 17.

2.2 Exercises proposition per Module

Module 1.

Chapter 5 (A basic LAN application) You can also follow Chapter 10 (Finding the North with Compass) since it presents some basic ways to use late binding.

Module 2.

Chapter 6 (Crafting a simple embedded DSL with Pharo). Another interesting exercise is Chapter 11 (A little expression interpreter) since it illustrates well object-oriented design and will be extended in Module 6.

Module 3.

To exercise Hooks you can work on the Chapter 15 section related to the LAN extension. You can also check the extension for the die rolling exercise.

Module 4.

To prepare the exercises of Module 6 we suggest doing Chapter 11 (A little expression interpreter). Here you can for example follow Chapters 13 (Tamagotchi Mechanics) and 14 (Civilization).

Module 5.

To exercise the Command pattern do the Chapter 9 (A little saturn PathFinder) and you can continue with Chapter 10 (Finding the North with Compass). In addition, Chapter 13 is a nice exercise to use the State pattern.

Module 6.

In this module you should do the exercises of Chapter 7 (Stone paper scissors) as well as the one of Chapter 12 (Understanding visitors). From then on, you can do the unguided exercises about AST in Chapter 15 (Little unguided projects) and the Microdown extensions of Chapter 17 (Microdown projects).

Module 7, 8, 9, 10.

From then on you should work on games (see Chapter 16 or Microdown extensions (see Chapter 17).

Part I

Teacher corner

CHAPTER 3

About Pharo and Moocs

In this short chapter, we want to stress some key properties of Pharo and the teaching material we developed around it.

3.1 A truly excellent and pedagogical language

Pharo is a truly excellent language and environment to teach object-oriented programming. Here are some reasons you will certainly recognize if you ever programmed or taught Pharo.

- **Tiny Syntax**. The full syntax of Pharo fits on half of a postcard (see Listing 3-1). A good element can learn Pharo syntax in a couple of hours and be productive in a couple of days.
- **A uniform object model without any exception**. Everything is an object and there is not a single exception. The execution model is the same at all levels. There are no special rules for class methods.
- **Ready to use out of the box**. Pharo is ready to use in 2 min. There is no need to configure Eclipse. The SUnit framework is ready to use.
- **Highly immersive IDE.** Developers get immersed in a sea of objects. They can interact with live objects and this is a huge win to deeply understanding the object-oriented metaphor. They talk to their objects.
- **Gorgeous TDD support**. Pharo is the best environment to develop following Test-Driven Design. In addition, Pharo supports eXtreme Test-Driven Design that takes full advantage of Test-Driven Design and liveness of Pharo. You literally develop in your debugger in the context of a test execution.

Listing 3-1 The full syntax of Pharo

```
exampleWithNumber: x
  "This method illustrates the complete syntax."
  <aMethodAnnotation>
  | y |
  true & false not & (nil isNil)
    ifFalse: [ self halt ].
  y := self size + super size.
  { #($a #a 'a' 1 1.0) . 1 + 2 }
    do: [ :each | Transcript
      show: (each class name);
      show: (each printString); show:''].
  ^ x<y
```

- **Advanced integrated tools**. Pharo comes out of the box with a large set of tools: It offers refactorings, code critics, and test coverage. It supports microcommits: all the changes and versions of all compiled methods are one click away. Developers can navigate back in their history, and run the tests to validate a change revert. It has a full integration with Git.

3.2 Some testimonies

While this book is the companion exercise for the *Advanced Object-Oriented Design*, it reuses a couple of concepts from the Pharo mooc. The Pharo Mooc is of high quality and received excellent feedback.

Here are some testimonies of the Pharo mooc.

In french

J'ai trouvé ça très intéressant, beaucoup plus que prévu ! je regrette de ne pas m'y être mis plus tôt. J'ai enfin l'impression de vraiment faire de la POO ! Ou à l'inverse je me rend que je n'en faisais pas vraiment... - Anonymous, 2019

Mooc absolument remarquable. Superbe voyage autour de Pharo. Le paradigme objet qui (re)prend enfin du sens ! - Anonymous - 01/03/2019

In English

I have just completed week seven of the Pharo Mooc (beginner and object oriented tracks) I am starting a redo of the Mooc with the web track (TinyBlog project). I have already learned so much ! I have spent the last 20 years or so in software development and, following this Mooc, I realized I hadn't really grasped the essence of object oriented design. - Anonymous

Really one of the best mooc I have ever attended. And I have attended quite a few (openSAP, openHPI). As an old fashioned ABAP developer I want to be reborn as Pharo developer in my next life :-) - Anonymous

Hi! I finished the MOOC some weeks ago and I would like to congratulate everybody involved! After a decade+ of Python programming I think I found my new favorite language :). I'm making a small Teapot server for Slack command bots, I'm goona push it to Github (yay Iceberg), if anyone is interested. - EduardoPadoan

I just completed the @pharoproject Mooc the best investment I have ever made of my time. MAQBOOL Hey all - I've just finished the Mooc - thanks for an excellent course and a thouroughly interesting look at a new way to program :smile: Looking forward to starting to play with Pharo on some upcoming ideas I've had. - Tieryn

As much as I thought I understand object-orientation, it is very clear NOW that without a truly useable Smalltalk, which Pharo is, it is impossible to really understand and exercise object-orientation. Thank you all soooo much. - Mike D. 06/12/2020

Hey all - I've just finished the Mooc - thanks for an excellent course and a thouroughly interesting look at a new way to program :smile: Looking forward to starting to play with Pharo on some upcoming ideas I've had - Anonymous

A general comment I wanted to make is that the MOOC so far has been great. Impressed with the quality and content, and grateful that it is available and free. Many thanks! - Aryeh

IMHO the videos were very well done. I would even say shockingly well done... for a bunch of programmers who are supposed to be clueless about design - SeanDeNigris - 10/26/2017

The more I'm learning about @pharoproject the more I appreciate it's beauty and simplicity, finally, object-oriented programming is done right - MAQBOOL

3.3 Conclusion

The two Moocs around Pharo are the results of more than 60 years teaching object-oriented programming. Our experience shows that developers do not program the same once they follow such Moocs.

CHAPTER 4

Lectures

Pharo Summer School was a game-changer! Before, I struggled with OOP, but now I get it. The presentations were polished, and the instructors made complex concepts simple and unforgettable. I've discovered the amazing possibilities of Pharo. It's way more than just a summer school. I highly recommend it to anyone in the programming world! Thank you to all the instructors and the Pharo Consortium for this amazing opportunity! I hope to see you all again soon! Q.

The material of the '*Advanced Object-Oriented Design*' MOOC available at http://advanced-mooc.pharo.org has been refined over several years and used in international summer schools. It has been used for many successful lectures at different levels and configurations.

We describe some possible pedagogical objectives, some key insights, and different setups to take advantage of such material. The conventions to refer to the slides and videos of the two Moocs following the ones of the original names:

- W (which stands for Week) is used in the Pharo Mooc and
- M (which stands for Module) is used in the Advanced Object-Oriented Mooc.

4.1 Possible pedagogical objectives

The material proposed by the MOOC can be used to give lectures on the following topics:

- Basic object-oriented programming (excluding for example Law of Demeter, Typing, Design Patterns)
- Test-Driven Design

- Basic Object-Oriented Design (creating Hooks and Template, designing reusable objects)
- Advanced object-oriented design (including Design Patterns and type consideration)

In addition and as we will show below, such goals can be easily enriched with soft skills such as:

- How to find information?
- How to report activity?
- How to get some help by asking questions in forums or online chat?
- How to report problems?

The *'Advanced Object-Oriented Design'* Mooc complements the Pharo Mooc available at http://mooc.pharo.org

4.2 Lecture: Essence of OO design from 1/2 to 1 day

For a lecture on object-oriented design of three hours we usually present the essence of dispatch, key points about inheritance, and core extension mechanism. We use the following material from http://advanced-mooc.pharo.org:

- M1-1 Essence of Dispatch: Taking Pharo Booleans as Example
- M1-2 Essence of Dispatch: Let the receiver decide
- M1-3 Inheritance Basics
- M1-4 Inheritance and Lookup: Self - Understand lookup once for all
- M1-5 About super
- M3-2 Message Sends are Plans for Reuse
- M3-3 Hooks and Template: One of the cornerstones of OOP

As a bonus, we often give a little introduction to unit testing (M2-1 Test 101: The minimum) you should know and a live demo of eXtreme Test-Driven Design as shown in (M2-4 Xtreme Test Driven Development: Getting a productivity boost).

4.3 Lecture: Pharo in 1 day

Even if teaching Pharo is not directed and supported by the Mooc http://advanced-mooc.pharo.org, the Pharo Mooc http://mooc.pharo.org can be used and combined to produce a simple introduction to Pharo.

Here is the material that we use:

- W1S05 PharoSyntaxInANutshell
- W1S06 Blocks
- W1S07 Basic-Blocks-Loops
- W1S07 BasicBooleansAndCondition
- W1S08 Loops
- W1S10 ClassAndMethodDefinition
- W2S01 Messages
- W2S02 Messages-ForTheJavaProgrammers
- W2S03 Basic-Variables
- W2S03 Messages-Precedence
- W2S04 Messages-Sequence
- W2S07 CharacterStringSymbol
- W2S08 Basic-ArraySetOrderedCollection
- W2S10 Iterators
- W2S11 Streams

Here are some extra lectures students may want to follow:

- W2S05 ParenthesisVsSquareBrackets
- W2S06 Yourself
- W2S09 UnderstandingMistakes
- W3S00 TeapotAsAPretext

We often also present some knowledge about tests. For this we use:

- M2-1 Test 101: The minimum you should know
- M2-3 Test-Driven Development
- M2-4 Xtreme Test-Driven Development: Getting a productivity boost

4.4 Lecture: Basic OOP in 1/2 to 1 day

When we are teaching object-oriented basics we use the lectures available as PreSequel of the Pharo mooc and available under http://rmod-pharo-mooc.lille.inria.fr/MOOC/2018-PreSequelOOP-FR/ and http://rmod-pharo-mooc.lille.inria.fr/MOOC/2018-PreSequelOOP-EN/

The five following lectures cover the basics.

- First Look At Class Object Methods

- What is An Object?
- What is A Class?
- Method Vs Messages
- Object-Oriented Paradigm

Since we believe that tests are also key knowledge we often also present:

- M2-1 Test 101: The minimum you should know
- M2-3 Test-Driven Development
- M2-4 Xtreme Test-Driven Development: Getting a productivity boost

Finally, if time allows it we present some of the lectures presented in Section 4.2. We focus in particular on the following 3 lectures:

- M1-1 Essence of Dispatch: Taking Pharo Booleans as Example
- M1-2 Essence of Dispatch: Let the receiver decide
- M3-2 Message Sends are Plans for Reuse

4.5 Lecture: Pharo and Object-oriented design in 2 days

Often we start by one day on Pharo and we take another day to show the essence of OOD. In addition, we often show the inspector and the fact that developers can extend to adapt to the model they manipulate. We show that having adaptable tools is a productivity boost. Finally we stress that eXtreme Test-Driven development as promoted by Pharo is really powerful since a test specifies a clear context that is then used to code in debugger. Coding in the debugger is not just fixing a bug, it is having all the information (the context) at hand in a specific configuration that helps focusing on points specified by a test. This is extremelly powerful. It combines the power of Test-Driven Design with the deep immersive interaction of Pharo.

4.6 Lecture: Advanced object-oriented design lecture example

During several years we used and developed around this material the following lecture whose description is available at https://github.com/pharo-mooc/AdvancedOODesignWithTDD

The lectures length is around 10 slots of 4 hours covering a mix of lectures and labs. The level of the students was Master 1. Their level is heterogeneous.

The objectives of the lectures were:

- Revisiting basic object-oriented programming
- Test-Driven Design
- Advanced object-oriented design
- Reverse engineering
- Test quality with mutation testing
- Soft skills:
  - reporting: how to report activities?
  - how to ask help, how to find information?
  - how to learn fast a new language?
  - how to report problems?

Setup

During the first two weeks they had to learn Pharo and report how they were learning, and what were their strategies to learn fast. We gave no lectures on Pharo. We only listed the following resources:

- http://mooc.pharo.org
- https://discord.gg/QewZMZa
- http://books.pharo.org
- http://books.pharo.org
- https://scg.unibe.ch/download/oorp/OORP.pdf

Each week, the students have to watch a couple of videos and one or two lectures are given by us. During the first 6 weeks, each week they had to do some exercises following some scripts or exercises such as the ones reported later in this book. Then in the subsequent weeks, students are asked to develop a little board game as shown in Chapter @games. In the new version of the lecture, we plan to make the lecture centered around the game design so that students get more time.

Calendar

Our calendar is the following one:

- 01 Week: Test introduction
- 02 Week: OOP refresh
- 03 Week: Reverse engineering
- 04 Week: Test Quality

- 05 Week: Presentation – *Presentations on Learning and data structure analysis*
- 06 Week: Hook and templates
- 07 Week: Double dispatch – Examen

Break

- 08 Week Visitor
- 09 Week Composite
- 10 Week Inheritance
- 11 Week Types

– Exam

4.7 Conclusion

Our experience shows that Pharo and its Moocs are excellent material for teaching a large range of lectures focusing on key and perennial knowledge that can be mapped to any hype language. In addition, an aspect that is often not stress enough if the large overhead that teachers are facing when using language such as Java. Pharo is a out-of-the-box running environment.

All the materials around Pharo are release under permissive licenses and the Pharo community (with exercism, discord channels) is welcoming newbies.

Part II

Guided Exercices

CHAPTER 5

A basic LAN application

The purpose of this mini-project is to define a little network simulator. If you understand well basic object-oriented concepts, you can skip this part of the book even if it is fun to code a little simulator and in particular its less guided extensions.

From an object-oriented point of view, it is really interesting because it shows that objects encapsulate responsibilities and that inheritance is used to define incremental behavior.

You will define step by step an application that simulates a simple Local Area Network (LAN). You will create several classes: `LNPacket`, `LNNode`, `LNWorkstation`, and `LNPrintServer`. We start with the simplest version of a LAN. In subsequent exercises, we will add new requirements and modify the proposed implementation to take them into account.

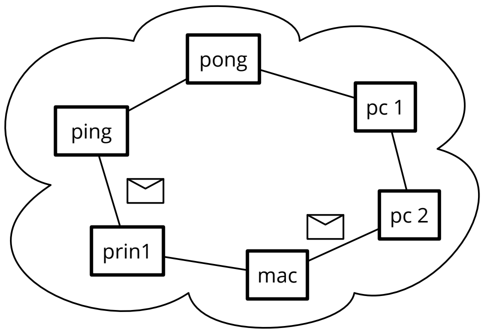


Figure 5-1 An example of a LAN with packets.

5.1 Creating the class LNNode

The class `LNNode` will be the root of all the entities that form a LAN: a printer, a server, and a computer. This class contains the common behavior for all nodes.

As a network is defined as a linked list of nodes, a node should always know its next node. A node should be uniquely identifiable with a name. It is the node's responsibility to send and receive packets of information. In the next section, we will define the class that represents packets.

```
LNNode inherits from Object
Collaborators: LNNode and LNPacket
Responsibility:
  - name (aSymbol) - returns the name of the current node.
  - hasNextNode - tells if the receiver has a next node.
  - accept: aPacket - receives a Packet and processes it. By
    default, it is sent to the next node.
  - send: aPacket - sends a Packet to its next node.
```

Exercise: Create a new package `SimpleLAN`

This package will contain all the code for this project. It will contain the classes for the simulation as well as the classes of the tests.

Exercise: Create a Test class

To help you write tests, we will define a test class.

```
TestCase << #LNNodeTest
  slots: {};
  package: 'SimpleLAN'
```

We will define some test methods as we go.

Exercise: Class creation

Create a subclass of `Object` called `LNNode`, with two instance variables: `name` and `nextNode`.

Exercise: Accessors

Create accessors and mutators for the two instance variables. Document the mutators to inform users that the argument passed to `name:` should be a `Symbol`, and the arguments passed to `nextNode:` should be a `LNNode`. Define the following test to validate such a simple behavior.

```
LNNodeTest >> testName
  | node |
  node := LNNode new.
  node name: #PC1.
  self assert: node name equals: #PC1
```

Exercise: Define the method hasNextNode

Define a method called `hasNextNode` that returns whether the receiver has a next `LNNode` or not. Notice that by default a newly created node does not have a next node. The following test should pass.

```
LNNode >> testHasNextNode
  self deny: LNNode new hasNextNode
```

5.2 Sending/receiving packets

A `LNNode` has two basic messages to send and receive packets.

When a packet is sent to a node, the node has to accept the packet and send it on. Note that with this simple behavior, the packet can loop infinitely in the LAN. We will propose some solutions to this issue later. To implement this behavior, you should add a protocol `send-receive`, and implement the following two methods:

```
LNNode >> accept: aPacket
  "Having received aPacket, send it on. This is the default
    behavior. My subclasses may override me to do something special."

  self send: aPacket
```

```
LNNode >> send: aPacket

  nextNode ifNotNil: [
    self name trace.
    ' sends a packet to: ' trace.
    nextNode name traceCr.
    nextNode accept: aPacket ]
```

Note that:

- `trace` displays in the Transcript the result of sending the message `printOn:` to the receiver.
- `traceCr` has a similar behavior but adds a carriage return at the end.

A little example.

The following snippet shows basic behavior of an open LAN composed of two nodes, Mac and PC1.

```
(LNNode new
    name: 'Mac' ;
    nextNode: (LNNode new name: 'PC1'))
          accept: (LNPacket new addresseeName: 'Mac')

On Transcript:
  Mac sends a packet to: PC1
```

5.3 Better printString

The textual representation of a node is not adequate to debug the code. It only proposes generic information such as ’aLLNode’. We should address this problem. For this, you will redefine the method `printOn:` which is responsible for the textual representation of an object. Now before coding head first, let us specify what output we want. We define a couple of tests.

Let us start from the simplest case: we have a node with a name and a next node. In this case, we want to have the name of the receiver followed by the name of its next node. The following test captures this behavior.

```
LNNode >> testPrintingWithANextNode

  self
    assert: (LNNode new
         name: 'LNNode1';
         nextNode: (LNNode new name: 'PC1')) printString
    equals: 'LNNode1 -> PC1'
```

The second case is when the receiver does not have a next node. In this case, we will use / to indicate it. The following test captures this behavior.

```
LNNode >> testPrintingWithoutNextNode

  self
    assert: (LNNode new
         name: 'LNNode1';
         printString)
    equals: 'LNNode1 -> /'
```

Create an instance method named `printOn:` that puts the class name and name variable on the argument `aStream`. We give a partial definition of the method `printOn:`. Fill this method definition up to make the tests pass.

```
LNNode >> printOn: aStream

  ... Your code here ...
  nextNode
    ifNil: [ aStream nextPutAll: '/' ]
    ifNotNil: [ aStream nextPutAll: nextNode name ]
```

Now there is one case that we should still cover: when the node was not given a name. The following test shows the expected result.

```
LNNode >> testPrintingJustInitializedNode

  self
    assert: LNNode new printString
    equals: 'unamed -> \'
```

From an implementation perspective, we could add a test in the `printOn:` method. There is, however, a better solution. We should make sure that every node has a default value for name. For this, we specialize the method `initialize` on the class `LLNode`. This method is automatically called on object creation.

Define the method `initialize` on the class `LNNode` and make sure that the tests are all passing.

```
LNNode >> initialize

  super initialize.
  ... Your code ...
```

5.4 Creating the class LNPacket

A packet is an object that represents a piece of information that is sent from node to node. The responsibilities of this object are to allow us to define the originator of the packet emission, the address of the receiver, and the contents.

```
LNPacket inherits from Object
Collaborators: LNNode
Responsibility:
  - addresseeName - returns the name of the node to which the packet
     is sent.
  - contents - describes the contents of the message sent.
  - originatorName -  that sent the packet.
```

Exercise: defining class LNPacket

In the `SimpleLAN` package:

- Create a subclass of `Object` called `LNPacket`, with three instance variables: `contents`, `addresseeName`, and `originatorName`.
- Initialize them to some default value (see test below).
- Create accessors and mutators for each of them in the `accessing` protocol.

```
LNPacketTest >> testInitialized

  | p |
  p := LNPacket new.
  self assert: p addresseeName equals: '/'.
  self assert: p originatorName equals: '/'.
  self assert: p contents equals: ''
```

Exercise: Adding `isAddressedTo:`

Define the method `isAddressedTo: aNode` which returns whether a packet is addressed to a given node.

```
LNPacketTest >> testIsAddressedTo

  ^ (LNPacket new addresseeName: 'Mac') isAddressedTo: (LNNode new
    name: 'Mac')
```

Exercise: adding a `printOn:` method

Define the method `printOn: aStream` that puts a textual representation of a `LNPacket` on its argument `aStream`.

Here is a test that you should make sure it passes.

```
LNPacketTest >> testPrintString

  self
    assert: (LNPacket new
         addresseeName: 'Mac';
         contents: 'Pharo is cool';
         yourself) printString
    equals: 'a LNPacket: Pharo is cool sentTo: Mac'
```

5.5 Creating the class `LNWorkstation`

Up until now our simulation only supports simple nodes whose behavior is just to pass the packet they receive around. We will now introduce new kinds of nodes with different behavior.

A workstation is the entry point for new packets onto the LAN network. It can emit packets to other workstations, printers or file servers. Since it is a kind of network node, but provides additional behavior, we will define it as a subclass of `LNNode`. Thus, it inherits the instance variables and methods defined in `LNNode`. Moreover, a workstation has to process packets that are addressed to it, therefore it will specialize the method `accept:`.

```
LNWorkstation inherits from LNNode
Collaborators: Node, Workstation and Packet
Responsibility: (the ones of LNNode)
  - send: aPacket - sends a packet.
  - accept: aPacket - performs an action on packets sent to the
    workstation (e.g. printing in the transcript). For the other
  packets just send them to the following node.
```

Exercise: Define LNWorkstation

In the package `SimpleLAN` create a subclass of `LNNode` called `LNWorkstation` without instance variables.

Exercise: Redefining the method accept:

Define the method `accept: aPacket` so that if the workstation is the destination of the packet, a message is written into the Transcript. When the packets are not addressed to the workstation they are sent to the next node of the current one.

The following two scenarios illustrate the expected behavior. First the one send a packet to the first node.

```
(LNWorkstation new
   name: 'Mac';
   nextNode: (LNNode new
        name: 'PC1';
        nextNode: (LNWorkstation new
            name: 'Mac2';
            yourself);
      yourself) accept: (LNPacket new addresseeName: 'Mac')
```

produces

```
Mac accepted packet
```

The second scenario shows that when the packet is not addressed to a node, it passes it to its next node.

```
LNWorkstation new
  name: 'Mac';
  nextNode: (LNNode new
      name: 'PC1';
      nextNode: (LNWorkstation new
          name: 'Mac2';
          yourself);
```

```
     yourself)) accept: (LNPacket new addresseeName: 'Mac2')
```

produces

```
Mac sends a packet to: PC1
PC1 sends a packet to: Mac2
Mac2 accepted packet
```

About good design.

To implement the behavior of the `accept:` method when packet is not addressed to the workstation, you could copy and paste the code of the `LNNode` class. However, this is a bad practice, decreasing the reuse of code and the ”Say it only once” rule. Indeed if we copy and paste, future changes in the superclass code may not be taken into account. This is why is better to invoke the behavior defined in the superclass and that is currently overridden by using `super`.

Exercise: Defining the method `emit:`

Define the method `emit:` that is responsible for inserting packets in the network in the method protocol `send-receive`. In particular a packet should be marked with its originator and then sent.

```
LNWorkstation >> emit: aLNPacket
    "This is how LNPackets get inserted into the network.
    This is a likely method to be rewritten to permit
    LNPackets to be entered in various ways."

  ... Your code here ...
```

This way we can now write the following scenario:

```
(LNWorkstation new
     name: 'Mac';
     nextNode: (LNNode new
          name: 'PC1';
          nextNode: (LNWorkstation new
               name: 'Mac2';
               yourself);
          yourself)) emit: (LNPacket new addresseeName: 'Mac2')
```

5.6 Creating the class `LNPrinter`

You will now create a class `LNPrinter`, a special node that when it receives packets addressed to it, prints them (on the Transcript). Define the class `LNPrinter`.

```
LNPrinter inherits from LNNode
Collaborators: LNNode and LNPacket
Responsibility:
  - accept: aLNPacket - if the packet is addressed to the printer,
    prints the packet contents else sends the packet
  to the following node.
  - print: aLNPacket - prints the contents of the packet (into the
    Transcript for example).
```

Specialize the method `accept:` on the class `LNPrinter` to print the contents of the packet if necessary.

Illustrating scenario

```
(LNWorkstation new
  name: 'Mac';
  nextNode: (LNNode new
      name: 'PC1';
      nextNode: (LNWorkstation new
            name: 'Mac2';
            nextNode: (LNPrinter new
                name: 'Printer1';
                yourself);
             yourself))) emit: (LNPacket new
     addresseeName: 'Printer1';
     contents: 'Pharo is cool';
     yourself)
```

produces the following trace:

```
Mac sends a packet to: PC1
PC1 sends a packet to: Mac2
Mac2 sends a packet to: Printer1
Pharo is cool
```

5.7 Simulating the LAN

Implement the following method on the class side of the class `LNNode`, in a protocol called `examples`.

```
LNNode class >> simpleLan
  <script>

  | mac pc node1 node2 igPrinter |

  "create the nodes, workstations and printers"
  mac := Workstation new name: 'mac'.
  pc := LNWorkstation new name: 'pc'.
  node1 := LNNode new name: 'Node1'.
```

```
node2 := LNNode new name: 'Node2'.
node3 := LNNode new name: 'Node3'.
igPrinter := LNPrinter new name: 'IGPrinter'.

"connect the different LNNodes."
mac nextNode: node1.
node1 nextNode: node2.
node2 nextNode: igPrinter.
igPrinter nextNode: node3.
node3 nextNode: pc.
pc nextNode: mac.

"create a LNPacket and start simulation"
packet := LNPacket new
  addresseeName: 'IGPrinter';
  contents: 'This LNPacket travelled around to the printer
   IGPrinter'.

mac emit: packet.
```

As you will notice the system does not handle loops, so we will propose a solution to this problem in the future. To break the loop, use Command.

5.8 Conclusion

You created a simple simulator of a local network. In the following chapters, we will revisit such a project to illustrate different points.

CHAPTER 6

Crafting a simple embedded DSL with Pharo

In this chapter, you will develop a simple domain-specific language (DSL) for rolling dice. Players of games such as Dungeons & Dragons are familiar with such DSL. An example of such DSL is the following expression: `2 D20 + 1 D6` which means that we should roll two 20-faces dice and one 6-face die. It is called an embedded DSL because the DSL uses the syntax of the language used to implement it. Here we use the Pharo syntax to implement the Dungeons & Dragons rolling die language.

This little exercise shows how we can (1) simply reuse traditional operators such as `+`, (2) develop an embedded domain-specific language and (3) use class extensions (the fact that we can define a method in another package than the one of the class of the method).

6.1 Getting started

Using the code browser, define a package named `Dice` or any name you like.

Create a test

It is always empowering to verify that the code we write is always working as we are defining it. For this purpose you should create a unit test. Remember unit testing was promoted by K. Beck first in the ancestor of Pharo. Nowadays this is a common practice but it is always useful to remember our roots!

Define the class `DieTest` as a subclass of `TestCase` as follows:

```
TestCase <<#DieTest
  package: 'Dice'
```

What we can test is that the default number of faces of a die is 6.

```
DieTest >> testInitializeIsOk
    self assert: Die new faces equals: 6
```

If you execute the test, the system will prompt you to create a class `Die`. Do it.

Define the class Die

The class `Die` inherits from `Object` and it has an instance variable, `faces` to represent the number of faces one instance will have. Figure 6-1 gives an overview of the messages.

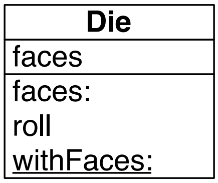


Figure 6-1 A single class with a couple of messages. Note that the method `withFaces:` is a class method.

```
Object <<
  ... Your solution ...
```

In the `initialization` protocol, define the method `initialize` so that it simply sets the default number of faces to 6.

```
Die >> initialize
  ... Your solution ...
```

Do not hesitate to add a class comment.

Now define a method to return the number of faces an instance of `Die` has.

```
Die >> faces
  ^ faces
```

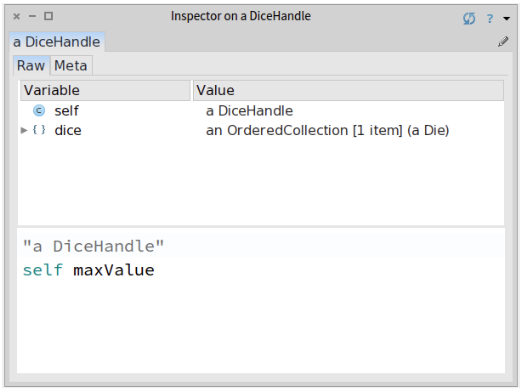
Now your tests should all pass (and turn green).

6.2 Rolling a die

To roll a die you should use the method from Number `atRandom` which draws randomly a number between one and the receiver. For example `10 atRandom` draws number between 1 to 10. Therefore we define the method `roll`:

```
Die >> roll
  ... Your solution ...
```

Now we can create an instance `Die new` and send it the message `roll` and get a result. Do `Die new inspect` to get an inspector and then type in the bottom pane `self roll`. You should get an inspector like the one shown in Figure 6-2. With it you can interact with a die by writing an expression in the bottom pane.



Variable	Value
self	a DiceHandle
dice	an OrderedCollection [1 item] (a Die)

```
"a DiceHandle"
self maxValue
```

Figure 6-2 Inspecting and interacting with a die.

6.3 Creating another test

But better, let us define a test that verifies that rolling a newly created dice with a default 6 faces only returns a value comprised between 1 and 6. This is what the following test method actually specifies.

```
DieTest >> testRolling
  | d |
  d := Die new.
  10 timesRepeat: [ self assert: (d roll between: 1 and: 6) ]
```

Important Often it is better to define the test even before the code it tests. Why? Because you can think about the API of your objects and a scenario that illustrates their correct behavior. It helps you to program your solution.

6.4 Instance creation interface

We would like to get a simpler way to create `Die` instances. For example, we want to create a 20-face die as follows: `Die withFaces: 20` instead of always having to send the new message to the class as in `Die new faces: 20`. Both expressions are creating the same die but one is shorter.

Let us look at it:

- In the expression `Die withFaces:`, the message `withFaces:` is sent to the class `Die`. It is not new, we constantly send the message `new` to `Die` to create instances.
- Therefore we should define a method that will be executed

Let us define a test for it.

```
DieTest >> testCreationIsOk
  self assert: (Die withFaces: 20) faces equals: 20
```

What the test clearly shows is that we are sending a message to the **class** `Die` itself.

Defining a class method

Define the *class* method `withFaces:` as follows:

- Click on the class button in the browser to make sure that you are editing a **class** method.
- Define the method as follows:

```
Die class >> withFaces: aNumber
  "Create and initialize a new die with aNumber faces."
  | instance |
  instance := self new.
  instance faces: aNumber.
  ^ instance
```

Let us explain this method

- The method `withFaces:` creates an instance using the message `new`. Since `self` represents the receiver of the message and the receiver of the message is the class `Die` itself then `self` represents the class `Die`.
- Then the method sends the message `faces:` to the instance and
- Finally returns the newly created instance.

Pay attention that a class method `withFaces:` is sent to a class, and an instance method is sent to the newly created instance `faces:`. Note that the class method could have also named `faces:` or any name we want, it does not matter, it is executed when the receiver is the class `Die`.

If you execute it will not work since we did not yet create the method `faces:`. Now is the time to define it. Pay attention that method `faces:` is sent to an instance of the class `Die` and not the class itself. It is an instance method, therefore make sure that you deselect the class button before editing it.

```
Die >> faces: aNumber
  faces := aNumber
```

Now your tests should run. So even if the class `Die` could implement more behavior, we are ready to implement a die handle.

Important A class method is a method executed in reaction to messages sent to a *class*. It is defined on the class side of the class. In `Die withFaces: 20`, the message `withFaces:` is sent to the class `Die`. In `Die new faces: 20`, the message `new` is sent to the *class* `Die` and the message `faces:` is sent to the *instance* returned by `Die new`.

[Optional] Alternate instance creation definition

In a first reading, you can skip this section. The *class* method definition `withFaces:` above is strictly equivalent to the one below.

```
Die class >> withFaces: aNumber
  ^ self new faces: aNumber; yourself
```

Let us explain it a bit. `self` represents the class `Die` itself. Sending it the message `new`, we create an instance and send it the `faces:` message. And we return the expression. So why do we need the message `yourself`. The message `yourself` is needed to make sure that whatever value the instance message `faces:` returns, the instance creation method we are defining returns the newly created instance. You can try to redefine the instance method `faces:` as follows:

```
Die >> faces: aNumber
  faces := aNumber.
  ^ 33
```

Without the use of `yourself`, `Die withFaces: 20` will return 33. With `yourself` it will return the instance.

The trick is that `yourself` is a simple method defined in `Object` class: The message `yourself` returns the receiver of a message. The use of `;` sends the message to the receiver of the previous message (here `faces:`). The message `yourself` is then sent to the object resulting from the execution of the expression `self new` (which returns a new instance of the class `Die`), as a consequence it returns the new instance.

6.5 First specification of a die handle

Let us define a new class `DieHandle` that represents a die handle. Here is the API that we would like to offer for now (as shown in Figure 6-3). We create a new handle and then add some dice to it.

```
DieHandle new
  addDie: (Die withFaces: 6);
  addDie: (Die withFaces: 10);
  yourself
```

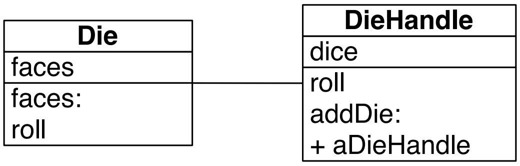


Figure 6-3 A die handle is composed of dice.

Of course we will define tests first for this new class. We define the class `DieHandleTest`.

```
TestCase << #DieHandleTest
  package: 'Dice'
```

Testing a die handle

We define a new test method as follows. We create a new handle and add one die of 6 faces and one die of 10 faces. We verify that the handle is composed of two dice.

```
DieHandleTest >> testCreationAdding
  | handle |
  handle := DieHandle new
      addDie: (Die withFaces: 6);
      addDie: (Die withFaces: 10);
      yourself.
  self assert: handle diceNumber equals: 2.
```

In fact we can do it better. Let us add a new test method to verify that we can even add two dice having the same number of faces.

```
DieHandleTest >> testAddingTwiceTheSameDice
  | handle |
  handle := DieHandle new.
  handle addDie: (Die withFaces: 6).
  self assert: handle diceNumber equals: 1.
  handle addDie: (Die withFaces: 6).
  self assert: handle diceNumber equals: 2.
```

Now that we specified what we want, we should implement the expected class and messages. Easy!

6.6 Defining the DieHandle class

The class `DieHandle` inherits from `Object` and it defines one instance variable to hold the dice it contains.

```
Object << ...
  ... Your solution ...
```

We simply initialize it so that its instance variable `dice` contains an instance of `OrderedCollection`.

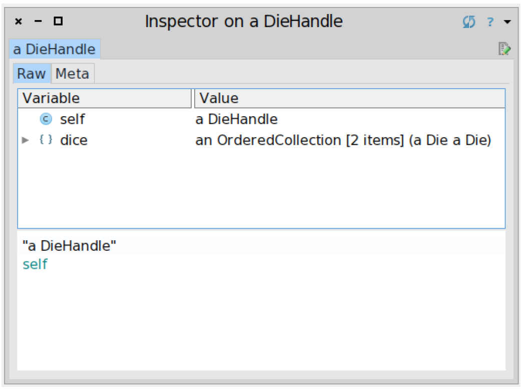
```
DieHandle >> initialize
  ... Your solution ...
```

Then define a simple method `addDie:` to add a die to the list of dice of the handle. You can use the message `add:` sent to a collection.

```
DieHandle >> addDie: aDie
  ... Your solution ...
```

Now you can execute the code snippet and inspect it. You should get an inspector as shown in Figure 6-4

```
DieHandle new
  addDie: (Die withFaces: 6);
  addDie: (Die withFaces: 10);
  yourself
```



Variable	Value
self	a DieHandle
dice	an OrderedCollection [2 items] (a Die a Die)

```
"a DieHandle"
self
```

Figure 6-4 Inspecting a DieHandle.

Finally, we should add the method `diceNumber` to the `DieHandle` class to be able to get the number of dice of the handle. We just return the size of the dice collection.

```
DieHandle >> diceNumber
  ^ dice size
```

Now your tests should run and this is a good moment to save and publish your code.

6.7 Improving programmer experience

Now when you open an inspector you cannot see well the dice that compose the die handle. Click on the `dice` instance variable and you will only get a list of `a Dice` without further information. What we would like to get is something like `a Die (6)` or `a Die (10)` so that at a glance we know the faces a die has.

```
DieHandle new
  addDie: (Die withFaces: 6);
  addDie: (Die withFaces: 10);
  yourself
```

This is the message `printOn:` that is responsible for providing a textual representation of the message receiver. By default, it just prints the name of the class prefixed with `'a'` or `'an'`. So we will enhance the `printOn:` method of the `Die` class to provide more information. Here we simply add the number of faces surrounded by parenthesis. The `printOn:` message is sent with a stream as an argument. It is in this stream that we should add information. We use the message `nextPutAll:` to add several characters to the stream. We concatenate the characters to compose `()` using the message `,` comma defined on collections (and that concatenate collections and strings).

```
Die >> printOn: aStream

  super printOn: aStream.
  aStream nextPutAll: ' (', faces printString, ')'
```

Now in your inspector, you can see effectively the number of faces a die handle has as shown by Figure 6-5 and it is now easier to check the dice contained inside a handle (See Figure 6-6).

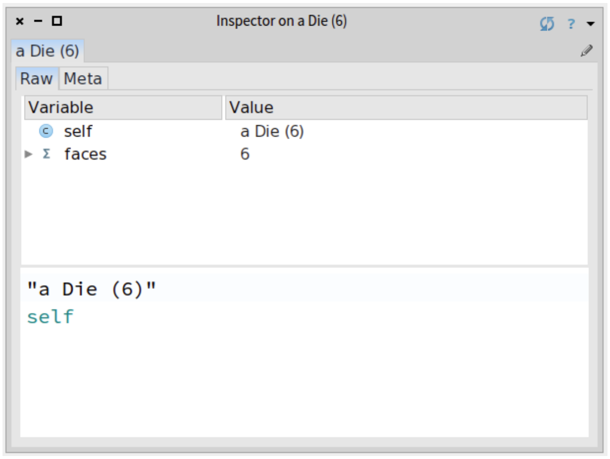
Optimization Remark.

Note that this implementation of `printOn:` is suboptimal since it is creating a separate stream (during the invocation of `faces printString`) instead of reusing the stream passed as an argument. A better solution is to rewrite `printOn:` as follows:

```
Die >> printOn: aStream

  super printOn: aStream.
  aStream nextPutAll: ' ('.
  aStream print: faces.
  aStream nextPutAll: ')'
```

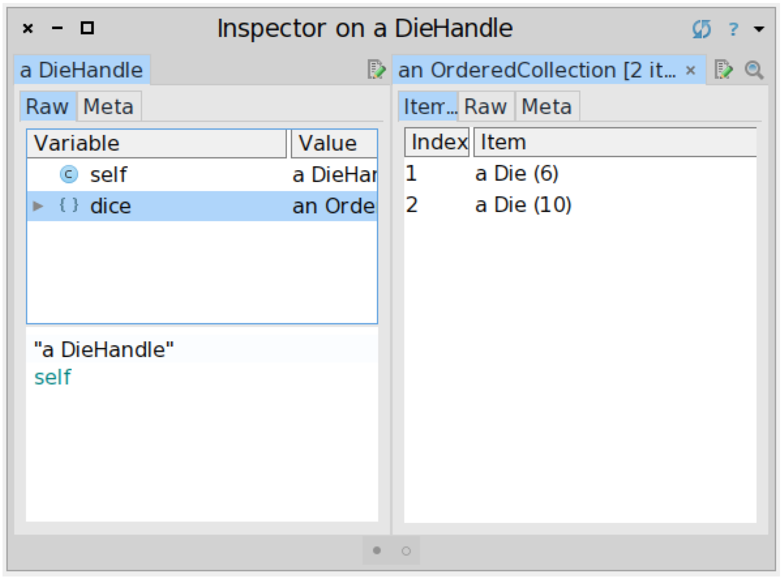
As an exercise we let you browse the methods `printString` on class `Object` and `print:` on class `Stream`.



Variable	Value
self	a Die (6)
faces	6

```
"a Die (6)"
self
```

Figure 6-5 Die details.



Variable	Value
self	a DieHar
dice	an Orde

```
"a DieHandle"
self
```

Index	Item
1	a Die (6)
2	a Die (10)

Figure 6-6 A die handle with more information.

6.8 Rolling a die handle

Now we can define the rolling of a die handle by simply summing the result of rolling each of its dice. Implement the `roll` method of the `DieHandle` class. This method must collect the results of rolling each dice of the handle and sum them.

You may want to have a look at the method `sum` in the class `Collection` or use a simple loop.

```
DieHandle >> roll
  ... Your solution ...
```

Now we can send the message `roll` to a die handle.

```
handle := DieHandle new
    addDie: (Die withFaces: 6);
    addDie: (Die withFaces: 10);
    yourself.
handle roll
```

Define a test to cover such behavior. Rolling a handle of n dice should be between n and the sum of the face number of each die.

```
DieHandleTest >> testRoll
  ... Your solution ...
```

6.9 About Dice and DieHandle API

It is worth spending some time looking at the relationship between `DieHandle` and `Dice`. A die handle is composed of dice. What is an important design decision is that the API of the main behavior (`roll`) is the same for a die or a die handle. You can send the message `roll` to a die or a die handle. This is an important property.

Why? Because it means that from a client's perspective, they can treat the receiver without having to take care about the kind of object it is manipulating. A client just sends the message `roll` to an object and gets back a number (as shown in Figure 6-7). The client is not concerned by the fact that the receiver is composed of a simple object or a complex one. Such design decision supports the *Don't ask, tell* principle.

Important Offering polymorphic API is a tenet of good object-oriented design. It enforces the *Don't ask, tell* principle. Clients do not have to worry about the type of the objects to which they talk to.

For example, we can write the following expression that adds a die and a dieHandle to a collection and collects the different values (we convert the result into an array so that we can print it in the book).

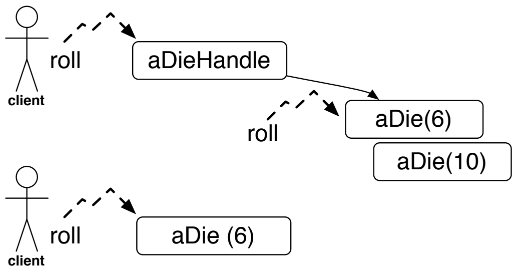


Figure 6-7 A polymorphic API supports the *Don't ask, tell* principle.

```
| col |
col := OrderedCollection new.
col add: (Die withFaces: 20).
col add: (DieHandle new addDie: (Die withFaces: 4); yourself).
(col collect: [:each | each roll]) asArray
>>> #(17 3)
```

About composition

Composite objects such as document objects (a book is composed of chapters, a chapter is composed of sections, a section is composed of paragraphs) often have a more complex composition relationship than the composition between a die and a die handle. Often the composition is recursive in the sense that an element can be the whole: for example, a diagram can be composed of lines, circles, and other diagrams. We will see an example of such composition in the Expression Chapter 11.

6.10 Handle's addition

Now what is missing is the possibility to add several handles together to form a new one. Of course, let's write a test first to be clear on what we mean.

```
DieHandleTest >> testSumOfHandles
  | hd1 hd2 hd3 |
  hd1 := DieHandle new addDie: (Die withFaces: 20); addDie: (Die
    withFaces: 20); yourself.
  hd2 := DieHandle new addDie: (Die withFaces: 10); addDie: (Die
    withFaces: 10); yourself.
  hd3 := hd1 + hd2.
  self assert: hd3 diceNumber equals: 4.
```

We will define a method `+` on the `DieHandle` class. In other languages, this is often not possible or is based on operator overloading. In Pharo `+` is just a message as any other, therefore we can define it in the classes we want.

Now we should ask ourselves what is the semantics of adding two handles. Should we modify the receiver of the expression or create a new one? We preferred a more functional style and chose to create a third one.

The method + creates a new handle then adds the dice of the receiver to it, and then one of the handles is passed as an argument to the message. Finally, we return it.

```
DieHandle >> + aDieHandle
  ... Your solution ...
```

Now we want to be able to execute the method `(2 D20 + 1 D6) roll` nicely and start playing role playing games, of course. So let us see that.

6.11 Role-playing syntax

Now we are ready to offer a syntax following the practice of role role-playing games, i.e., using `2 D20` to create a handle of two dice with 20 faces each. For this purpose we will define class extensions: we will define methods in the class `Integer` but these methods will be only available when the package Dice is loaded.

But first, let us specify what we would like to obtain by writing a new test in the class `DieHandleTest`. Remember to always take any opportunity to write tests. When we execute `2 D20` we should get a new handle composed of two dice and can verify that. This is what the method `testSimpleHandle` is doing.

```
DieHandleTest >> testSimpleHandle
  self assert: 2 D20 diceNumber equals: 2.
```

Verify that the test is not working! It is much more satisfactory to get a test running when it was not working before. Now define the method `D20` with a protocol named `*NameOfYourPackage` (`'*Dice`' if you named your package `'Dice'`). The `*` (star) prefixing a protocol name indicates that the protocol and its methods belong to another package than the package of the class. Here we want to say that while the method `D20` is defined in the class `Integer`, it should be saved with the package `Dice`.

The method `D20` simply creates a new die handle, adds the correct number of dice to this handle, and returns the handle.

```
Integer >> D20
  ... Your solution ...
```

About class extensions

We asked you to place the method `D20` in a protocol starting with a star and having the name of the package (`'*Dice'`) because we want this method to be saved (and packaged) together with the code of the classes we already created (`Die`, `DieHandle`,...) Indeed in Pharo, we can define methods in classes that are not defined in our package. Pharoers call this action a class extension: we can add methods to a class that is not ours. For example `D20` is defined on the class `Integer`. Now such methods only make sense when the package `Dice` is loaded. This is why we want to save and load such methods with the package we created. This is why we are defining the protocol as `'*Dice'`. This notation is a way for the system to know that it should save the methods with the package and not with the package of the class `Integer`.

Now your tests should pass and this is probably a good moment to save your work either by publishing your package and to save your image.

We can do the same for the default dice with different face numbers: 4, 6, 10, and 20. But we should avoid duplicating logic and code. So first we will introduce a new method `D:` and based on it we will define all the others.

Make sure that all the new methods are placed in the protocol `'*Dice'`. To verify you can press the button Browse of the Monticello package browser and you should see the methods defined in the class `Integer`.

```
Integer >> D: anInteger
  ... Your solution ...
```

```
Integer >> D4
  ^ self D: 4
```

```
Integer >> D6
  ^ self D: 6
```

```
Integer >> D10
  ^ self D: 10
```

```
Integer >> D20
  ^ self D: 20
```

We obtain a compact form to create dice and we are ready for the last part: the addition of handles. We can write a new test named `testSumming`.

```
DiceHandleTest >> testSumming

  | handle |
  handle := 2 D20 + 3 D10.
  self assert: handle diceNumber equals: 5.
```

6.12 Conclusion

This chapter illustrates how to create a small DSL based on the definition of some domain classes (here `Dice` and`DieHandle`) and the extension of core classes such as `Integer`. It also shows that we can create packages with all the methods that are needed even when such methods are defined on classes external (here `Integer`) to the package. It shows that in Pharo we can use usual operators such as `+` to express natural models.

CHAPTER 7

Stone paper scissors

As we already saw sending a message is in fact making a choice. Indeed when we send a message, the method associated with the method in the class hierarchy of the receiver will be selected and executed.

Now we often have cases where we would like to select a method based on the receiver of the message and one argument. Again there is a simple solution named double dispatch that consists of sending another message to the argument hence making two choices one after the other.

This technique while simple can be challenging to grasp because programmers are so used to thinking that choices are made using explicit conditionals. In this chapter, we will show an example of double dispatch via the paper-stone-scissors game.

This exercise will show you an important paradigmatic shift where you will go from asking questions (conditionals) to sending orders. It is a clear illustration of the 'Don't ask, Tell' design principle.

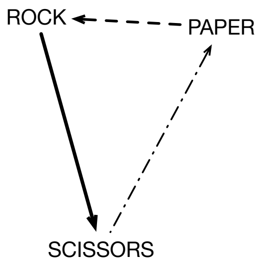


Figure 7-1 Stone paper scissors.

7.1 Starting with a couple of tests

We start by implementing a couple of tests. Let us define a test class named `StonePaperScissorsTest`.

```
TestCase << #StonePaperScissorsTest
  package: 'StonePaperScissors'
```

Now we can define a couple of tests showing for example that a paper is winning when a stone plays against a paper. We consider that the following tests are self-explanatory.

```
StonePaperScissorsTest >> testStoneAgainstPaperIsWinning
  self assert: (Stone new play: Paper new) equals: #paper
```

```
StonePaperScissorsTest >> testScissorAgsinstPaperIsWinning
  self assert: (Scissors new play: Paper new) equals: #scissors
```

```
StonePaperScissorsTest >> testStoneAgainsStone
  self assert: (Stone new play: Stone new) equals: #draw
```

Define them because we will use the tests in the future.

7.2 Creating the classes

First, let us create the classes that will correspond to the different players.

```
Object << #Paper
  package: 'StonePaperScissors'
```

```
Object << #Scissors
  package: 'StonePaperScissors'
```

```
Object << #Stone
  package: 'StonePaperScissors'
```

They could share a common superclass but we left it to you.

7.3 With messages

We are ready to make sure that the first test is passing. Let us work on `test-PaperIsWinning`.

```
StonePaperScissorsTest >> testStoneAgainstPaperIsWinning
  self assert: (Stone new play: Paper new) = #paper
```

The first method that we define is `play:` and it takes another player as an argument.

```
Stone >> play: anotherTool
  ... Your code ...
```

To implement this method we will use the fact that we know when its body is executed what is the receiver of the message. Here we are sure that the receiver is an instance of the class `Stone`.

So let us imagine that we have another method named `playAgainstStone:`

In the class `Paper`, it is clear that the method should return `#paper` because a paper wins against a stone. So just define it.

```
Paper >> playAgainstStone: aStone
  ... Your code ...
```

Now using the method `playAgainstStone:`, we can easily implement the previous method `play:` in the class `Stone`.

Do it and the test should pass now.

playAgainstStone:

Since we have started to implement `playAgainstStone:`, let us continue and implement two other methods one in the class `Scissors` and the other in the class `Stone`.

In the class `Scissors` the method should return that a stone wins.

```
Scissors >> playAgainstStone: aStone
  ... Your code ...
```

In the class `Stone`, the method should return a draw.

```
Stone >> playAgainstStone: aStone
  ... Your code ...
```

Let us verify that the following tests are passing. For this, we only execute the tests whose receiver of the `play:` message are stone instance.

First, we add a test to check the new scenario and now we have all the scenarios where a stone is the receiver.

```
StonePaperScissorsTest >> testStoneAgainstScissorsIsWinning
  self assert: (Stone new play: Scissors new) equals: #stone
```

```
StonePaperScissorsTest >> testStoneAgainsStone
  self assert: (Stone new play: Stone new) equals: #draw
```

The case where a stone is the receiver of the message play is handled and we can pass to another class, for example, `Scissors`.

Scissors now

Let us write first a test if this is already done. What we see is that a scissor is winning against a paper.

```
StonePaperScissorsTest >> testScissorIsWinning
  self assert: (Scissors new play: Paper new) equals: #scissors
```

Now we are ready to define the corresponding methods. First, we define the methods `playAgainstScissors:` in the corresponding classes.

```
Scissors >> playAgainstScissors: aScissors
  ... Your code ...
```

```
Paper >> playAgainstScissors: aScissors
  ... Your code ...
```

```
Stone >> playAgainstScissors: aScissors
  ... Your code ...
```

Now we are ready to we define the method `play:` in the class `Scissors`.

```
Scissors >> play: anotherTool
  ... Your code ...
```

You can define a couple of tests to make sure that your code is correct.

```
StonePaperScissorsTest >> testScissorAgainstStoneIsLosing
  self assert: (Scissors new play: Stone new) equals: #stone
```

```
StonePaperScissorsTest >> testScissorAgainstScissors
  self assert: (Scissors new play: Scissors new) equals: #draw
```

Paper now

We are now ready to do the same with the case of `Paper`. You should start to see the pattern. Define the method `playAgainstPaper:` in their corresponding classes.

```
Scissors >> playAgainstPaper: aPaper
  ... Your code ...
```

```
Paper >> playAgainstPaper: aPaper
  ... Your code ...
```

```
Stone >> playAgainstPaper: aPaper
  ... Your code ...
```

And now we can define the method `play:` in the `Paper` class.

```
Paper >> play: anotherTool
  ... Your code ...
```

Let us add more tests to cover the new cases.

```
StonePaperScissorsTest >> testPaperAgainstScissorIsLosing
  self assert: (Paper new play: Scissor new) equals: #scissors
```

```
StonePaperScissorsTest >> testPaperAgainstStoneIsWinning
  self assert: (Paper new play: Stone new) equals: #paper
```

```
StonePaperScissorsTest >> testPaperAgainstPaper
  self assert: (Paper new play: Paper new) equals: #draw
```

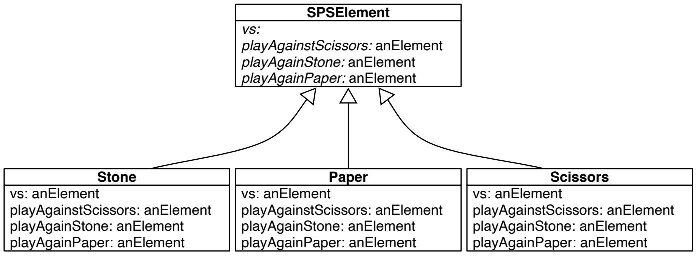

SPSElement
vs:
playAgainstScissors: anElement
playAgainStone: anElement
playAgainPaper: anElement
Stone
Paper
Scissors
vs: anElement
playAgainstScissors: anElement
playAgainStone: anElement
playAgainPaper: anElement

Figure 7-2 An overview of a possible solution using double dispatch.

The methods could return a value such as 1 when the receiver wins, 0 when there is a draw and -1 when the receiver loses. Add new tests and check this version.

7.4 About double dispatch

This exercise about double dispatch is really simple and it has two aspects that you may not find in other situations:

First, it is symmetrical. You play a stone against a paper or the inverse. Not all the double dispatches are symmetrical. For example, when drawing an object against a canvas the operation for example `drawOn: aCanva` is directed. It does not change much about the double dispatch but we wanted to make clear that it does not have to be this way.

Second, the secondary methods (`playAgainstXXX`) do not use the argument and this is because the example is super simple. In real-life examples, the secondary methods do use the argument for example to call back behavior on the argument. We will see this with the visitor design pattern.

7.5 A Better API

Both previous approaches either returning a symbol or a number are working but we can ask ourselves how the client will use this code.

Most of the time he will have to check again the returned result to perform some actions.

```
(aGameElement play: anotherGameElement) = 1
  ifTrue: [ do something for aGameElement ]
  (aGameElement play: anotherGameElement) = -1
```

So all in all, while this was a good exercise to help you understand that we do not need to have explicit conditionals and that we can use message passing instead, it felt a bit disappointing.

But there is a much better solution using double dispatch. The idea is to pass the action to be executed to the object and the object decides what to do.

```
Paper new competeWith: Paper new
  onDraw: [ Game incrementDraw ]
  onReceiverWin: [ ]
  onReceiverLose: [ ]
```

```
Paper new competeWith: Stone new
  onDraw: [ ]
  onReceiverWin: [ Game incrementPaper ]
  onReceiverLose: [ ]
```

Propose an implementation.

7.6 About alternative implementations

Here is a possible alternate implementation.

```
Paper >> play: anElement
  onDraw: aDrawBlock
  onWin: aWinBlock
  onLose: aLoseBlock

  ^ anElement
    playAgainstPaper: self
    onDraw: aDrawBlock
    onReceiverWin: aWinBlock
    onReceiverLose: aLoseBlock
```

```
Paper >> playAgainstPaper: anElement
  onDraw: aDrawBlock onReceiverWin:
  aWinBlock
  onReceiverLose: aLoseBlock
  ^ aDrawBlock value
```

What we see is that this new API is not that nice. Being forced to create blocks is not that great. A possibility would be to pass an object that knows what to do.

```
Paper new competeWith: Paper new
  result: aResultHolder
```

Here is a sketch of a possible implementation:

```
Paper >> competeWith: anElement result: aResultHolder
  ^ anElement playAgainstPaper: self result: aResultHolder
```

We still have the double dispatch but we only need one object taking take of the results.

```
Stone >> playAgainstPaper: anElement result: aResultHolder
  aResultHolder paperWins
```

7.7 Conclusion

Sending a message is making a choice among several methods. Depending on the receiver of a message the correct method will be selected. Therefore sending a message is making a choice and the different classes represent the possible alternatives.

Now this example illustrates this point but goes even further. Here we wanted to be able to make a choice depending on both an object and the argument of the message. The solution shows that it is enough to send back another message to the argument to perform a second selection that because of the first message now realizes a choice based on a message receiver and its argument.

CHAPTER 8

Revisiting the Die DSL: a case for double dispatch

In Chapter 6, using the Die DSL we could only sum die handles together as in `2 D20 + 1 D4`. In this new chapter, we extend the Die DSL implementation to support the sum of a die with another one or with a die handle (and vice versa).

One of the challenges is that the message `+` should be able to manage different types of receivers and arguments. The message will have either a die or a die handle as receiver and arguments, so we should manage the following possibilities: die `+` die handle, die `+` die, die handle `+` die handle, and die handle `+` die. While this extension at first may look trivial, we will take it as a way to explore double dispatch.

Double dispatch is a technic that avoids hardcoding type checks and also can define incrementally the behavior handling all the possible cases. Indeed double dispatch does not use any explicit conditionals and is the basis of more advanced Design Patterns such as the Visitor.

Double dispatch is based on the *Don't ask, tell* object-oriented principle applied twice. In the case of the `+` message, there is a first dispatch to select the adequate method. Then a second dispatch happens when in this method a new message is sent to the *argument* of the `+` message telling this argument the way the current receiver should be summed. This description is too abstract so we will go over a full example to explain it.

8.1 A little reminder

In a previous chapter, you implemented a small DSL to add dice and manage die handles. With this DSL, you could create dice and add them to a die handle. Later on, you could sum two different die handles and obtain a new one following the ”Dungeons and Dragons” ruling book.

The following tests show these two behaviors: First the dice handle creation and second the sum of die handles.

```
DieHandleTest >> testCreationAdding
  | handle |
  handle := DieHandle new
    addDie: (Die withFaces: 6);
    addDie: (Die withFaces: 10);
    yourself.
  self assert: handle diceNumber equals: 2
```

```
DieHandleTest >> testSummingWithNiceAPI
  | handle |
  handle := 2 D20 + 3 D10.
  self assert: handle diceNumber equals: 5
```

The implementation of + was simple since we could only sum die handles together. The method + creates a new handle, adds the dice of the receiver and of the argument to the newly created handle and returns it.

```
DieHandle >> + aDieHandle
  "Returns a new handle that represents the addition of the receiver
    and the argument."
  | handle |
  handle := self class new.
  self dice do: [ :each | handle addDie: each ].
  aDieHandle dice do: [ :each | handle addDie: each ].
  ^ handle
```

8.2 [Optional] Alternate way

We could also implement + using by asking the argument die handle to add its own dice as follows:

```
DieHandle >> + aDieHandle
  "Returns a new handle that represents the addition of the receiver
    and the argument."
  | handle |
  handle := self class new.
  self dice do: [ :each | handle addDie: each ].
  aDieHandle addDiceTo: handle.
  ^ handle
```

Implement the corresponding method `addDiceTo:` and verify that your tests still pass.

8.3 New requirements

The first requirement we have is that we want to be able to add two dice together and of course we should obtain a die handle as illustrated by the following test.

We want to add two dice together:

```
(Die withFaces: 6) + (Die withFaces: 6)
```

The second requirement is that we want to be able to mix and add a die to a die handle or vice versa as illustrated below:

```
2 D20 + (Die withFaces: 6)
```

```
 (Die withFaces: 6) + 2 D20
```

8.4 Turning requirements into tests

Since we are test-infested, we turn such expected behavior into automatically testable expected behavior: we write them as tests.

We want to add two dice together:

```
DieTest >> testAddTwoDice
  | hd |
  hd := (Die withFaces: 6) + (Die withFaces: 6).
  self assert: hd diceNumber equals: 2.
```

The second requirement is that we want to be able to mix and add a die to a die handle or vice versa as illustrated by the two following tests:

```
DieTest >> testAddingADieAndHandle
  | hd |
  hd := (Die withFaces: 6)
    +
    (DieHandle new
      addDie: (Die withFaces: 10);
      yourself).
  self assert: hd diceNumber equals: 2
```

```
DieHandleTest >> testAddingAnHandleWithADie
  | handle res |
  handle := DieHandle new
    addDie: (Die withFaces: 6);
    addDie: (Die withFaces: 10);
    yourself.
  res := handle + (Die withFaces: 20).
  self assert: res diceNumber equals: 3
```

The two previous tests are not really robust so we will introduce a little behavior to make sure that we can have much better tests.

8.5 Introducing faces on DieHandle

The previous test `testAddingADieAndHandle` is not really good because it can pass just if we add two objects in the die handle and this is not satisfactory. We will introduce `numberOfFaces`. This method should satisfy the following test:

```
DieTest >> testNumberOfFaces
  | hd |
  hd := (DieHandle new
      addDie: (Die withFaces: 10);
      addDie: (Die withFaces: 6);
      yourself).
  self assert: hd faces equals: 16
```

Define the method `faces` on `DieHandle`. It is following nearly the same logic as the method `roll`.

```
DieHandle >> faces
  "return the number of faces of the receiver"
  ...
```

Now we are ready to implement such requirements.

8.6 The first implementation

The first solution is to explicitly type-check the argument to decide what to do.

```
DieHandle >> + aDieOrADieHandle

  ^ (aDieOrADieHandle class = DieHandle)
    ifTrue: [ | handle |
          handle := self class new.
          self dice do: [ :each | handle addDie: each ].
          aDieOrADieHandle dice do: [ :each | handle addDie: each ].
          handle ]
    ifFalse: [ | handle |
          handle := self class new.
          self dice do: [ :each | handle addDie: each ].
          handle addDie: aDieOrADieHandle.
          handle  ]
```

```
Die >> + aDieOrADieHandle
  | selfAsDieHandle |
  selfAsDieHandle := DieHandle new addDie: self.
  ^ selfAsDieHandle + aDieOrADieHandle
```

The problem with this solution is that it does not scale. As soon as we will have other kinds of arguments we will have to check more and more cases. You may think that this is just a spurious argument. But when you have a model that has around 35 different kinds of nodes as in Pillar, the document processing system used to produce this book, this kind of testing logic becomes a nightmare to maintain and extend.

8.7 Sketching double dispatch

We can do better. The logic of the solution we have in mind is quite simple but it may be destabilizing at first. Let us sketch it.

- When we execute a method we know its receiver and the kind of receiver we have: it can be a die or a die handle. The method dispatch will select the correct method at runtime. Imagine that we have two `+` methods for each class `Die` and `DieHandle`. When a given method `+` will be executed, we will know the exact kind of the receiver. For example, when the method `+` defined on the class `Die` is executed, we will know that the receiver is a die (instance of this class). Similarly, when the method `+` defined on the class `DieHandle` is executed, we will know that the message receiver is a die handle. This is the power of method dispatch: it selects the right method based on the message receiver.
- Then the idea is to tell the argument that we want to sum it with that given receiver. It means that each `+` method on a different class has just to send a different message based on the fact that the receiver was a die or a die handle to its argument and let the method dispatch to act once again. After this second dispatch, the correct method will be selected.

But let us make this concrete.

8.8 Adding two dice

Let us step back and start by supporting the sum of two dice. This is rather simple we create and return a die handle to which we add the receiver and the argument.

```
Die >> + aDie

  ^ DieHandle new
    addDie: self;
    addDie: aDie; yourself
```

Our first test should pass `testAddTwoDice`. But this solution does not support the fact that the argument can be either a die or a die handle.

8.9 Adding a die and a die or a handle

Now we want to handle the fact that we can add a die or a die handle to the receiver as illustrated by the test `testAddingADieAndHandle`.

```
DieTest >> testAddingADieAndHandle
  | hd |
  hd := (Die withFaces: 6)
    +
    (DieHandle new
      addDie: 6;
      yourself).
  self assert: hd diceNumber equals: 2
```

The previous method `+` is definitively what we want to do when we have two dice. So let us rename it as `sumWithDie:` so that we can invoke it later.

```
Die >> sumWithDie: aDie
  ... Your code ...
```

Now what we can do is to implement `+` as follows. Notice that we named the argument `aDicable` because we want to convey that the argument can be either a die or a die handle.

```
Die >> + aDicable
  ... Your code ...
```

We tell the argument `aDicable` (which can be a die or a die handle) that we want to add a die to it (we know that `self` in this method is a `Die` because this is the method of this class that is executed). When rewriting the `+` method, we switched `self` and `aDicable` to send the new message `sumWithDie:` to the argument (`aDicable`). This switch kicks a new method dispatch and we finally have a double dispatch (one of `+` and one for `sumWithDie:`).

In our two tests `testAddTwoDice` and `testAddingADieAndHandle` we know that the receiver is a die because the method is defined in the class of `Die`. At this point, the test `testAddTwoDice` should pass because we are adding two dice as shown in Figure 8-1.

8.10 When the argument is a die handle

Now we still have to find a solution for the case where the argument to the message `+` is a die handle. In fact, the argument will receive the message `sumWithDie:`. Therefore if we define a method with that name in the class `DieHandle` it will be executed when the argument of message `+` is a die handle.

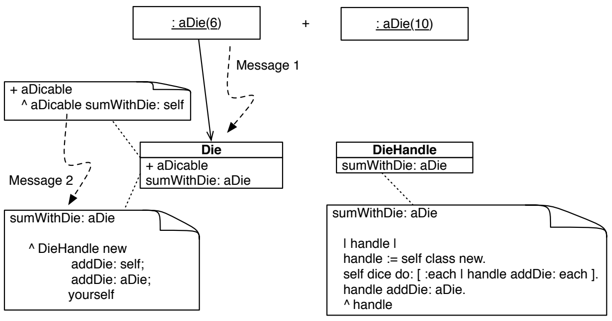


Figure 8-1 Summing two dice and be prepared for more.

We know how to sum a die with a die handle: we simply create a new die handle, add all the die of the previous die handle to the new one and add the argument too.

So we just have to define the method `sumWithDie:` to the class `DieHandle` implementing this logic.

```
DieHandle >> sumWithDie: aDie
  ... Your code ...
```

Now we can sum a die with a die handle as shown in Figure 8-2. The test `testAddingADieAndHandle` should now pass.

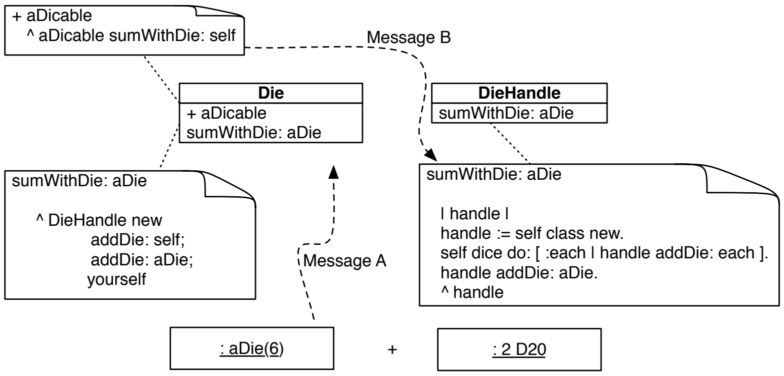


Figure 8-2 Summing a die and a dicable.

8.11 Stepping back

You may ask why this is working. We defined two methods `sumWithDie:` one on the class `Die` and one in the class `DieHandle` and when the method `+` on class `Die` will send the message `sumWithDie:` to either a die or a die handle, the message dispatch will select the correct method `sumWithDie:` for us as shown in Figure 8-3.

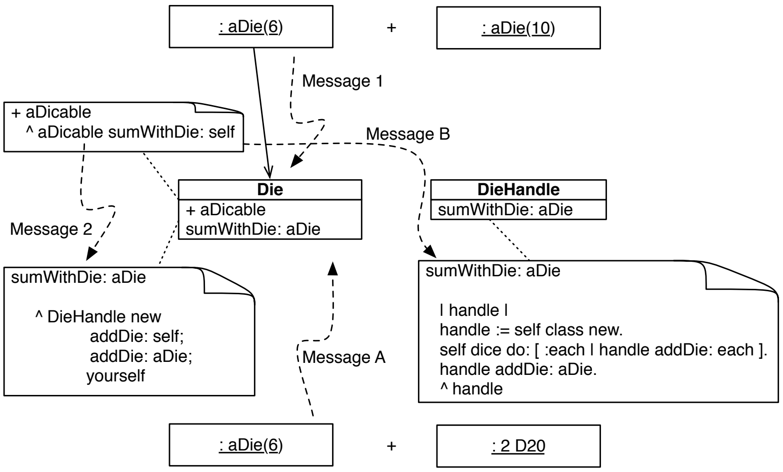


Figure 8-3 Summing a die and a dicable

8.12 Now a DieHandle as receiver

Our solution does not handle the case where the receiver is a die handle. This is what we will address now. Now we are ready to apply the same pattern as before but for the case where the receiver is a die handle. We will just say to the argument of the message `+` that we want to sum it with a *die handle* this time.

We know how to sum two die handles, it is the code we already defined in the previous chapter. We rename the `+` method as `sumWithHandle:` to be able to invoke it while redefining the method `+`. Basically, this method creates a new handle, then adds the dice of the receiver and the argument to it, and returns the new handle.

```
DieHandle >> sumWithHandle: aDieHandle
  ... Your code ...
```

Now we can define a more powerful version of `+` by simply sending the message `sumWithHandle:` to the **argument** (aDicable) of the message `+`. Again we send a message to the argument (`aDicable`) to kick in a new message lookup and dispatch for the message `sumWithHandle:`.

```
DieHandle >> + aDicable
  ... Your code ...
```

We said that this version of `+` is more powerful than the one of `sumWithHandle:` because once we will implement the missing method `sumWithHandle:` on the class `Die`, the `+` method will be able to sum a die handle with a die or two die handles.

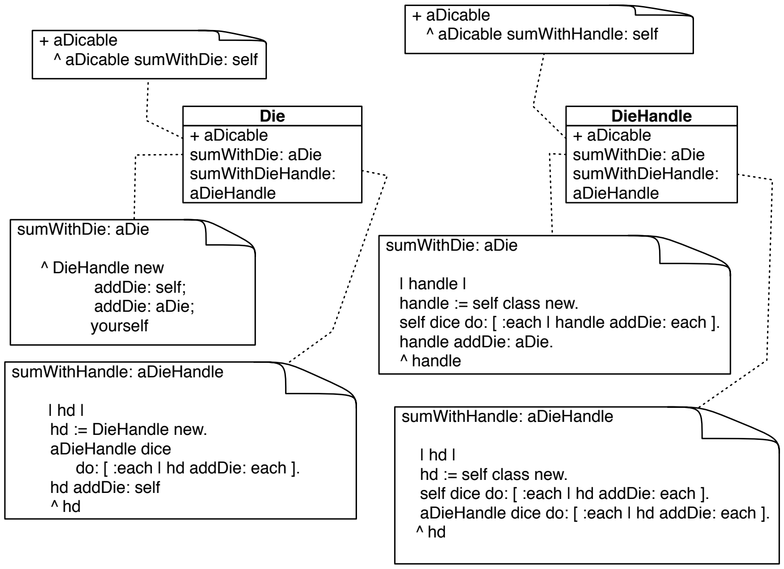

+ aDicable
^ aDicable sumWithDie: self
+ aDicable
^ aDicable sumWithHandle: self
Die
+ aDicable
sumWithDie: aDie
sumWithDieHandle: aDieHandle
DieHandle
+ aDicable
sumWithDie: aDie
sumWithDieHandle: aDieHandle
sumWithDie: aDie
^ DieHandle new addDie: self; addDie: aDie; yourself
sumWithDie: aDie
| handle |
handle := self class new.
self dice do: [:each | handle addDie: each].
handle addDie: aDie.
^ handle
sumWithHandle: aDieHandle
| hd |
hd := DieHandle new.
aDieHandle dice do: [:each | hd addDie: each].
hd addDie: self
^ hd
sumWithHandle: aDieHandle
| hd |
hd := self class new.
self dice do: [:each | hd addDie: each].
aDieHandle dice do: [:each | hd addDie: each].
^ hd

Figure 8-4 Handling all the cases: summing a die/die handle with a die/die handle .

Up until here, we did not change much and all the tests adding two die handles should continue to run.

8.13 sumWithHandle: on Die class

To get the possibility to sum a die handle with a single die, we just have to define a new method `sumWithHandle:` on the `Die` class. The logic is similar to the one adding one die to one die handle

```
Die >> sumWithHandle: aDieHandle
  ... Your code ...
```

Note that we could have sent the message `aDieHandle sumWithDie: self` as the body of `sumWithHandle:` definition.

Figure 8-4 shows the full setup. We suggest following the execution of messages for the different cases to understand that just sending a new message to the argument and relying on method dispatch produces modular conditional execution. Now the following test should pass and we are done.

```
DieHandleTest >> testAddingAnHandleWithADie
  | handle res |
  handle := DieHandle new
    addDie: (Die faces: 6);
    addDie: (Die faces: 10);
    yourself.
  res := handle + (Die withFaces: 20).
  self assert: res diceNumber equals: 3
```

8.14 Conclusion

When we step back, we see that we applied the *Don't ask, tell* principle twice: First the message `+` selects the corresponding methods in either `Die` or `DieHandle` classes. Then a more specific message is sent to the argument and the dispatch kicks in again selecting the correct method for the messages `sumWithDie:` or `sumWithHandle:`.

In this chapter, we presented double dispatch. The idea is to use the method of dispatch two times. While the resulting design is simple, it is not trivial to deeply understand and it requires time to digest double dispatch. At its core, double dispatch relies on the fact that sending a message to an object selects the correct method – and sending another message to the message argument will select a new method. Therefore we have effectively selected a method according to the receiver and the argument of a message.

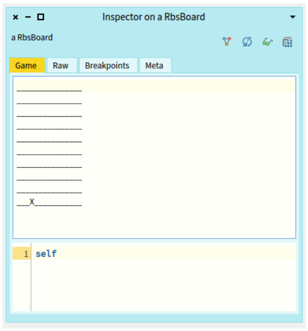
Double dispatch is the basis for the Visitor Design pattern that is effective when dealing with complex data structures such as documents, and compilers. In such context, it is not rare to have more than 30 or 40 different nodes that should be manipulated together to produce specific behavior.

CHAPTER 9

A little Ssaturn PathFinder

We launched a pathfinder robot on Saturn and the communication with the robot is difficult. To interact with this robot we send it *orders* in *scripts* from Earth and the robot executes them. Energy and communication are limited so we use a compact representation of scripts.

Your mission is to define different orders and functionalities such as replay, way back home, and path optimizations.



```
_______________
_______________
_______________
_______________
_______________
_______________
_______________
_______________
_______________
___X___________
```

```
self
```

Figure 9-1 A 2D space and a robot in ascii.

Doing this project you will learn the Command design pattern and delegation to objects that encapsulate their behavior.

9.1 A robot in its space

A robot lives in a 2D space. It starts in a location. The following code snippet is producing Figure 9-1.

```
| rb b |
rb := RbsRobot new.
b := RbsBoard new.
rb setBoard: b.
rb x: 4 y: 1.
rb inspect
```

A board is composed of cells. Pay attention that a cell in the board only contains one element: a ground tile or a robot. So when moving a robot to a cell will 'erase' the background. So when moving a robot should put back the previous tile.

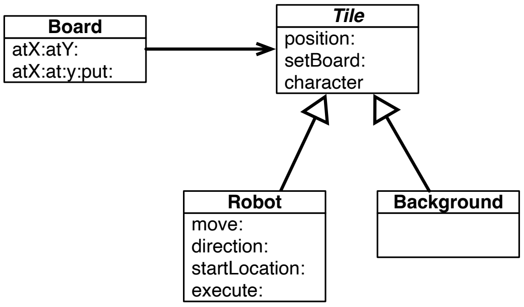


Figure 9-2 A minimal design.

9.2 Scripts

A robot receives a script as strings containing *orders*. The following test illustrates this.

- First a robot is created.
- Second a board is created. The robot is placed in the space.
- Third the robot can execute a script.

```
testExecute

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  rb execute:
```

```
'dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4
```

The script contains different orders: such as `mov 3`, `dir #north`.

9.3 Getting the code

To help you develop this project we provide some core behavior. The robot code is available at: `https://github.com/pharo-mooc/AdvancedDesignMoocProjectCo`

- To start, load the baseline name `RobotsProject`, it contains the board logic, and board tests in addition to basic behavior for tiles composing the space. Note that this
- Once you define the class `RbsRobot` (for example in a package named `Robots`) as a subclass of `RbsAbstractRobot`, load the package `Robots-Tests`. It contains the tests for the behavior you will have to define.

9.4 Basic robot behavior

Define methods `direction:` and `direction` to define the direction of the robot and initialize it for example to point to the east.

```
testRobotDefaultDirection

  | rb |
  rb := RbsRobot new.
  self assert: rb direction equals: #east
```

9.5 Robot move

The first step is to implement orders such as `mov`, `dir`. Each order can be implemented by defining a method such as `move: aDistance` and `direction:`. Propose an implementation for these methods. Here is a possible test for the `move:`.

```
testRobotMove

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  "should make sure that previous tile is put back"
  self assert: (rb board atX: 4 atY: 1) equals: rb.
  rb move: 10.
  self assert: (rb board atX: 14 atY: 1) equals: rb.
  self assert: (rb board atX: 14 atY: 1) equals: rb
```

Pay attention that `move:` should put back the ground after moving.

To help you we propose to use the following method `computeNewPosition:`, but there is a bug (it does not return a point). Write a couple of tests and fix the method.

```
computeNewPosition: anInteger
  "Returns a point representing the location of the next move."
  ^ direction = #east
    ifTrue: [ self x + anInteger ]
    ifFalse: [ direction = #west
      ifTrue: [ self x - anInteger  ]
      ifFalse: [ direction = #north
          ifTrue: [ self y + anInteger ]
          ifFalse: [ self y - anInteger ].
        ]
      ]
```

The method `move:` now handles the fact that we put back the background tile when moving the robot. But we were tired and there was a bug in that method, fix it!

```
move: anInteger

  | newPosition |
  newPosition := self computeNewPosition: anInteger.
  self previousTile position: newPosition.
  previousTile := self board atPosition: newPosition.
  self position: newPosition.
```

The following test should pass:

```
testRobotMovePreservesGround

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  self assert: rb previousTile class equals: RbsGround.
  self assert: rb previousTile x equals: 4.
  rb move: 10.
  self assert: (rb board atX: 4 atY: 1) class equals: RbsGround.
  self assert: (rb board atX: 14 atY: 1) equals: rb.
  self assert: rb previousTile position equals: 14@1
```

9.6 Sending order to robots

Now we are ready to implement the method `execute:` that will execute the orders. The following helper method splits the script into line based orders.

```
RbsRobot >> identifyOrdersOf: aString

  | orders |
  orders := aString splitOn: Character cr.
  orders := orders collect: [ :each | each splitOn: Character space
    ].
  ^ orders
```

In addition you can use the following expression `Object readFrom: aString` to get the Pharo object represented by the string.

```
Object readFrom: '1'
> 1

Object readFrom: 'true'
> true
```

You should make the following test passes:

```
testExecute

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  rb execute:
'dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4
```

9.7 Adding new orders

We propose now to introduce new orders.

Base

It was strange to not have the base position as part of the script so we propose to introduce a new order `base` taking two numbers as x and y.

```
base 10 20
```

Here is a test that should pass.

```
testStartPositionAsOrder

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb execute:
'base 4 1
dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4
```

Dropping an item

Introduce the possibility for the robot to drop an item on the map. Introduce the class `RbsItem` with, for example, the character `o` as textual representation and handle the order in the `execute:` method.

```
dropL
```

9.8 Introducing commands

Imagine that you originally defined the `execute:` method as follows:

```
execute: aString

  (self identifyOrdersOf: aString)
    do: [ :each |
    each first = #mov
      ifTrue: [ self move: (Object readFrom: each second) ]
      ifFalse: [
        each first = #dir ifTrue: [
          self direction: (Object readFrom: each second) ] ] ]
```

You certainly saw that adding a new order is tedious and make the conditional statements more and more complex. This can get even more complex if we want to implement a replay of the orders. We propose to use Commands. Commands are objects representing actions.

Load the package named `Robots-BasicCommands-Tests`. It contains some tests to help you creating commands.

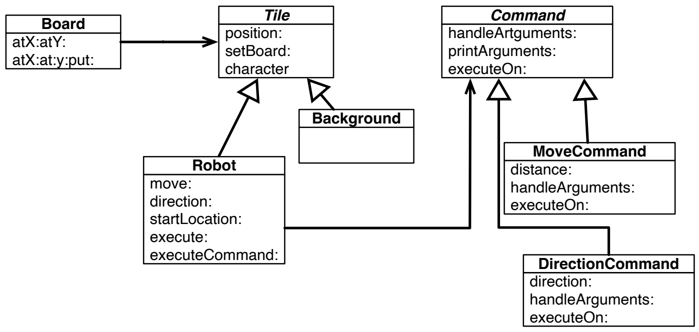


Figure 9-3 A design with Command.

Command

Each command can have its own state and in addition its should know how to execute itself and convert the order arguments into a Pharo object. Here is an example for the `RbsMoveCommand`. What you see is that it has its own state, an `executeOn:` method and a way to handle the arguments of the script.

```
RbsCommand << #RbsMoveCommand
  slots: { #distance };
  package: 'Robots'
```

```
RbsMoveCommand >> executeOn: aRobot
  aRobot move: distance
```

```
RbsMoveCommand >> handleArguments: aCol
  distance := Object readFrom: aCol first
```

Registering commands

We need a way to associate orders to commands. We do it by defining the method `commandName` on the class side of the command class.

```
RbsMoveCommand class >> commandName
  ^ 'mov'
```

The robot class should have a way to map 'mov' to the class of the command Something like:

```
initializeCommandMapping

  cmdMap := Dictionary new.
  RbsCommand allSubclassesDo: [ :each |
    cmdMap at: each commandName put: each
    ]
```

that the `executeCommandBased: aString` should use when creating and executing commands.

```
executeCommandBased: aString

  (self identifyOrdersOf: aString) do: [ :each |
    ((self commandClassFor: each first) new
      handleArguments: each allButFirst; yourself) executeOn: self ]
```

Implement all the commands so that the following test should pass.

```
testExecuteCommandBased

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  rb executeCommandBased:
'dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4
```

9.9 Challenge: Replay

We would like to monitor what the robot is doing to be able to replay it. Load the package `Robots-Replay-Tests`. Here is a typical script and we can replay it with another starting position.

```
testReplay

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 4 1'.
  rb executeCommandBased:
'dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4.
  rb x: 5 y: 1.
  rb replay.
  self assert: rb position equals: 10@4
```

Let us imagine that the method execute commandBased: was implemented as

```
RbsRobot >> executeCommandBased: aString

  (self identifyOrdersOf: aString) do: [ :each |
    ((self commandClassFor: each first) new
      handleArguments: each allButFirst; yourself) executeOn: self ]
```

You should introduce a way to keep the created commands so that they can be replayed. For example consider adding an instance variable path initialized as an OrderedCollection and add commands when you create them in the previous method.

Introduce new commands to control replay

Note that in the test above we used rb x:5 y: 1. instead of rb executeCommandBased: 'base 5 1'. This is due to the fact that we cannot control when the recording is starting and that we cannot reset it or stop it either. We propose you to introduce the following commands: startM, stopM, restM, and replay.

Add a new instance variable monitoring to the robot class and two methods to control it as well as an initialization.

```
startMonitoring
  monitoring := true
```

```
stopMonitoring
  monitoring := false
```

The following test shows that we are registering stopM as a command. We will fix that below.

```
testMonitoringIsOnPerDefault

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 5 1
dir #east
stopM
mov 3'.
  self assert: rb path size equals: 3
```

The following test verifiess that once the monitoring is stopped and the path reset, the path is empty

```
testReset

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 5 1
dir #east
stopM
resM
mov 3'.
  self assert: rb path size equals: 0
```

9.10 Non recording commands

The following test may loop so pay attention because replay will replay the sequence that will replay itself endlessly.

```
testReplayAsCommand

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 4 1'.
  rb executeCommandBased:
'resM
dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4.
rb executeCommandBased: 'base 5 1
replay'.
  self assert: rb position equals: 10@4
```

We could rely on the script programmer to always stop the monitoring before placing a replay order. But to have better security and avoid endless loop because `replay` would be replaying itself, it is important that `replay` is not added to the path of commands. The following test loops because the replay order is causing itself to be kicked.

Propose one solution where `replay` is not added to the path. Such a solution can be defined without any conditional by giving each command the responsibility to add itself to the path.

Instead of doing a conditional before adding the command the path, we can just ask the command to add itself to the path of the robot. This way the replay command can ignore it. So introducing a hook in place of calling directly the path addition (`path addLast: cmd.`) is a nice solution because each command can define its own behavior.

```
executeCommandBased: aString
  ...
  path addLast: cmd.
  ...
```

becomes

```
executeCommandBased: aString
  ...
  cmd addToPathOf: self
  ...
```

This forces us to introduce a method named for example `addToPath:` in the robot class to expose path addition. Once the corresponding logic is added and used the following test will pass.

```
testAddToPathCommandsDoesNotContainReplay

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 5 1
dir #east
mov 3
replay'.
  self assert: rb path size equals: 3
```

Once the command `stop`, `start`, `reset` and `replay` are not recorded anymore the tests should be changed. For example `testMonitoringIsOnPerDefault` checks that the path is now only containing two commands.

```
testMonitoringIsOnPerDefault

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 5 1
dir #east
stopM
mov 3'.
  self assert: rb path size equals: 2
```

Now we are ready to use `replay` as an order. The following test verifies it.

```
testReplayAsCommand

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 4 1'.
  rb executeCommandBased:
'resM
dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4.
rb executeCommandBased: 'stopM
base 5 1
replay'.
  self assert: rb position equals: 10@4
```

9.11 Challenge: Automatic way back home

It can be tedious to bring back the robot to its location be inverting one by one the orders that compose a script. We propose to enhance our robots with a `wayBack` order. Load the package `Robots-WayBack-Tests`. A way back action with take a list of commands and produce a new list of commands with the opposite actions. Figure 9-4 illustrates the behavior:

When we have a simple path

```
dir #east
mov 5
dir #north
mov 3
dir #east
mov 4
wback
```

the robot should perform the following path back. We stressed that the directions should be inversed.

```
dir #east => west
mov 4
dir #north
mov 3
dir #east => west
mov 5
```

What we see is that we should not only

- remove the way back order
- reverse the list
- but also convert direction in the opposite ones.

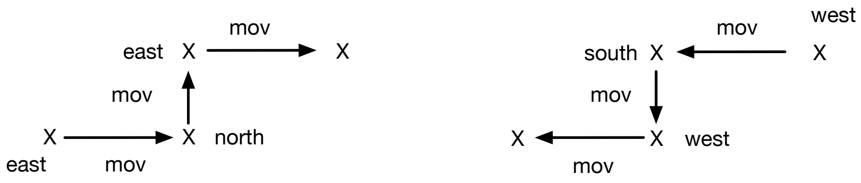


Figure 9-4 A simple path and a way back home.

Notice that multiple mov orders can be before a change direction as in the equivalent path:

```
dir #east => west
mov 2
mov 2
dir #north
mov 1
mov 1
mov 1
dir #east => west
mov 5
```

A a first step we propose to introduce a simple message on the direction command class and the root of command.

```
RbsCommand >> asWayBack

  ^ self
```

Imagine the implementation for the direction commands.

```
testDirectionWyaBAck
  | opposite |
  opposite := (RbsDirectionCommand new direction: #east) asWayBack.
  self assert: opposite direction equals: #west.
  opposite := (RbsDirectionCommand new direction: #west) asWayBack.
  self assert: opposite direction equals: #east.
```

To help you in this challenge we propose you to use the following method `ifCutOn: isSplitterBlock doWithCutAndUncuts: aTwoArgBlock finally: aBlock`. If it is not available in Pharo, just define it on `SequenceableCollection`. The following tests should illustrate clearly what the method does.

```
testCut

  | res |
  res := OrderedCollection new.
  #(2 2 #east 1 1 1 #north 5 #east 666)
    ifCutOn: [ :s | s isSymbol ]
    doWithCutAndUncuts: [ :cut :before | res addLast: cut; addAll:
     before ]
    finally: [:u | res addLast: u].
  self assert: res equals: #(#east 2 2 #north 1 1 1 #east 5 666)
     asOrderedCollection
```

```
SequenceableCollection >> ifCutOn: isSplitterBlock
    doWithCutAndUncuts: aTwoArgBlock finally: aBlock
  "Applies aTwoArgBlock (with current splitter objects and previous
    unsplit objects) to the receiver. When uncuts are left executes
    aBlock with them.
  An optimised version could work with indexes to avoid creating
    intermediate collections."

  | uncuts cut current |
  uncuts := OrderedCollection new.
  1 to: self size do: [ :i |
    current := self at: i.
    cut := isSplitterBlock value: current.
    cut
      ifFalse: [ uncuts addLast: current ]
      ifTrue: [
          aTwoArgBlock value: current value: uncuts.
          uncuts := OrderedCollection new ]].
  uncuts isEmpty
    ifFalse: [ aBlock value: uncuts ]
```

Extensions

- We could introduce a turn back message that given a command return its opposite based on its previous state. Given a path sequence east mov 5 north mov 3 east mov 7 it would generate the sequence west mov 7 north mov 3 south mov 5...

9.12 Challenge: Path optimizations

This extension is about supporting path optimizations. Load the package 'Robots-Optimize-Tests. `Let us imagine that the treatment of a command is costly on Saturn. Then it can be better to optimize the received script before executing it. Optimization can be` quite simple, indeed n mov‘ commands can be merged as a single move command with the sum of the commands.

The following orders

```
move 10
move 20
move 5
```

can be replaced by a single one:

```
move 35
```

Several following direction commands can be merged as the last command.

The following sequence

```
dir #east
dir #south
dir #north
```

is optimized as

```
dir #north
```

We suggest the following design. Introduce a message `aCommand mergeWith: anotherCommand` that returns a list containing the situation after trying to merge:

- When two commands can merge, returns a list containing the command resulting from the merge.
- When two commands do not merge, returns a list containing the two original commands.

You can use double dispatch to determine how commands of different classes are merged. As a default you can decide that different commands do not merge.

```
RbsRobotTest >> testMergeMoveCommandsProducesTheSum

  | cmdList |
  cmdList := (RbsMoveCommand new distance: 10; yourself)
    mergeWith: (RbsMoveCommand new distance: 10; yourself).
  self assert: cmdList size equals: 1.
  self assert: cmdList first distance equals: 20.
```

```
RbsRobotTest >> testMergeUNmergeableCommandsBecauseDifferent

  | cmdList |
  cmdList := (RbsMoveCommand new distance: 10; yourself)
    mergeWith: (RbsDirectionCommand new direction: #east; yourself).
  self assert: cmdList size equals: 2.
  self assert: cmdList first distance equals: 10.
```

Once the merge semantics is in place you can use this logic to optimize full paths as illustrated by the following test. Pay attention because this is a bit tricky in particular since

```
mov 1
mov 2
mov 3
dir #east
```

leads to

```
mov 3
mov 3
dir #east
```

and then finally to

```
mov 6
dir #east
```

The following test should pass.

```
testOptimizeMergeThreeMovesAndOthers

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  rb optimizePath:
'mov 2
mov 3
mov 4
dir #east'.
  self
    assert: (rb path collect: [ :each | each printString ])
    equals: #( 'mov 9' 'dir #east')
```

Extensions

You can also add the fact that a mov 5 followed by a mov -5 does not produce any command. Returning an empty list should be managed.

9.13 Extensions

Here is a list of extensions:

- The robot should be able to pick an item.
- It can have a certain capacity and cannot carry too many items.

- Passing a symbol to the direction is bad because the script developer may mistype it and exposing the internal logic is a bad idea. Propose a solution.
- The definition of the new location of a robot is based on a boring conditional. Can you imagine a better way?

```
computeNewPosition: anInteger
  "Returns a point representing the location of the next move."
  ^ direction = #east
    ifTrue: [ self x + anInteger ]
    ifFalse: [ direction = #west
      ifTrue: [ self x - anInteger  ]
      ifFalse: [ direction = #north
          ifTrue: [ self y + anInteger ]
          ifFalse: [ self y - anInteger ].
        ]
      ]
```

To give you a hint, we could have a little hierarchy with direction and each direction would decide the new location when told to compute it.

```
East computeFor: 4@1 distance: 10
> 14@1
```

9.14 Conclusion

This micro project shows you that representing actions as objects lets us manipulate programs at the right level of abstractions. Functionality as undo, replay, or path optimizations are easier to develop using commands. In addition refraining from using conditions is interesting because it forces us to delegate responsibilities to the objects and this makes your design more modular.

CHAPTER 10

Finding the North with Compass

In this chapter, we will work on an alternative way to represent directions and move computation in the 2D plan.

10.1 Existing situation

Computing new position based on a direction.

In the Robot implementation proposed in Chapter 9, we computed the new position of a robot as follows:

```
computeNewPosition: anInteger
  "Returns a point representing the location of the next move."

  ^ direction = #east
    ifTrue: [ self x + anInteger @ self y ]
    ifFalse: [ direction = #west
      ifTrue: [ self x - anInteger @self y ]
      ifFalse: [ direction = #north
          ifTrue: [ self x @ (self y + anInteger)]
          ifFalse: [ self x @ (self y - anInteger) ].
        ]
      ]
```

This is not that nice.

Opposite direction

Similarly, we computed the opposite direction as follows:

```
computeOppositeDirection: aDirection
  "Returns the opposite direction.
  Note that this implementation should be rewritten taking into
    account Compass' way of representing direction and their
    computation'"

  ^ aDirection = #east
    ifTrue: [ #west ]
    ifFalse: [ aDirection = #west
      ifTrue: [ #east ]
      ifFalse: [ aDirection = #north
          ifTrue: [ #south]
          ifFalse: [ #north].
        ]
      ]
```

10.2 Representing directions

We propose that you define a little hierarchy with the class `CpDirection` as a root and as subclasses the four main directions and based on it compute the opposite and a new position in an adjacent position.

Note that by design we avoided directly referring to subclasses but use the root as a factory of instances of its subclasses.

Make sure that the following tests pass and define new ones for each scenario.

```
testSouthReturnOneRowDownPosition

  | newPos |
  newPos := CpDirection south * ( 3 @ 2).
  self assert:  newPos x equals: 3.
  self assert:  newPos y equals: 3.
```

```
testWestReturnLeftPosition

  | newPos |
  newPos := CpDirection west * ( 3 @ 2).
  self assert:  newPos x equals: 2.
  self assert:  newPos y equals: 2.
```

New position at a given distance

While the message `*` was given the next adjacent position, define tests and introduce the message `in: aDistance from: aPosition`.

```
testEastInDistanceReturnRightPosition

  | newPos |
  newPos := CpDirection east in: 3 from: (3 @ 2).
  self assert:  newPos x equals: 6.
  self assert:  newPos y equals: 2.
```

10.3 Introducing NorthWest, SouthEast, and friends

Now that you have got your four positions and all your tests green. Introduce the missing directions: NorthWest, NorthEast, SouthEast, and SouthWest. And enjoy this design.

CHAPTER 11

A little expression interpreter

In this chapter, you will build a small mathematical expression interpreter. For example, you will be able to build an expression such as (3 + 4) * 5 and then ask the interpreter to compute its value. You will revisit tests, classes, messages, methods, and inheritance. You will also see an example of expression trees similar to the ones that are used to manipulate programs. For example, compilers and code refactorings as offered in Pharo and many modern IDEs are doing such manipulation with trees representing code. In addition, we will extend this example to present the Visitor Design Pattern.

11.1 Starting with constant expression and a test

We start with a constant expression. A constant expression is an expression whose value is always the same, obviously.

Let us start by defining a test case class as follows:

```
TestCase << #EConstantTest
  package: 'Expressions'
```

We decided to define one test case class per expression class and this even if at the beginning the classes will not contain many tests. It is easier to define new tests and navigate them.

Let us write a first test making sure that when we get a value, sending it the `evaluate` message returns its value.

```
EConstantTest >> testEvaluate
  self assert: (EConstant new value: 5) evaluate equals: 5
```

When you compile such a test method, the system should prompt you to get a class `EConstant` defined. Let the system drive you. Since we need to store the value of a constant expression, let us add an instance variable `value` to the class definition.

At the end you should have the following definition for the class `EConstant`.

```
Object << #EConstant
  slots: {'value'};
  package: 'Expressions'
```

We define the method `value:` to set the value of the instance variable `value`. It is simply a method taking one argument and storing it in the `value` instance variable.

```
EConstant >> value: anInteger
  value := anInteger
```

You should define the method `evaluate`: it should return the value of the constant.

```
EConstant >> evaluate
  ... Your code ...
```

Your test should pass.

11.2 Negation

Now we can start to work on expression negation. Let us write a test and for this define a new test case class named `ENegationTest`.

```
TestCase << #ENegationTest
  package: 'Expressions'
```

The test `testEvaluate` shows that a negation applies to an expression (here a constant) and when we evalute we get the negated value of the constant.

```
ENegationTest >> testEvaluate
  self assert: (ENegation new expression: (EConstant new value: 5))
     evaluate equals: -5
```

Let us execute the test and let the system help us to define the class. A negation defines an instance variable to hold the expression that it negates.

```
Object << #ENegation
  slots: { #expression };
  package: 'Expressions'
```

We define a setter method to be able to set the expression under negation.

```
ENegation >> expression: anExpression
  expression := anExpression
```

Now the `evaluate` method should request the evaluation of the expression and negate it. To negate a number the Pharo library proposes the message `negated`.

```
ENegation >> evaluate
  ... Your code ...
```

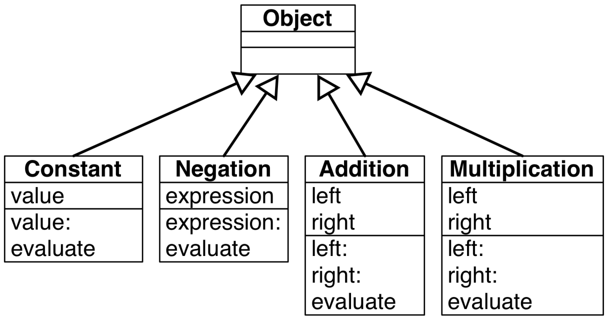


Figure 11-1 A flat collection of classes (with a suspect duplication).

Following the same principle, we will add expression addition and multiplication. Then we will make the system a bit more easy to manipulate and revisit its first design.

11.3 Adding expression addition

To be able to do more than constant and negation we will add two extra expressions: addition and multiplication and after we will discuss about our approach and see how we can improve it.

To add an expression that supports addition, we start to define a test case class and a simple test.

```
TestCase << #EAdditionTest
  package: 'Expressions'
```

A simple test for addition is to make sure that we add correctly two constants.

```
EAdditionTest >> testEvaluate
  | ep1 ep2 |
  ep1 := (EConstant new value: 5).
  ep2 := (EConstant new value: 3).
  self assert: (EAddition new right: ep1; left: ep2) evaluate
    equals: 8
```

You should define the class `EAddition`: it has two instance variables for the two subexpressions it adds.

```
EExpression << #EAddition
  slots: { #left . #right};
  package: 'Expressions'
```

Define the two corresponding setter methods `right:` and `left:`.

Now you can define the `evaluate` method for addition.

```
EAddition >> evaluate
  ... Your code ...
```

To make sure that our implementation is correct we can also test that we can add negated expressions. It is always good to add tests that cover *different* scenario.

```
EAdditionTest >> testEvaluateWithNegation
  | ep1 ep2 |
  ep1 := ENegation new expression: (EConstant new value: 5).
  ep2 := (EConstant new value: 3).
  self assert: (EAddition new right: ep1; left: ep2) evaluate
    equals: -2
```

11.4 Multiplication

We do the same for multiplication: create a test case class named `EMultiplicationTest`, a test, a new class `EMultiplication`, a couple of setter methods and finally a new `evaluate` method. Let us do it fast and without much comments.

```
TestCase << #EMultiplicationTest
  package: 'Expressions'
```

```
EMultiplicationTest >> testEvaluate
  | ep1 ep2 |
  ep1 := (EConstant new value: 5).
  ep2 := (EConstant new value: 3).
  self assert: (EMultiplication new right: ep1; left: ep2) evaluate
    equals: 15
```

```
Object subclass: #EMultiplication
  slots: { #left . #right};
  package: 'Expressions'
```

```
EMultiplication >> right: anExpression
  right := anExpression
```

```
EMultiplication >> left: anExpression
  left := anExpression
```

```
EMultiplication >> evaluate
  ... Your code ...
```

11.5 Stepping back

It is interesting to look at what we built so far. We have a group of classes whose instances can be combined to create complex expressions. Each expression is in fact a tree of subexpressions as shown in Figure 11-2. The figure shows two main trees: one for the constant expression `5` and one for the expression `-5 + 3`. Note that the diagram represents the number `5` as an object because in Pharo even small integers are objects in the same way the instances of `EConstant` are objects.

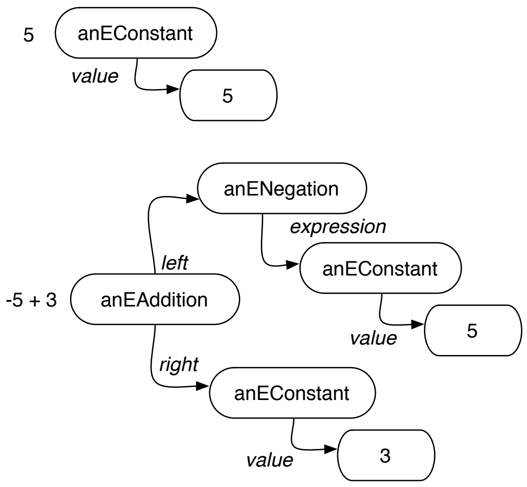


Figure 11-2 Expressions are composed of trees.

Messages and methods

The implementation of the `evaluate` message is worth discussing. What we see is that *different* classes understand the same message but execute different methods as shown in Figure 11-3. A message represents an intent: it represents *what* should be done. A method represents a specification of *how* something should be executed. What we see is that sending a message `evaluate` to an expression is making a choice among the different implementations of the message. This point is central to object-oriented programming. Sending a message is making a choice among all the methods with the same name.

About common superclass

So far we did not see the need to have an inheritance hierarchy because there is not much to share or reuse. Now adding a common superclass would be useful to convey to the reader of the code or a future extender of the library that such concepts are related and are different variations of expression.

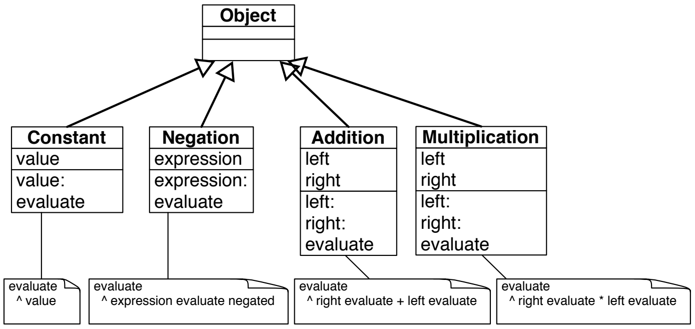


Figure 11-3 Evaluation: one message and multiple method implementations.

Design corner: About addition and multiplication model

We could have just one class called for example BinaryOperation and it can have an operator and this operator will be either the addition or multiplication. This solution can work and as usual having a working program does not mean that its design is any good.

In particular having a single class would force us to start to write conditional based on the operator as follows

```
BinaryExpression >> evaluate
  operator = #+
    ifTrue: [ left evaluate + right evaluate ]
    ifFalse: [ left evaluate * right evaluate]
```

There are ways in Pharo to make such code more compact but we do not want to use it at this stage. For the interested reader, look for the message `perform:` that can execute a method based on its name.

This is annoying because the execution engine itself is made to select methods for us so we want to avoid to bypass it using explicit condition. In addition when we will add power, division, subtraction we will have to have more cases in our condition making the code less readable and more fragile.

As we will see as a general message in this book, sending a message is making a choice between different implementations. Now to be able to choose we should have different implementations and this implies having different classes. Classes represent choices whose methods can be selected during message passing. Having more little classes is better than few large ones. What we could do is to introduce a common superclass between `EAddition` and EMultiplication but keep the two subclasses. We will probably do it in the future

11.6 Negated as a message

Negating an expression is expressed in a verbose way. We have to create explicitly each time an instance of the class ENegation as shown in the following snippet.

```
ENegation new expression: (EConstant new value: 5)
```

We propose to define a message negated on the expressions themselves that will create such instance of ENegation. With this new message, the previous expression can be reduced too.

```
(EConstant new value: 5) negated
```

negated message for constants

Let us write a test to make sure that we capture well what we want to get.

```
EConstantTest >> testNegated
  self assert: (EConstant new value: 6) negated evaluate equals: -6
```

And now we can simply implement it as follows:

```
EConstant >> negated
  ^ ENegation new expression: self
```

negated message for negations

```
ENegationTest >> testNegationNegated
  self assert: (EConstant new value: 6) negated negated evaluate
     equals: 6
```

```
ENegation >> negated
  ^ ENegation new expression: self
```

This definition is not the best we can do since in general it is a bad practice to hardcode the class usage inside the class. A better definition would be

```
ENegation >> negated
  ^ self class new expression: self
```

But for now we keep the first one for the sake of simplicity

negated message for additions

We proceed similarly for additions.

```
EEAdditionTest >> testNegated
  | ep1 ep2 |
  ep1 := EConstant new value: 5.
  ep2 := EConstant new value: 3.
  self assert: (EAddition new right: ep1; left: ep2) negated
    evaluate equals: -8
```

```
EAddition >> negated
  ... Your code ...
```

negated message for multiplications

We proceed similarly for multiplications.

```
EMultiplicationTest >> testEvaluateNegated
  | ep1 ep2 |
  ep1 := EConstant new value: 5.
  ep2 := EConstant new value: 3.
  self assert: (EMultiplication new right: ep1; left: ep2) negated
    evaluate equals: -15
```

```
EMultiplication >> negated
  ... Your code ...
```

Now all your tests should pass. And it is a good moment to save your package.

11.7 Annoying repetition

Let us step back and look at what we have. We have a working situation but again object-oriented design is to bring the code to a better level.

Similarly to the situation of the `evaluate` message and methods we see that the functionality of `negated` is distributed over different classes. Now what is annoying is that we repeat the exact *same* code over and over and this is not good (see Figure 11-4). This is not good because if tomorrow we want to change the behavior of negation we will have to change it four times while in fact one time should be enough.

What are the solutions?

- We could define another class `Negator` that would do the job and each current classes would delegate to it. But it does not really solve our problem since we will have to duplicate all the message sends to call `Negator` instances.
- If we define the method `negated` in the superclass (`Object`) we only need one definition and it will work. Indeed, when we send the message `negated` to an instance of `EConstant` or `EAddition` the system

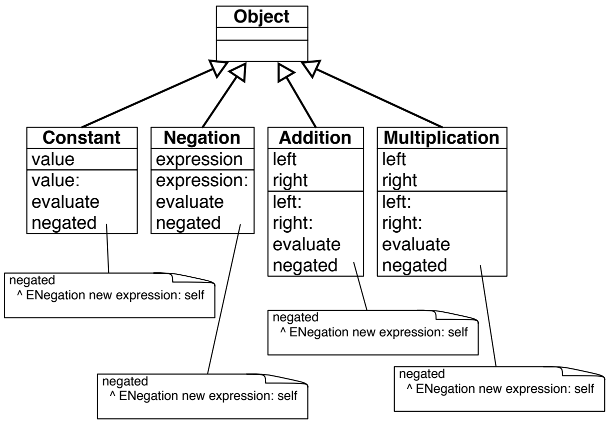


Figure 11-4 Code repetition is a bad smell.

will not find it locally but in the superclass `Object`. So no need to define it four times but only one in class `Object`. This solution is nice because it reduces the number of similar definitions of the method `negated` but it is not good because even if in Pharo we can add methods to the class `Object` this is not a good practice. `Object` is a class shared by the entire system so we should take care not to add behavior only making sense for a single application.

- The solution is to introduce a new superclass between our classes and the class `Object`. It will have the same property than the solution with `Object` but without poluting it (see Figure 11-5). This is what we do in the next section.

11.8 Introducing Expression class

Let us introduce a new class to obtain the situation depicted by Figure 11-5. We can simply do it by adding a new class:

```
Object << #EExpression
  package: 'Expressions'
```

and changing all the previous definitions to inherit from `EExpression` instead of `Object`. For example the class `EConstant` is then defined as follows.

```
EExpression << #EConstant
  slots: { #value};
  package: 'Expressions'
```

We can also use for the first transformation the class refactoring *Insert superclass*. Refactorings are code transformations that do not change the behavior

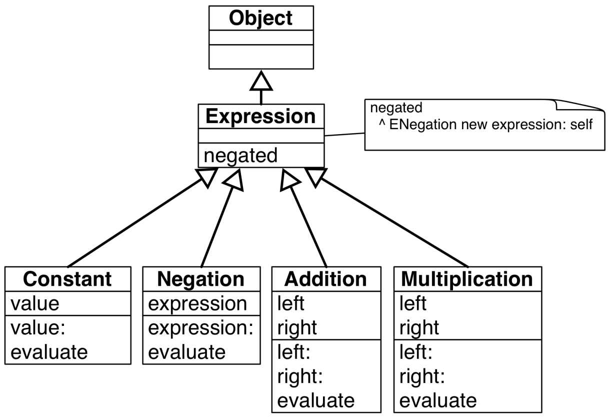


Figure 11-5 Introducing a common superclass.

of a program. You can find it under the refactorings list when you bring the menu on the classes. Now it is only useful for the first changes.

Once the classes `EConstant`, `ENegation`, `EAddition`, and `EMultiplication` are subclasses of `EEXpression`, we should focus on the method `negated`. Now the method refactoring *Push up* will help us.

- Select the method `negated` in one of the classes
- Select the refactoring *Push up*

The system will define the method `negated` in the superclass (`EExpression`) and remove all the negated methods in the classes. Now we obtain the situation described in Figure 11-5. It is a good moment to run all your tests again. They should all pass.

Now you could think that we can introduce a new class named Arithmetic-Expression as a superclass of `EAddition` and `EMultiplication`. Indeed this is something that we could do to factor out common structure and behavior between the two classes. We will do it later because this is basically just a repetition of what we have done.

11.9 Class creation messages

Until now we always sent the message new to a class followed by a setter method as shown below.

```
EConstant new value: 5
```

We would like to take the opportunity to show that we can define simple **class** methods to improve the class instance creation interface. In this example it is simple and the benefits are not that important but we think that this is a nice example. With this in mind the previous example can now be written as follows:

```
EConstant value: 5
```

Notice the important difference that in the first case the message is sent to the newly created instance while in the second case it is sent to the class itself.

To define a class method is the same as to define an instance method (as we did until now). The only difference is that using the code browser you should click on the classSide button to indicate that you are defining a method that should be executed in response to a message sent to a class itself.

Better instance creation for constants

Define the following method on the class `EConstant`. Notice the definition now use `EConstant class` and not just `EConstant` to stress that we are defining the class method.

```
EConstant class >> value: anInteger
  ^ self new value: anInteger
```

Now define a new test to make sure that our method works correctly.

```
EConstantTest >> testCreationWithClassCreationMessage
  self assert: (EConstant value: 5) evaluate equals: 5
```

Better instance creation for negations

We do the same for the class `ENegation`.

```
ENegation class >> expression: anExpression
  ... Your code ...
```

We write of course a new test as follows:

```
ENegationTest >> testEvaluateWithClassCreationMessage
  self assert: (ENegation expression: (EConstant value: 5)) evaluate
    equals: -5
```

Better instance creation for additions

For the addition we add a class method named `left:right:` taking two arguments

```
EAddition class >> left: anInteger right: anInteger2
  ^ self new left: anInteger ; right: anInteger2
```

Of course, since we are test infested we add a new test.

```
EEAdditionTest >> testEvaluateWithClassCreationMessage
  | ep1 ep2 |
  ep1 := EConstant value: 5.
  ep2 := EConstant value: 3.
  self assert: (EAddition left: ep1 right: ep2) evaluate equals: 8
```

Better instance creation for multiplications

We let you do the same for the multiplication.

```
EMultiplication class >> left: anExp right: anExp2
  ... Your code ...
```

And another test to check that everything is ok.

```
EMultiplicationTest >> testEvaluateWithClassCreationMessage
  | ep1 ep2 |
  ep1 := EConstant new value: 5.
  ep2 := EConstant new value: 3.
  self assert: (EMultiplication new left: ep1; right: ep2) evaluate
    equals: 15
```

Run your tests! They should all pass.

11.10 Introducing examples as class messages

As you saw when writing the tests, it is quite annoying to repeat all the time the expressions to get a given tree. This is especially the case in the tests related to addition and multiplication as the one below:

```
EEAdditionTest >> testNegated
  | ep1 ep2 |
  ep1 := EConstant new value: 5.
  ep2 := EConstant new value: 3.
  self assert: (EAddition new right: ep1; left: ep2) negated
    evaluate equals: -8
```

One simple solution is to define some class method returning typical instances of their classes. To define a class method remember that you should click the class side button.

```
EConstant class >> constant5
  ^ self new value: 5
```

```
EConstant class >> constant3
  ^ self new value: 3
```

This way we can define the test as follows:

```
EEAdditionTest >> testNegated
  | ep1 ep2 |
  ep1 := EConstant constant5.
  ep2 := EConstant constant3.
  self assert: (EAddition new right: ep1; left: ep2) negated
    evaluate equals: -8
```

The tools in Pharo support such a practice. If we tag a class method with the special annotation `<sampleInstance>` the browser will show a little icon on the side and when we click on it, it will open an inspector on the new instance.

```
EConstant class >> constant3
  <sampleInstance>
  ^ self new value: 3
```

using the same idea we defined the following class methods to return some examples of our classes.

```
EAddition class >> fivePlusThree
  <sampleInstance>
  | ep1 ep2 |
  ep1 := EConstant new value: 5.
  ep2 := EConstant new value: 3.
  ^ self new left: ep1 ; right: ep2
```

```
EMultiplication class >> fiveTimesThree
  <sampleInstance>
  | ep1 ep2 |
  ep1 := EConstant constant5.
  ep2 := EConstant constant3.
   ^ EMultiplication new left: ep1 ; right: ep2
```

What is nice about such examples is that

- they help to document the class by providing objects that we can directly use,
- they support the creation of tests by providing objects that can serve as input for tests,
- they simplify the writing of tests.

So think about using them.

11.11 Printing

It is quite annoying that we cannot really see an expression when we inspect it. We would like to get something better than `'aEConstant'` and `'anEAddition'` when we debug our programs. To display such information the debugger and inspector send to the objects the message `printString` which by default just prefix the name of the class with 'an' or 'a'.

Let us change this situation. For this, we will specialize the method `printOn: aStream`. The message `printOn:` is called on the object when a program or the system send to the object the message `printString`. From that perspective `printOn:` is a system customization point that developers can take advantage to enhance their programming experience.

Note that we do not redefine the method `printString` because it is more complex and `printString` is reused for all the objects in the system. We just have to implement the part that is specific to a given class. In object-oriented design jargon, `printString` is a template method in the sense that it sets up a context that is shared by other objects and it hosts hook methods which are program customization points. `printOn:` is a hook method. The term hook comes from the fact that code of subclasses is invoked in the hook place (see Figure 11-6).

The default definition of the method `printOn:` as defined on the class `Object` is the following: it grabs the class name, checks if it starts with a vowel or not and writes to the stream the 'a/an class'. This is why by default we got `'anEConstant'` when we printed a constant expression.

```
Object >> printOn: aStream
  "Append to the argument, aStream, a sequence of characters that
  identifies the receiver."
  | title |
  title := self class name.
  aStream
    nextPutAll: (title first isVowel ifTrue: ['an '] ifFalse: ['a
    ']);
    nextPutAll: title
```

A word about streams

A stream is a container for a sequence of objects. Once we get a stream we can either read from it or write to it. In our case we will write to the stream. Since the stream passed to printOn: is a stream expecting characters we will add characters or strings (sequence of characters) to it. We will use the messages: `nextPut: aCharacter` and `nextPutAll: aString`. They add to the stream the arguments at the next position and following. We will guide you and it is simple. You can find more information on the chapter about Stream in the book: Pharo by Example available at http://books.pharo.org

Printing constant

Let us start with a test. Here we check that a constant is printed as its value.

```
EConstantTest >> testPrinting
  self assert: EConstant constant5 printString equals: '5'
```

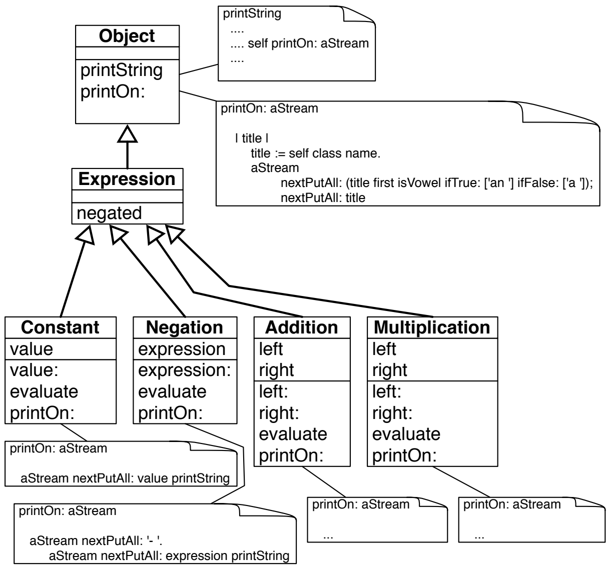


Figure 11-6 printOn: and printString a "hooks and template" in action.

The implementation is then simple. We just need to put the value converted as a string to the stream.

```
EConstant >> printOn: aStream
  aStream nextPutAll: value printString
```

Printing negation

For a negation we should first put a '-' and then recurvisely call the printing process on the negated expression. Remember that sending the message `printString` to an expression should return its string representation. At least until now it will work for constants.

```
(EConstant value: 6) printString
>>> '6'
```

Here is a possible definition

```
ENegation >> printOn: aStream
  aStream nextPutAll: '- '
  aStream nextPutAll: expression printString
```

By the way since all the messages are sent to the same object, this method can be rewritten as:

```
ENegation >> printOn: aStream
  aStream
    nextPutAll: '- ';
    nextPutAll: expression printString
```

We can also define it as follows:

```
ENegation >> printOn: aStream
  aStream nextPutAll: '- '.
  expression printOn: aStream
```

The difference between the first solution and the alternate implementation is the following: In the solution using `printString`, the system creates two streams: one for each invocation of the message `printString`. One for printing the expression and one for printing the negation. Once the first stream is used the message `printString` converts the stream contents into a string and this new string is put inside the second stream which at the end is converted again as a string. So the first solution is not really efficient. With the second solution, only one stream is created and each of the methods just put the needed string elements inside. At the end of the process, the single `printString` message converts it into a string.

Printing addition

Now let us write yet another test for addition printing.

```
EAdditionTest >> testPrinting
  self assert: (EAddition fivePlusThree) printString equals:  '( 5 +
    3 )'.
  self assert: (EAddition fivePlusThree) negated printString equals:
     '- ( 5 + 3 )'
```

Printing an addition is: put an open parenthesis, print the left expression, put ' + ', print the right expression and put a closing parenthese in the stream.

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

Printing multiplication

And now we do the same for multiplication.

```
EMultiplicationTest >> testPrinting
  self assert: (EMultiplication fiveTimesThree) negated printString
    equals:  '- ( 5 * 3 )'
```

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

11.12 Revisiting negated message for Negation

Now we can go back to negating an expression. Our implementation is not nice even if we can negate any expression and get the correct value. If you look at it carefully negating a negation could be better. Printing a negated negation illustrates well the problem: we get two minus operations instead of none.

```
(EConstant value: 11) negated
>> '- 11'

(EConstant value: 11) negated negated
>> '- - 11'
```

A solution could be to change the `printOn:` definition and to check if the expression that is negated is a negation and in such case to not emit the minus. Let us say it now, this solution is not nice because we do not want to write code that depends on explicitly checking if an object is of a given class. Remember we want to send a message and let the object do some actions.

A good solution is to *specialize* the message `negated` so that when it is sent to a *negation* it does not create a new negation that points to the receiver but instead returns the expression itself, otherwise the method implemented in `EExpression` will be executed. This way the trees created by a `negated` message can never have negated negation but the arithmetic values obtained are correct. Let us implement this solution, we just need to implement a different version of the method `negated` for `ENegation`.

Let us write a test! Since evaluating a single expression or a double negated one gives the same results, we need to define a structural test. This is what we do with the expression `exp negated class = ENegation` below.

```
NegationTest >> testNegatedStructureIsCorrect
  | exp |
  exp := EConstant value: 11.
  self assert: exp negated class = ENegation.
  self assert: exp negated negated equals: exp.
```

Now you should be able to implement the `negated` message on `ENegation`.

```
ENegation >> negated
  ... Your code ...
```

Understanding method override

When we send a message to an object, the system looks for the corresponding method in the class of the receiver then if it is not defined there, the lookup continues in the superclass of the previous class.

By adding a method in the class `ENegation`, we created the situation shown in Figure 11-7. We said that the message `negated` is overridden in `ENega-`

tion because for instances of ENegation it hides the method defined in the superclass EExpression.

It works the following:

- When we send the message negated to a constant, the message is not found in the class EConstant and then it is looked up in the class EExpression and it is found there and applied to the receiver (the instance of EConstant).
- When we send the message negated to a negation, the message is found in the class ENegation and executed on the negation expression.

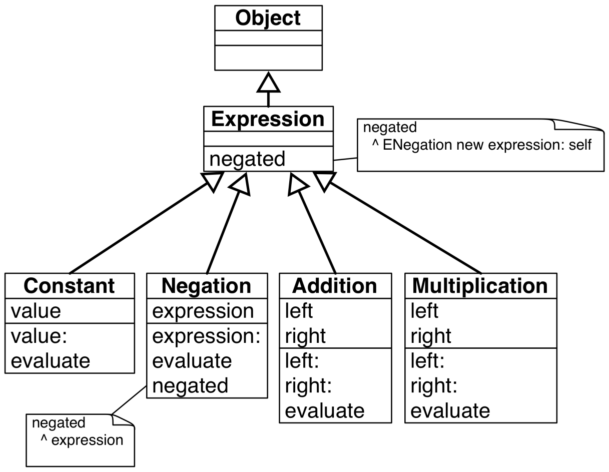


Figure 11-7 The message negated is overridden in the class ENegation.

11.13 Introducing BinaryExpression class

Now we will take a moment to improve our first design. We will factor out the behavior of EAddition and EMultiplication.

```
EExpression << #EBinaryExpression
  package: 'Expressions'
```

```
EBinaryExpression << #EAddition
  slots: { #left . #right'};
  package: 'Expressions'
```

```
EBinaryExpression << #EMultiplication
  slots: { #left . #right};
  package: 'Expressions'
```

Now we can use again a refactoring to pull up the instance variables `left` and `right`, as well as the methods `left:` and `right:`.

Select the class `EMuplication`, bring the menu, and select in the Refactoring menu the instance variables refactoring *Push Up*. Then select the instance variables.

Now you should get the following class definitions, where the instance variables are defined in the new class and removed from the two subclasses.

```
EExpression << #EBinaryExpression
  slots: { #left . #right };
  package: 'Expressions'
```

```
EBinaryExpression << #EAddition
  package: 'Expressions'
```

```
EBinaryExpression << #EMultiplication
  package: 'Expressions'
```

We should get a situation similar to the one in Figure 11-8. All your tests should still pass.

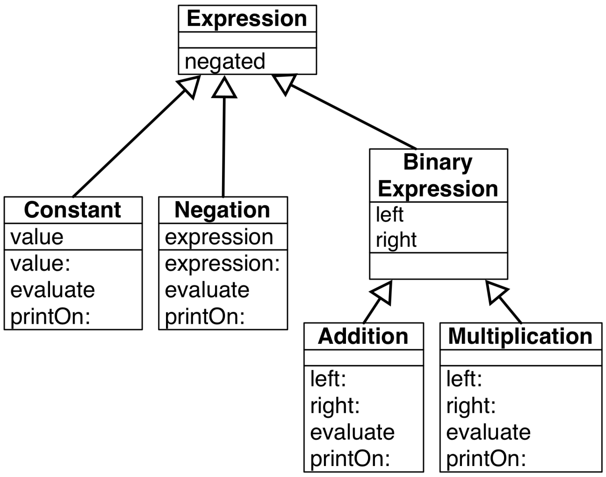


Figure 11-8 Factoring instance variables.

Now we can move the same way the methods. Select the method `left:` and apply the refactoring *Pull Up Method*. Do the same for the method `right:`.

Creating a template and hook method

Now we can look at the methods `printOn:` of additions and multiplications. They are really similar: Just the operator is changing. Now we cannot simply copy one of the definitions because it will not work for the other. But what we can do is apply the same design point that was implemented for `printString` and `printOn:`: we can create a template and hooks that will be specialized in the subclasses.

We will use the method `printOn:` as a template with a hook redefined in each subclass.

Let define the method `printOn:` in `EBinaryExpression` and remove the other ones from the two classes `EAddition` and `EMultiplication`.

```
EBinaryExpression >> printOn: aStream
  aStream nextPutAll: '( '.
  left printOn: aStream.
  aStream nextPutAll: ' + '.
  right printOn: aStream.
  aStream nextPutAll: ' )'
```

Then you can do it manually or use the *Extract Method* Refactoring: This refactoring creates a new method from a part of an existing method and sends a message to the new created method: select the ' + ' inside the method pane and bring the menu and select the Extract Method refactoring, and when prompt gives the name `operatorString`.

Here is the result you should get:

```
EBinaryExpression >> printOn: aStream
  aStream nextPutAll: '( '.
  left printOn: aStream.
  aStream nextPutAll: self operatorString.
  right printOn: aStream.
  aStream nextPutAll: ' )'
```

```
EBinaryExpression >> operatorString
  ^ ' + '
```

Now we can just redefine this method in the `EMultiplication` class to return the adequate string.

```
EMultiplication >> operatorString
  ^ ' * '
```

11.14 What did we learn

The introduction of the class `EBinaryExpression` is a rich experience in terms of lessons that we can learn.

- Refactorings are more than simple code transformations. Usually, refactorings pay attention that their application does not change the behavior of programs. As we saw refactorings are powerful operations that really help doing complex operations in a few actions.

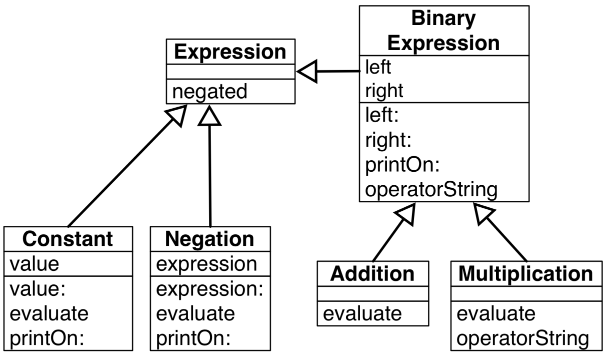


Figure 11-9 Factoring instance variables and behavior.

- We saw that the introduction of a new superclass and moving instance variables or methods to a superclass does not change the structure or behavior of the subclasses. This is because (1) for the state, the structure of an instance is based on the state of its class and all its superclasses, (2) the lookup starts in the class of the receiver and looks in superclasses.
- While the method `printOn:` is by itself a hook for the method `printString`, it can also play the role of a template method. The method `operatorString` reuses the context created by the `printOn:` method which acts as a template method. In fact, each time we do a self-send we create a hook method that subclasses can specialize.

11.15 About hook methods

When we introduced `EBinaryExpression` we defined the method `operatorString` as follows:

```
EBinaryExpression >> operatorString
  ^ ' + '
```

```
EMultiplication >> operatorString
  ^ ' * '
```

And you may wonder if it was worth to create a new method in the superclass and so that such one subclass redefines it.

Creating hooks is always good

First creating a hook is also a good idea. Because you rarely know how your system will be extended in the future. On this little example, we suggest you to add raising to power, division and this can be done with one class and two methods per new operator.

Avoid not documenting hooks

Second, we could have just defined one method `operatorString` in each subclass and no method in the superclass `EBinaryExpression`. It would have worked because `EBinaryExpression` is not meant to have direct instances. Therefore there is no risk that a `printOn:` message is sent to one of its instances and causes an error because no method `operatorString` is found.

The code would have looked like the following:

```
EAddition >> operatorString
  ^ ' + '
```

```
EMultiplication >> operatorString
  ^ ' * '
```

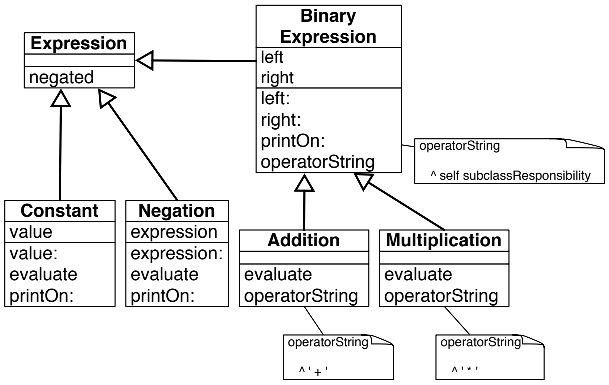


Figure 11-10 Better design: Declaring an abstract method as a way to document a hook method.

Now such design is not really good because as a potential extender of the code, developers will have to guess reading the subclass definitions that they should also define a method `operatorString`. A much better solution in that case is to define what we can an abstract method in the superclass as follows:

```
EBinaryExpression >> operatorString
  ^ self subclassResponsibility
```

Using the message `subclassResponsibility` declares that a method is abstract and that subclasses should redefine it explicitly. Using such an approach we get the final situation represented in Figure 11-10.

In the solution presented before (section 11.13) we decided to go for the simplest solution and it was to use one of the default value (' + ') as a default definition for the hook in the superclass `EExpression`. It was not a good solution and we did it on purpose to be able to have this discussion. It was not a good solution since it was using a specific subclass. It is better to define a default value for a hook in the superclass when this default value makes sense in the class itself.

Note that we could also define `evaluate` as an abstract method in `EExpression` to indicate clearly that each subclass should define an `evaluate`.

11.16 Variables

Up until now our mathematical expressions are rather limited. We only manipulate constant-based expressions. What we would like is to be able to manipulate variables too. Here is a simple test to show what we mean: we define a variable named `'x'` and then we can later specify that `'x'` should take a given value.

Let us create a new test class named `EVariableTest` and define a first test `testValueOfx`.

```
EVariableTest >> testValueOfx
  self assert: ((EVariable new id: #x) evaluateWith: {#x -> 10}
    asDictionary) equals: 10.
```

Some technical points

Let us explain a bit what we are doing with the expression `{#x -> 10} asDictionary`. We should be able to specify that a given variable name is associated with a given value. For this we create a dictionary: a dictionary is a data structure for storing keys and their associated value. Here a key is the variable and the value its associated value. Let us present some details first.

Dictionaries

A dictionary is a data structure containing pairs (key value) and we can access the value of a given key. It can use any object as key and any object as values. Here we simply use a symbol `#x` since symbols are unique within the system and as such we are sure that we cannot have two keys looking the same but having different values.

```
| d |
d := Dictionary new
  at: #x put: 33;
  at: #y put: 52;
  at: #z put: 98.
d at: y
```

```
>>> 52
```

The previous dictionary can be easily expressed more compactly using `{#x -> 33 . #y -> 52 . #z -> 98} asDictionary`.

```
{#x -> 33 . #y -> 52 . #z -> 98} asDictionary at: #y
>>> 52
```

Dynamic Arrays

The expression `{ }` creates a dynamic array. Dynamic arrays executes their expressions and store the resulting values.

```
{2 + 3 . 6 - 2 . 7-2 }
>>> ==#(5 4 5)==
```

Pairs

The expression `#x -> 10` creates a pair with a key and a value.

```
| p |
p := #x -> 10.
p key
>>> #x
p value
>>> 10
```

Back to variable expressions

If we go a step further, we want to be able to build more complex expressions where instead of having constants we can manipulate variables. This way we will be able to build more advanced behavior such as expression derivations.

```
EExpression << #EVariable
  slots: { #id};
  package: 'Expressions'
```

```
EVariable >> id: aSymbol
  id := aSymbol
```

```
EVariable >> printOn: aStream
  aStream nextPutAll: id asString
```

What we see is that we need to be able to pass bindings (a binding is a pair key, value) when evaluating a variable. The value of a variable is the value of the binding whose key is the name of the variable.

```
EVariable >> evaluateWith: aBindingDictionary
  ^ aBindingDictionary at: id
```

Your tests should all pass at this point.

For more complex expressions (the ones that interest us) here are two tests.

```
EVariableTest >> testValueOfxInNegation
  self assert: ((EVariable new id: #x) negated
    evaluateWith: {#x -> 10} asDictionary) equals: -10
```

What the second test shows is that we can have an expression and given a different set of bindings the value of the expression will differ.

```
EVariableTest >> testEvaluateXplusY
  | ep1 ep2 add |
  ep1 := EVariable new id: #x.
  ep2 := EVariable new id: #y.
  add := EAddition left: ep1 right: ep2.

  self assert: (add evaluateWith: { #x -> 10 . #y -> 2 }
    asDictionary) equals: 12.
  self assert: (add evaluateWith: { #x -> 10 . #y -> 12 }
    asDictionary) equals: 22
```

Non working approaches

A non working solution would be to add the following method to `EExpression`

```
EEXpression >> evaluateWith: aDictionary
  ^ self evaluate
```

However, it does not work for at least the following reasons:

- It does not use its argument. It only works for trees composed exclusively of constant.
- When we send a message `evaluateWith:` to an addition, this message is then turned into an `evaluate` message sent to its subexpression and such subexpression do not get an `evaluateWith:` message but an `evaluate`.

Alternatively we could add the binding to the variable itself and only provide an `evaluate` message as follows:

```
(EVariable new id: #x) bindings: { #x -> 10 . #y -> 2 } asDictionary
```

But it fully defeats the purpose of what a variable is. We should be able to give different values to a variable embedded inside a complex expression.

The solution: adding evaluateWith:

We should transform all the implementations and message sends from `evaluate` to `evaluateWith:` Since this is a tedious task we will use the method refactoring *Add Parameter*. Since a refactoring applies itself to the complete system, we should be a bit cautious because other Pharo classes implement methods named `evaluate` and we do not want to impact them.

So here are the steps that we should follow.

- Select the Expression package
- Choose Browse Scoped (it brings a browser with only your package)
- Using this browser, select a method evaluate
- Select the *Add Parameter* refactoring: type `evaluateWith:` as the method selector and proceed when prompted for a default value `Dictionary new`. This last expression is needed because the engine will rewrite all the messages `evaluate` but `evaluateWith: Dictionary new`.
- The system is performing many changes. Check that they only touch your classes and accept them all.

A test like the following one:

```
EConstant >> testEvaluate
  self assert: (EConstant constant5) evaluate equals: 5
```

is transformed as follows:

```
EConstant >> testEvaluate
  self assert: ((EConstant constant5) evaluateWith: Dictionary new)
    equals: 5
```

Your tests should nearly all pass except the ones on variables. Why do they fail? Because the refactoring transformed message sends `evaluate` but `evaluateWith: Dictionary new` and this even in methods `evaluate`.

```
EAddition >> evaluateWith: anObject
  ^ (right evaluateWith: Dictionary new) + (left evaluateWith:
    Dictionary new)
```

This method should be transformed as follows: We should pass the binding to the argument of the `evaluateWith:` recursive calls.

```
EAddition >> evaluateWith: anObject
  ^ (right evaluateWith: anObject) + (left evaluateWith: anObject)
```

Do the same for the multiplications.

```
ENegation >> evaluateWith: anObject
  ^ (expression evaluateWith: anObject) negated
```

Figure 11-11 shows the final situation.

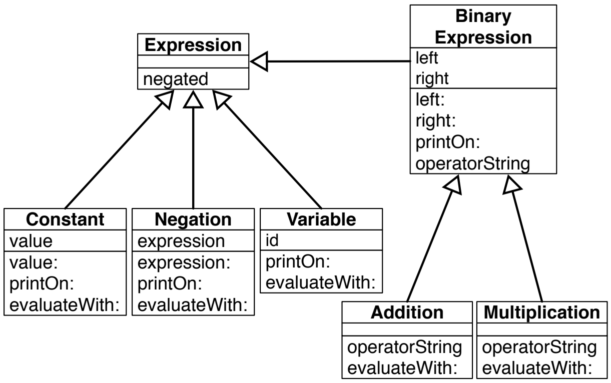


Figure 11-11 Variables and their evaluation.

11.17 Conclusion

This little exercise was full of learning potential. Here is a little summary of what we explained and we hope you understood.

- A message specifies an intent while a method is a named list of execution. We often have one message and a list of methods with the same name.
- Sending a message is finding the method corresponding to the message selector: this selection is based on the class of the object receiving the message. When we look for a method we start in the class of the receiver and go up the inheritance link.
- Tests are a really nice way to specify what we want to achieve and then to verify after each change that we did not break something. Tests do not prevent bugs but they help us build confidence in the changes we make by identifying fast errors.
- Refactorings are more than simple code transformations. Usually refactorings pay attention to their application does not change the behavior of program. As we saw refactorings are powerful operations that really help do complex operations in a few action.
- We saw that the introduction of a new superclass and moving instance variables or methods to a superclass does not change the structure or behavior of the subclasses. This is because (1) for the state, the structure of an instance is based on the state of its class and all its superclasses, (2) the lookup starts in the class of the receiver and look in superclasses.
- Each time we send a message, we create a potential place (a hook) for subclasses to get their code definition used in place of the superclass's

one.

CHAPTER 12

Understanding visitors

In a previous chapter, you built a simple mathematical expression interpreter. You were able to build an expression such as `(3 + 4) * 5` and then ask the interpreter to compute its value. In this chapter we will introduce Visitors. A Visitor is a way to represent an action on a structure (often a tree) as its own object. The action can be complex and need its own specific state. What is nice about a visitor is that it embeds its own state and behavior which would be otherwise mixed with the ones of the structure and other actions. In addition we can have multiple visitors visiting the same structure without mixing their concerns. Finally a visitor is modular because you may execute one and not another one or even load another one.

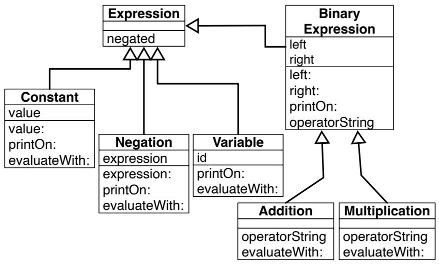


Figure 12-1 A simple hierarchy of expressions.

You will build two simple visitors that evaluate and print an expression.

Let us start with the previous situation.

12.1 Existing situation: expression trees

Figure 12-1 shows the simple hierarchy of expressions that we developed in a previous chapter. We basically have the different possible parts of an expression (variable, addition, value...) represented by their own node. Each node holds some state and in addition specifies how it computes its value. This is often done by a recursive call sending message `evaluateWith:` to subexpressions.

Note that expression trees are similar to the ones that are used to manipulate programs. For example, compilers and code refactorings as offered in Pharo and many modern IDEs are doing such manipulation with trees representing code (often called Abstract Syntax Trees).

In the rest of this chapter we will introduce step by step a visitor and we will incrementally replace the recursive calls by calls to the a visitor. Doing so we will make sure that all the tests still pass.

12.2 Visitor's key principle

The previous solution is using a simple recursive process to compute the value of an expression. Now we will define the evaluation using a visitor.

The key principle about visitor is the following one: a visitor declares to a structure that it wants to visit it (i.e., apply a treatment to it) and then the structure replies by indicating to the visitor how this visitor should visit it. This interaction is a double dispatch: it means that given a visitor and a structure, the correct method will be executed without having to explicitly test the class of the structure.

You do not have to deeply understanding this now. This interaction will emerge from the exercise.

Here is a typical illustration: The class `EConstant` defines the method `accept:` to say to the visitor that it should visit the expression using the message `visitConstant:`.

```
EConstant >> accept: aVisitor
  ^ aVisitor visitConstant: self
```

The visitor defines the specific action that he will perform:

```
EEvaluatorVisitor >> visitConstant: aConstant
  ^ aConstant value
```

Here is how the interaction starts: We ask the structure to accept a visitor.

```
| constant |
constant := EConstant value: 5.
constant accept: EEvaluatorVisitor new.
```

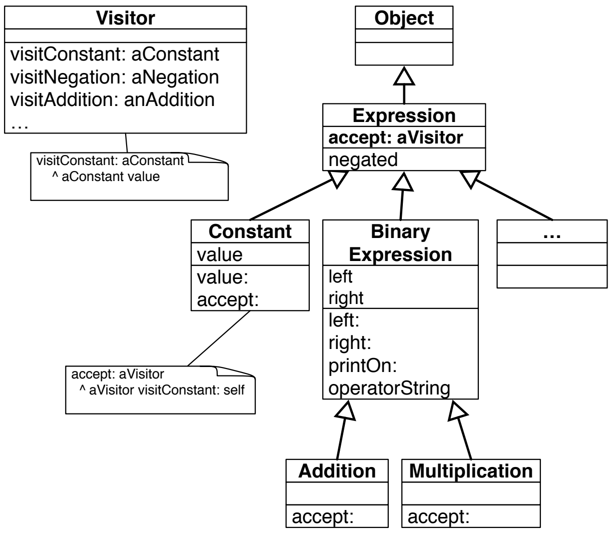


Figure 12-2 Visitor principle.

Let us step by step implement an evaluating visitor.

12.3 Introducing an evaluating Visitor

We start by adding an abstract method `accept:` in the `Expression` class to document that any expression can *welcome* a visitor and tells it how to react.

Here is the definition of the the abstract method `accept:`:

```
EExpression >> accept: aVisitor

  self subclassResponsibility
```

Now we take a concrete node expression: we start with constant expressions. When the visitor visit a constant, the constant tells the visitor that it should visit the constant as a constant. This is literaly what the following method is doing.

```
EConstant >> accept: aVisitor

  ^ aVisitor visitConstant: self
```

Defining the visitor class

Now it is time to define class representing the evaluating visitor.

```
Object subclass: #EEvaluatorVisitor
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions-Model'
```

Once the class is created we can define what is it to visit a constant expression. This is simple, it is just to return the constant value. We define the `visitConstant:` as follows:

```
EEvaluatorVisitor >> visitConstant: aConstant

  ^ aConstant value
```

Adding a test class

To make sure that we control what we are doing, we add a test class.

```
TestCase subclass: #EEvaluatorVisitorTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions-Test'
```

We are ready to write our first test

```
EEvaluatorVisitorTest >> testVisitConstantReturnsConstantValue

  | constant result |
  constant := EConstant value: 5.
  result := constant accept: EEvaluatorVisitor new.
  self assert: result equals: 5
```

We can rewrite the old method `evaluateWith:` method to invoke the visitor.

```
EConstant >> evaluateWith: anObject

  ^ self accept: EEvaluatorVisitor new
```

You can execute your new and old tests and both should work. Note that once the visitor is in place, we will remove this method and only define it once in the superclass.

12.4 Now handling addition

We will do the same with adddition. First we define a new `accept:` method on the `Addition` class to say to the visitor which method it should execute on the structure.

```
EAddition >> accept: aVisitor

  ... Your code ...
```

Notice again that the visitor announces itself and that the addition tells it that it should be treated this time as an addition. This pattern is key to the visitor logic. You will see that we will repeat again and again. Each expression will declare how it should considered by the visitor.

Adding a new test

Now we can define a new test to validate that the execution of an addition is correct.

```
EEvaluatorVisitorTest >> testVisitAdditionReturnsAdditionResult

  | expression result |
  expression := EAddition
    left: (EConstant value: 7)
    right: (EConstant value: -2).
  result := expression accept: EEvaluatorVisitor new.
  self assert: result equals: 5
```

We create the accessors `left` and `right`.

```
EBinaryExpression >> left
  ^ left
```

```
EBinaryExpression >> right
  ^ right
```

Defining visitAddition:

Now we are ready to define the method `visitAddition:` so that it adds the value returned by each sub expression:

```
EEvaluatorVisitor >> visitAddition: anEAddition
  ... Your code ...
```

The method `visitAddition:` should pass the visitor to each subexpression. And once each value is known the visitor will perform the addition.

We also redefine the method `evaluateWith:` to use the visitor. As you recognize it, it is the same as in the class `EConstant`. We will remove it later.

```
EAddition >> evaluateWith: anObject
  ^ self accept: EEvaluatorVisitor new
```

Again all your new and old tests should pass.

12.5 Supporting negation

We will focus on the negation. Again we start by defining a test method.

```
EEvaluatorVisitorTest >> testVisitNegationReturnsNegatedConstant

  | expression result |
  expression := (EConstant value: 7) negated.
  result := expression accept: EEvaluatorVisitor new.
  self assert: result equals: -7
```

We follow the same process. We define the `accept:` method for the negation.

```
ENegation >> accept: aVisitor
  ... Your code ...
```

We add the `expression` accessor.

```
ENegation >> expression
  ^ expression
```

Defining visitNegation:

We define the `visitNegation:` as follows:

```
EEvaluatorVisitor >> visitNegation: anENegation
  ... Your code ...
```

What you should see is that again the method `visitNegation:` is invoking the visitor on a subexpression, here the negated expression.

Again redefining evaluateWith:

We redefine the `evaluateWith:` method on a negation to invoke the visitor.

```
ENegation >> evaluateWith: anObject
  ^ self accept: EEvaluatorVisitor new
```

12.6 Supporting Multiplication

You start to get it and we will do exactly the same for multiplication.

Adding a test

```
EEvaluatorVisitorTest >>
    testVisitMultiplicationReturnsMultiplicationResult

  | expression result |
  expression := EMultiplication
    left: (EConstant value: 7)
    right: (EConstant value: -2).
  result := expression accept: EEvaluatorVisitor new.
  self assert: result equals: -14
```

Defining the accept: method

We define the `accept:` method on the `Multiplication` class.

```
EMultiplication >> accept: aVisitor

  ... Your code ...
```

Defining the visitMultiplication

We are not ready to define the method `visitMultiplication:` on the evaluating visitor. Its logic is similar to the one of the addition: get the value of the children and multiplying it.

```
EEvaluatorVisitor >> visitMultiplication: anEMultiplication

  ... Your report ...
```

Figure 12-3 describes the situation.

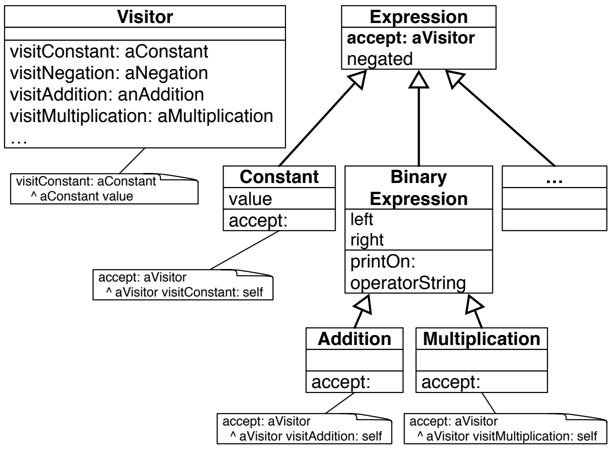


Figure 12-3 Visitor at work.

12.7 Supporting Division

As you can guess the logic is exactly the same to support division. You should start to get the pattern.

First two tests

```
EEvaluatorVisitorTest >> testVisitDivisionReturnsDivisionResult

  | expression result |
  expression := EDivision
    numerator: (EConstant value: 6)
    denominator: (EConstant value: 3).
  result := expression accept: EEvaluatorVisitor new.
  self assert: result equals: 2
```

```
EEvaluatorVisitorTest >> testVisitDivisionByZeroThrowsException

  | expression result |
  expression := EDivision
    numerator: (EConstant value: 6)
    denominator: (EConstant value: 0).
  self
    should: [expression accept: EEvaluatorVisitor new]
    raise: EZeroDenominator
```

Improving the creation API

We introduce the class message `numerator:denominator:` to ease division creation.

```
EDivision class >> numerator: aNumeratorExpression denominator:
    aDenominatorExpression

  ^ self new
    numerator: aNumeratorExpression;
    denominator: aDenominatorExpression;
    yourself
```

We define accessors so that the visitor can access to subexpression.

```
EDivision >> numerator
  ^ numerator
```

```
EDivision >> denominator
  ^ denominator
```

Defining accept:

Then we define the method `accept:` for divisions.

```
EDivision >> accept: aVisitor

  ... Your code ...
```

Defining the visitDivision:

We define the `visitDivision:` method as follows. It is similar to others. In addition here we prevent division by Zero and raise an exception instead.

```
EEvaluatorVisitor >> visitDivision: aDivision
  ... Your code ...
```

12.8 Moving up evaluateWith:

Since we get bored to always redefine the method `evaluateWith:` we define it in the superclass, the class `Expression` and we remove it from all the subclasses except `Variable` since we will still have to transform it.

```
EExpression >> evaluateWith: anObject

  ^ self accept: EEvaluatorVisitor new
```

12.9 Supporting variables

Now we can focus on supporting variable in the expression. The following test show that we can have an expression which is a variable (here named `answerToTheQuestion`) and that we can set the value of this variable using the message `at:put:`. The test then shows that when we are evaluating the expression we should get the corresponding value, (here 42).

```
EEvaluatorVisitorTest >> testVisitVariableReturnsVariableValue
  | expression result visitor |
  expression := EVariable id: #answerToTheQuestion.

  visitor := EEvaluatorVisitor new.
  visitor at: #answerToTheQuestion put: 42.

  result := expression accept: visitor.
  self assert: result equals: 42
```

Extending the visitor state

To support variable the visitor should hold a kind of environment with the value of each variable. We introduce an instance variable named `bindings`. This is a good example that shows that a visitor is the natural place to store state about the specific behavior it represents.

```
Object subclass: #EEvaluatorVisitor
  instanceVariableNames: 'bindings'
  classVariableNames: ''
  package: 'Expressions-Model'
```

We initialize this variable to a dictionary.

```
EEvaluatorVisitor >> initialize

  super initialize.
  bindings := Dictionary new
```

We define a little helper to set the value of a variable.

```
EEvaluatorVisitor >> at: anId put: aValue
  bindings at: anId put: aValue
```

We define a class method `id:` to name a variable.

```
EVariable class >> id: anId

  ^ self new id: anId; yourself
```

Visiting a variable

We have to define a method `accept:` on the class `EVariable`.

```
EVariable >> accept: aVisitor
  ... Your code ...
```

Now we are ready to define the meaning of evaluating a variable. The method `visitVariable:` of the `EEvaluatorVisitor` is responsible of this.

```
EEvaluatorVisitor >> visitVariable: aVariable

  ... Your code ...
```

12.10 Redefine evaluateWith:

We modify the method `evaluateWith:` to make sure that the initial bindings are stored in the visitor.

```
EExpression >> evaluateWith: anEnvironment

  | visitor |
  visitor := EEvaluatorVisitor new.
  visitor bindings: anEnvironment.
  ^ self accept: visitor.
```

```
EEvaluatorVisitor >> bindings: aDictionary

  bindings := aDictionary
```

12.11 A new visitor

Using a visitor is particularly interesting when we have multiple behavior that we want to encapsulate. Such behaviors are applied on a structure without mixing the state of the structure with the state of the behavior or mixing multiple behaviors together.

Now that each kind of expression is declaring in its respective methods how a visitor should visit it, other visitors can be easily expressed. And this is what we will show now.

Defining a new visitor

Now we show how we can have another visitor, an expression printer. Let us define the following class.

```
Object subclass: #EPrinterVisitor
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions-Model'
```

Define some tests to make sure that you are getting the correct results. We let you do it.

```
TestCase subclass: #EPrinterVisitorTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions-Model'
```

12.12 Visiting methods

We start by defining some typical visit methods as follows:

```
EPrinterVisitor >> visitConstant: aConstant
  ^ aConstant value asString
```

```
EPrinterVisitor >> visitMutiplication: aMultiplication

  | left right |
  left := aMultiplication left accept: self.
  right := aMultiplication right accept: self.
  ^ '(', left , ' * ', right, ')'
```

Now you should be in position to finish the implementation.

```
EPrinterVisitor >> visitAddition: anAddition
  ... Your code ...
```

```
EPrinterVisitor >> visitDivision: aDivision
  ... Your code ...
```

```
EPrinterVisitor >> visitNegation: aNegation
  ... Your code ...
```

```
EPrinterVisitor >> visitVariable: aVariable
  ... Your code ...
```

12.13 Conclusion

In this chapter we show how you can pass from a behavior inside a class hierarchy to a separate object and how once this architecture is in place (basically the `accept:` methods) other visitors can be easily expressed.

The visitor pattern is a nice design. It supports encapsulate behavior on complex structure. In addition it lets users develop their own functionality indepently of others.

Now you should pay attention not to over use it. It is also more suitable for systems whose domain does not change because else each time you add a kind of object in your composite (here the expression) you would have to touch each visitor.

Part III

Unguided exercises

12.13 Conclusion

In this part, we propose two less-guided projects and multiple extensions to the previous projects. It is fun to challenge ourselves to see how we could support the proposed variations.

CHAPTER 13

Tamagotchi Mechanics

13.1 Problem Context

Bandai, the creator of the Tamagotchi game hires us to create a new version of the game. We have to implement the behavior logic behind the simulated pets. In this new version, we are going to have two Tamagotchis interacting with each other and performing some basic activities together.

Our virtual pet can be sad, hungry, or happy. And its behavior depends on its state. The state of the virtual pet changes only when an action is performed.

A new pet starts with a happiness level of 0 and it is in a happy state. If a pet is already in a given state, nothing should happen if it is required to change to that state (e.g., if it is happy, and I tell the pet to be happy, nothing should change in its state).

13.2 Behavior

You will have to implement the following behavior.

Eating.

A pet can eat then:

- If it is hungry, it gets happy.
- If it is happy, it increases its happiness level by one.
- If it is sad, it will not eat, and an error should happen.

Playing Alone.

A pet can play with itself then:

- If it is hungry, its happiness level is reduced by 4.
- If it is happy, it increases its happy level by two. If it plays two times since eating,

it gets hungry.

- If it is sad, it gets happy.

Playing With Other Pets.

A pet can play with another pet then:

- If any of them is hungry, they don't play and nothing changes.
- If the pet is happy, it increases its happy level by four. If it plays two times since

eating, it gets hungry.

- If any of them is sad, it gets happy.

13.3 Extensions

As the game got so popular, we have to add new mechanics to it. We are going to add two additional types of Pets.

Dog.

- Whenever it eats, its happiness level rises to 5, then it rises normally by 1.

Lonely Cat.

- Whenever it plays with another pet, it gets sad.

13.4 Tests

It is required to implement a set of test cases covering all features. It is important to select what test cases to write, having an excessive number of dummy or uninteresting tests is not good. Test cases should cover the different states and the possible actions. For the added extensions is only needed to test the variation of behavior, e.g., for the Lonely Cat is only needed to make it play.

13.5 About behavior description

All the rules, descriptions, and requirements are open to different understanding. If a point is unclear or it raises different alternatives, you are free to choose the one that better suits you. The only condition is to explain it and that test cases cover the variation.

CHAPTER 14

Civilization

In this chapter you will design a simplified version of the combat mechanics in turn-based strategy games such as Civilization or FreeCiv as an open-source alternative (See https://en.wikipedia.org/wiki/Freeciv).

14.1 General Rules

Here are some general rules:

- We have a game board made of a grid of tiles, each tile may contain (or not) one unit.
- Each unit has a given type (e.g. Archer, Marine, Pikeman, Knight, etc), a health and experience level.
- The health level is a percentage and goes from 0% to 100%.
- The experience level is a value from 1 to 3.
- Units in neighbor tiles can attack each other.
- The attacking unit will provoke damage to the defending unit and also it will receive damage from it.
- The amount of damage received from each of the units is calculated independently using two math formulas, one for the attacker and one for the defender.
- After the battle, the health of the attacker will be reduced by the amount calculated and the same will happen to the defending unit.
- If the attacking unit gets to zero or lower health, it should be removed from the board.

- If the defending unit gets to zero or lower health it should be removed from the board.
- If the defending unit is removed from the board, the attacking unit will move into the tile occupied by the defending unit.

14.2 Defending Unit Damage Formula

The calculation of the impact on the defending unit is calculated by the formula by the following formulae: `Dd = Ap * Atm + Dti`

Defending Unit Damage (Dd).

It is the impact on the health of the defending unit. This value is calculated by the formula Figure 1 and depends on the attacking unit's offensive power, the attacking unit's type and the terrain occupied by the defending unit.

Attacking Unit Power (Ap).

It is the power of the attacking unit type multiplied by the factor of experience. The factor of experience is equal to 1 for level 1, 1.5 for level 2, and 2 for level 3.

Defending / Attacking Unit Types Combination Modifier (Atm).

This value depends on the combination of the types of defending and attacking units. Some units have benefits when fighting against other units. For example, If a Knight attacks a Warrior unit, the Knight has a modifier of 2, as the Knight can charge a Warrior unit. If a Knight attacks a Pikeman unit, the Knight has a modifier of 1 as the Pikeman can repeal the charge of the knight. For a complete list, see the table of units (Section 3).

Defending Unit Terrain Impact (Dti).

The combat is performed in the tile occupied by the defending unit. Each tile has a given type of terrain, and each type of terrain provides its own Defending Unit Terrain Impact value. For a complete list, see the table of terrains (Section 4).

14.3 Attacking Unit Damage Formula

The calculation of the impact on the attacking unit is calculated by the formula: `Ad = Dp * Dtm + Ati`

tacking Unit Damage (Ad).

It is the impact on the health of the attacking unit. This value is calculated by the formula Figure 2 and depends on the defending unit's defensive power, the defending unit's type and the terrain occupied by the defending unit.

Defending Unit Power (Dp).

It is the power of the defending unit type multiplied by the factor of experience. The factor of experience is equal to 1 for level 1, 1.75 for level 2, and 2.5 for level 3.

Defending / Attacking Unit Types Combination Modifier (Dtm).

This value depends on the combination of the types of defending and attacking units. Some units have benefits when fighting against other units. For example, if a Knight attacks a Pikeman unit, the Pikeman has a modifier of 2 as it can repeal the charge of the knights. If a Knight attacks a Warrior unit, the warrior has a modifier of 1, as it does not have any advantage in the defense. For a complete list, see the table of units (Section 3).

Attacking Unit Terrain Impact (Ati).

The combat is performed in the case occupied by the defending unit. Each case has a given type of terrain, and each type of terrain provides its own Attacking Unit Terrain Impact value. For a complete list, see the table of terrains (Section 4).

14.4 Units

We consider the following units: warrior, archer, pikeman, and knight.

Warrior.

A basic foot soldier holding a sword

- Attacking Unit Power: 10
- Defense Unit Power: 10
- Relations with other units: N/A

Archer.

A basic foot soldier with a bow and arrows.

- Attacking Unit Power: 20

- Defense Unit Power: 5
- Relations with other units: N/A

Pikeman.

A soldier armed with a Pike. Effective against charges from the Knights.

- Attacking Unit Power: 5
- Defense Unit Power: 20
- Relations with other units: When it is attacked by a knight, it has a Dtm

value of 2.

Knight.

The Knight is a heavy cavalry unit.

- Attacking Unit Power: 20
- Defense Unit Power: 5
- Relations with other units: When charging on any non-Pikeman unit, it has a Atm value of 2.

14.5 Terrains

We consider the following terrains:

Flat Terrain.

This terrain does not have an impact on the combat. It has Ati and Dti equals to 0.

Hilly Terrain.

This terrain makes it easier for the attacking part. It has an Ati of 0 and a Dti of 10. If the attacking unit is a Knight, the Dti is doubled.

14.6 Tests

You should at least implement tests for the following use cases. Note that there should be asserts for the health of the units, and the discounted points for each of them.

- Make combat two warrior units of level 1 in a flat terrain.

- Make a Knight (level 2) attack a Warrior (level 3) in hilly terrain.
- Make a Knight (level 2) attack a Pikeman (level 3) in hilly terrain.
- Make a Knight (level 2) attack a Pikeman (level 3) in a flat terrain.
- Make an Archer (level 1) attack a Pikeman (level 3) in a flat terrain.
- Make an Archer (level 1) attack a Pikeman (level 3) in a hilly terrain.

CHAPTER 15

Little unguided projects

In this chapter, we present a list of small projects that we encourage you to code. Such projects are fun and playing with them will force you to practice different coding idioms or design patterns. In many exercises, you should avoid relying on conditionals.

15.1 LAN simulator

As shown in Chapter 5, a LAN is composed of different kinds of nodes. Packets circulate inside the LAN from node to node.

A node has an address and a next node to which it forwards packets that are not addressed to it. When the packet is addressed to a node, then depending on the node kind it performs its action. A packet has an address and contents.

A simple LAN is composed of simple nodes. Simple nodes just forward the packets that are not addressed to them to their next node. Check the tests that are provided to get an understanding.

15.2 Loading LAN code.

If you did not follow Chapter 5, the code is available at:

```
Metacello new
  baseline: 'SimpleLAN';
  repository: 'github://Ducasse/SimpleLAN/src';
  load.
```

15.3 LAN extensions

There are several possible extensions:

Hook for accept

Subclasses such as `LNWorkstation` or `LNPrinter` redefine the `accept:` method and systematically check that the packet is sent to the receiver to perform a specific action else they pass it to the next node. The following method illustrates this behavior:

```
LNPrinter >> accept: aPacket

  (aPacket isAddressedTo: self)
    ifTrue: [ 'Node ' , aPacket originatorName , ' sent to printer: '
    , aPacket contents traceCr ]
  ifFalse: [ super accept: aPacket ]
```

This repetition indicates that this behavior could be factored out in the superclass. So define such behavior in the superclass and introduce a new method called `treatPacket:`. Such a new method will be redefined in all the subclasses.

Understanding execution flow

Better way to follow the execution of the simulation. In the default implementation, the `accept:` method writes on the Transcript. This is not good since the trace is mixed with other program traces. (Note that the Transcript is a global variable shared by the complete environment). In addition, it is not easy to write tests for the `send:` and `accept:` methods. One solution is to have a stream shared by all the nodes of a simulation, stream on which all the messages written to the Transcript are now written to. Propose one solution. To validate your solution, write some tests for `send:` and `accept:`. They should be able to run without writing to the Transcript.

Star node

Introduce a star node that supports the creation of star network. A star node does not have one but many following nodes.

Avoiding some conditions.

The way the instance variable `nextNode` is managed (having nil or a node as value) is based on conditions in different places: the method `accept:` or the method `send:`. Propose a solution to be able to remove such conditions. One possible solution is to apply the NullObject design pattern.

Trottling node

This node accumulates packets and waits before forwarding them. They wait to have received a certain number of packets before sending a batch of collected packets.

Handling loops

Introduce an origin to the packet and make sure that if it reaches the node that emitted it is not propagated.

Limited packet distance

We decided that a packet can only be forwarded a given number of times. After that, it should reemitted or is not propagated if it did not reach its destination. Introduce a repeating node that recharges the distance number of packet can have.

Different kinds of packets

In this extension we introduce several kinds of packet.

- Some packets can auto replicate themselves to flood the LAN, it means that when they are accepted by a node and even if they are addressed to a node receiving them they force their forwarding.
- Urgent packets cannot be trottled by trollting nodes.

Signing node

A signing node encodes the contents of a package and only nodes with the same signing behavior can decode the contents. We would like to have different combinations of contents encodings. One possible design is to use a Decorator design pattern, where the decorators will expose the same API that the packet but they implement different encodings. Here is a list of encodings:

- adding one to each character of the contents,
- substituing one character by another one based on a map, or
- uuencoding - Check the base64 encoder available in Pharo.

15.4 Die players

We want to model dice (as in the DSL chapter) but with different kinds of players and dices. Let us imagine that we have die and die handle and that we can roll a die and a die handle.

A normal player is one player that roll normally its die handle. Introduce the class Player and make sure that it gets the possibility to roll die handles.

Kind of players

Now we want to introduce different kind of players.

- A cheater player is one player that will take the max of value of its die handle value and multiply by the number of dices. For example if he gets 2,3,4, after rolling its dice, he will say that he hot 4,4,4.
- A lucky player is a player that will always have plus one to its roll. For example, rolling its dice returns, 2,3,4 and the lucky player will have 10 and not 9.
- A super lucky player is a player that will always have plus one to the values of all the value of its dices. For example, rolling its dice returns 2,3,4 and the super lucky player will return 3,4,5 i.e. 12.

Different kinds of die

Now we introduce different kinds of dice.

- A normal die has equi proportional values from 1 to its max face number: e.g., 1, 2, 3, 4, 5, 6.
- A middle die has no 1 and 6 but two 3 and two 4: e.g. 2, 3, 3, 4, 4, 5.
- A cheated die has no 1 and 2 but 3 6 values: e.g. 3, 4, 5, 6, 6, 6.

Pairs of player and die

Now certain dices can only be played by certain player:

- A middle die can only be played by a lucky or super lucky player.
- A cheating dice that can only be played by a cheater.

15.5 About the die DSL

In Chapter 8 we introduce double dispatch to mix die and dice handles, we can implement the expected behavior without double dispatch. Do it.

15.6 About Visitors

Pharo has a Visitor library for its Abstract Syntax Tree (AST). Browse the class `RBProgramNodeVisitor`. This class is an abstract class. It defines all the methods available for building specialized Visitors. All the nodes in an

AST are subclasses of RBProgramNode. The following shows the core elements where the indentation reflects the inheritance.

```
RBProgramNode
  RBComment
  RBMethodNode
  RBPragmaNode
  RBReturnNode
  RBSequenceNode
  RBValueNode
    RBArrayNode
    RBAssignmentNode
    RBBlockNode
    RBCascadeNode
    RBLiteralNode
      RBLiteralArrayNode
      RBLiteralValueNode
    RBMessageNode
    RBSelectorNode
    RBVariableNode
      RBArgumentNode
      RBGlobalNode
      RBInstanceVariableNode
      RBSelfNode
      RBSuperNode
      RBTemporaryNode
      RBThisContextNode
```

The following script shows how to execute a visitor on the AST of the method `Point>>#degrees`.

```
MyVisitor new visit: (Point>>#degrees) ast
```

Here is a list of possible Visitors that you can simply define:

- a visitor that checks whether a method is a utility method: it does not access instance variables not self or super.
- a visitor that returns the list of instance variables accessed by the method.
- a visitor that checks all the self-message sends of a method and returns the list of the compiled method found in the class or its superclass. You can use the method `lookupSelector:` defined on `Class` to find the corresponding method.
- a visitor that adds a return to the expression given. For SequenceNode it will put the return node on the last statement of the sequence node.

Part IV

Unguided Projects

In this part, we propose you design some simple board games using the Bloc graphical framework taking as an example the games of the Myg project.

CHAPTER 16

Designing little board games

This chapter lists some game descriptions. The idea is to be able to use elements of such a list as design exercises. We suggest to focus first on designing the model of the game with tests. In a second step you can add an UI layer based on the new Bloc framework.

16.1 Support

All the games in this chapter requires a 2D grid that can be found in the package Array2D of the following http://github.com/Pharo-contribution repository.

```
Metacello new
  baseline: 'ContainersArray2D';
  repository: 'github://pharo-containers/Containers-Array2D/src';
  load.
```

16.2 Loading Myg and Bloc

You can get some ideas how to develop your game UI by studying the Myg framework. The Myg framework is a framework to build little 2D board games. A Miner, Sokoban, Memory, and Takuzu have been built on top of it. It provides ways to build levels and other facilities.

```
Metacello new
  baseline: 'Myg';
  repository: 'github://Ducasse/Myg:v1.0.1/src';
  onConflictUseIncoming;
  load
```

You will just need to execute `MygSokoban open` to get a little Sokoban game.

16.3 Bloc

Bloc is a new graphical library that will be part of Pharo. If you see the name Toplo, Toplo is a new widget library built on top of Bloc. It is still under heavy development.

You can find some slides:

- https://www.slideshare.net/esug/bloc-for-pharo-current-state-and-future-perspective
- http://www.github.com/pharo-graphics/Bloc
- You can find the presentations of Bloc at ESUG 2022 and 2023 on youtube.

16.4 Possible Games

Here is a list of possible games to develop:

- Light Beamer
- Mimesweeper
- Flood it
- 2048
- Memory
- Tetris
- Picross (nonogram)
- SlideOut
- SameGame
- Taquin
- Bomberman
- Maze generators

16.5 Other resources

Besides the Myg framework mentioned above, there are some resources available that you can study.

The book Learning Object-oriented Design with TDD in Pharo available on http://books.pharo.org contains a chapter building step by step a model of Snakes and Ladders.

In addition the book ”Building a memory game with Bloc” available on http://-book.pharo.org and https://github.com/SquareBracketAssociates/Booklet-BuildingMemoryGameWithBloc presents how to build a simple memory game. It does not use Myg.

In the Tutorial project of the Bloc repository there is an implementation of the 2048 game. It is in draft mode but you can get inspiration from it too.

```
Metacello new
    baseline: 'BlocTutorials';
    repository: 'github://pharo-graphics/Tutorials:dev-1.0/src';
    load
```

Note that the implementation of 2048 is a sketch and was used to brainstorm on skinning. It should be rewritten using more recent implementations of Bloc.

16.6 Some generic extensions

Here is a list of generic extensions that can be applied to many games:

- Support the definition of levels by proposing a progression in terms of difficulties or different challenges.
- Offer the possibility to replay a given level.
- Offer the possibility to save a game.
- Offer the possibility to replay a game up to a certain point.
- Record the time or the number of moves to finish a game with high score management.

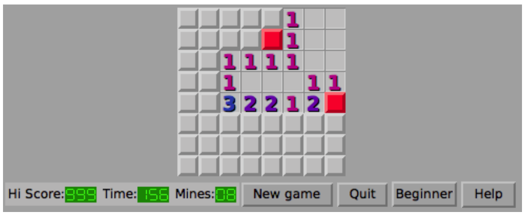


Figure 16-1 Minesweeper: identifying mines based on the number of adjacent cells containing a bomb.

16.7 Minesweeper

The user should identify the bombs using hints based on the number of adjacent bombs in the 8 directions.

The user has two kinds of action:

- declare that a cell contains a bomb or
- declare that a cell is free and ask for a validation.

Free cell declaration:

If the user is wrong and there is a bomb then he loses the game. If the user is right then the cell displays the number of adjacent bomb around the cell.

Bomb declaration:

when a cell is declared as a bomb a bomb is displayed.

When all the cells have been revealed or marked as a bomb, the game proceeds to the validation. If the bombs are correctly identified the user wins.

Since the Myg framework already propose an implementation we suggest the following possible extensions:

Specific extensions:

- Define multiple algorithms to place the bombs. Right now it is fully random. For example make sure that the user with all the information does not have to select a tile randomly.

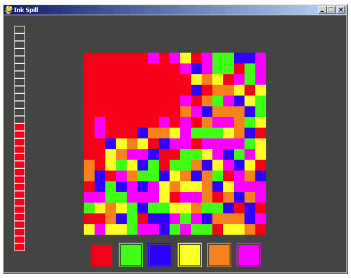


Figure 16-2 Flood it: change the color of any adjacent tiles with the same color.

16.8 Flood it

A certain configuration of tiles of different colors is placed on the board (See Figure 16-2) The player can do a ”flood fill” on the top left tile, changing the color of any adjacent tiles of the same color. The player wins if he is able to make the entire board a single color within a certain number of moves.

Specific extensions:

- you can introduce a color that matches multiple ones
- tiles that do not match any colors
- tiles that change colors every n actions

16.9 Tetris and variations

With Tetris, shapes fall from the top of the screen and lines can be eliminated only when they are entirely filled up.

Specific extensions

Here are some possible extensions:

- Add tiles that cannot be removed.
- Add shapes that shrink/expand when placed.

Using Tetris like tile shapes, many games can be built with different gameplay. Figure 16-3 shows a game where the shapes do not fall from the top of the screen but the player has to select them and drop them. When a line is full it is removed not changing the lines on top.

Specific extensions

- Adding special tiles with diamonds and others and we different scoring and objectives.
- Placing tiles that cannot be matched and removed.

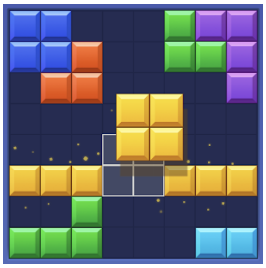


Figure 16-3 Tetris variations: Here the user places forms to remove rows and lines.

16.10 2048

The user should merge multiple of two numbers that are randomly drawn. Two numbers of the same value are merged. The resulting number of the sum replaces the sum. The user decides the merge by choosing one direction. All the numbers that can merge are merged in the chosen direction.

The game starts with 2 and continues with the sum e.g., 4, 8, 16, 64.... It should adapt the numbers that are placed on the board in the sense that it does not have to 2 when the average numbers are 512.

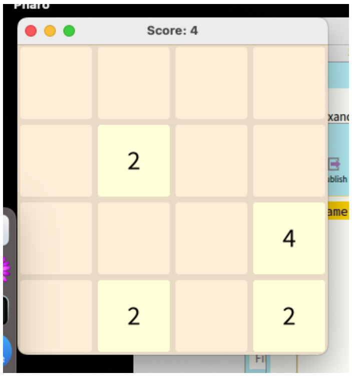


Figure 16-4 2048

The game ends when the board is full.

Specific extensions

- Merging two numbers could produce an explosion and destroy unmergeable tiles that are close to the resulting tile.

16.11 Memory

With the memory game, two pictures are revealed one by one and the user should pair them across the game. Each player gets points based on the pairs he found.

Specific extensions

- Pairing three similar tiles. Instead of identify two similar tiles, the game would have three similar tiles.
- Some tiles could be blurry and provide more points when paired.

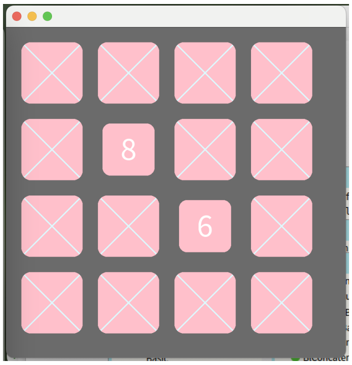


Figure 16-5 Memory

- Pairing many similar tiles could be a different game and the largest sequence could give more points: 2 for 1 points, 3 for 3 points, 4 for 6 points.
- Adding joker tiles. Joker tiles could pair with any tiles.

16.12 SlideOut

Figure 16-6 describes the game: it is composed on shapes that slide in one direction. By sliding the different pieces, the player should be able to move the red elements to the exit.

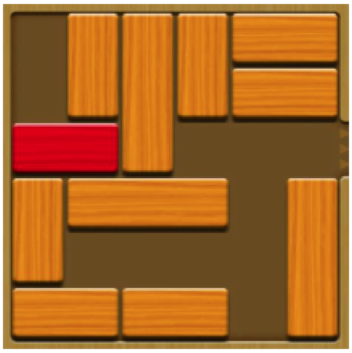


Figure 16-6 SlideOut: Elements can slide in one direction but are blocked by others. The goal is to get the red element out.

16.13 Laser game

A laser beam is activated from a source specific location. As the laser beam traverses the grid it can hit deflecting mirrors. These mirrors will divert the laser beam's direction as it travels. Ultimately the beam should hit a target location inside our grid.

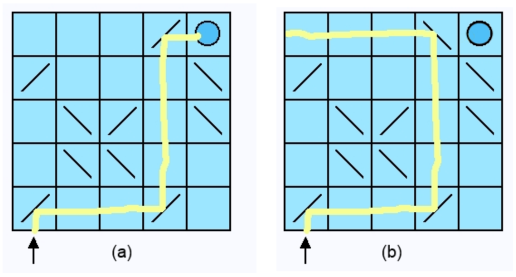


Figure 16-7 (a) Starting from its origin, the laser beam should reach the target. (b) Rotating a mirror changes the path of the laser beam

It becomes more interesting to have the user manipulate the mirrors to find the longest possible path to the target.

That's the general idea. We can add laser cell-path counters and other game instrumentation as we develop.

16.14 Same game

The goal of the game is to eliminate all the colored cells of the game. A group of connected cells of the same color are eliminated. When a column is empty, it is eliminated so that the two sides are touching each other. Figure 16-8 shows a same game instance.

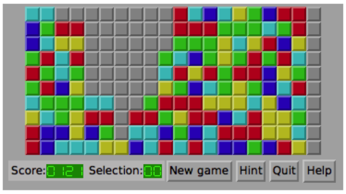


Figure 16-8 SameGame: collapsing columns by removing one by one colored of the cell of the same color.

16.15 Nonogram

Nonograms are logic puzzles where you use the number clues around the sides to color the cells in the grid to reveal a picture. The other names of nonograms are Griddler or Picross (Nintendo game).

https://delightfulpaths.com/what-are-nonograms-or-griddlers explains the rules of the game. You can see an example in Figure 16-9. Nonograms can be in black and white or with colors. The player should select a tile and declare whether it contains or not a color. The players can make a certain number of mistakes before losing the game.

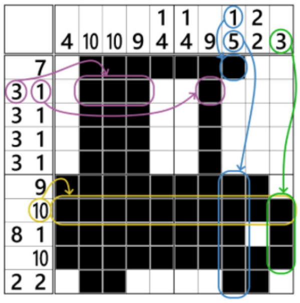


Figure 16-9 Nonograms: coloring cells based on number clues.

16.16 Conclusion

Developing such games using Testing Driven Development is pedagogically worth because each test can validate a given aspect of the game. In addition, a bunch of UML diagrams do not easily capture the essence of a game, as some new developers may think. Adding new features may invalidate certain previously passing tests and often require refactoring the model.

We definitively encourage you to play the game and develop a game and variations to exercise your object-oriented skills.

CHAPTER 17

Microdown miniprojects

Microdown is a markup language compatible with a subset of markdown. It is used by the Pharo community to produce slides, booklets, and documentation. Microdown heavily uses Visitors. The repository is at

```
https://github.com/pillar-markup/microdown
```

17.1 Blog and its posts

A nice little project is to use Microdown to define a blog and its posts. A potential roadmap is the following:

- Given a file repository we should generate a little table of contents. For this, we can reuse the HTML generation of Microdown. To do so we can do it either by generating a microdown document using the microdown builder and passing it to the HTML visitor. Or by creating the tree of objects for the table of contents and applying the HTML generation on it.
- You can also render all the post to generate HTML files.
- Finally having a visitor that extract the title of a post and a couple of line of the first paragraph so that the user can see a summary before clicking to get access to the full post can be nice.

17.2 Link checker

In Microdown, the writer can refer to figures and anchors as well as to web site as shown here after.

```
## A Section
@anchor1

![A caption](figures/fig.png label=figanchor)

[ Pharo web site ](https://pharo.org)
https://pharo.org

In Section *@anchor1@* we can find Fig. *@figanchor@*.
```

We would like to have a checker that reports to the users the set of references (defined using the `*@xxx@*` instruction that are not found.

17.3 Table of contents

We would like to have a table of content builder. Given a book, the Toc builder will generate a microdown document tree containing references to the corresponding book entities (chapter, section)

17.4 Book Sanitizer

When writing documents, we often have some writing guidelines.

Guideline sample

Here are some guidelines about writing style and spelling

- Write ”backend”, not ”back-end” or ”back end”.
- Write ”subpresenter”, not ”sub-presenter”.
- Methods without comment have an empty line after the method selector.
- Methods do not have a period on the last line.
- Write ”Pharo image”, not ”Pharo Image.
- Do not use protocol references because they are not useful and may change.
- Write ”Section 6.1”, not ”section 6.1”.
- Write ”Figure 6.2”, not ”figure 6.2”.
- Caption should start with an uppercase and terminate by a period.
- Titles (section, chapter, ...) should just have the first letter capitalized.

```
### The great book.
```

- Figure caption should end with a period and be capitalized.

```
![The great figure.]()
```

- Class names should be surrounded by ‘
- Only use tab for code indentation
- For paragraphs terminate the label with a period

```
#### Cap.
```

Job

A book sanitizer can perform modifications of the document tree to reflect them. A subsequent version could change the files to reflect such change so that the user can save them.

A sanitizer should be configurable to take into account book guidelines.

17.5 Automatic Numbering

When writing a book users do not want to be forced to write in the exact same level of nesting than the template interpretation. For example, in this book # is to represent a LaTeX part, and ## for a chapter. However, it would possible to simply change the numbering so that a writer can write # for title and to use meta data at the level of the file to control this.

```
{  "nesting":0 }
```

could mean that # is for a chapter title, ## a section, ### a paragraph

Conversely

```
{  "nesting":1 }
```

could mean that ## is for a chapter title, ### a section,

17.6 Rendered math downloader

To render math in non latex mode, the microdown renderer in Pharo performs a request to an online service. A cool feature is to change the logic to do change the logic so that

- After each request to the server, save the gif or png representing the render math expression with a unique name in the ressources folder of the current file directory. A possible name is to take two letters of path elements plus a counter. So the 3 expressions in foo/bar/ will be named foba3.png
- [optional] Change the math code to reflect that the corresponding rendered expression is available

```
$$
    \frac{1}{2}
$$
```

into

```
$$ % renderedAs=foba3.png
    \frac{1}{2}
$$
```

- Modify the system so that the request is only performed if there is no ressources for the given expression. When the ressources is found, it should use the corresponding rendered result graphic object.

Better contents encoding

Note that computing the name based on the order of appareance in the text is not really robust to change. If the user inserts a new math expression this will invalidate the pre existing renderer. Propose a better system based on the expression contents for example the hash of a zip.