Patterns of Design

Stéphane Ducasse

March 12, 2024

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

Contents

Illustrations

CHAPTER 1

About this book

1.1 Structure of the book

We will start with the exploration of message passing. As we described in *Learning Object-Oriented Programming, Design and TDD with Pharo* sending a message is making a choice. The execution engine selects and executes for us the correct method. Such mechanism is really powerful and we will explore what we can then do by just sending messages. To make sure that you can read this book in isolation, I will reuse two chapters from *Learning Object-Oriented Programming, Design and TDD with Pharo* available at http://books.pharo.org: First the *Crafting a simple embedded DSL with Pharo* chapter and second with the *A little expression interpreter* chapter.

- Reusing and extending the *Crafting a simple embedded DSL with Pharo*. We will extend the Dice mini system to support more additions between die and die handle. We will explore the notion of double dispatch. Double dispatch is a subtle notion that requires time to master. Indeed even when we believe that we fully understand it, our old deamons can push us to blindly use conditionals when this is not needed.
- Reusing and extending *A little expression interpreter*, we will as a basis to explore the Visitor design pattern. In fact the Visitor is a generalisation of double dispatch.

1.2 Typographic conventions

Pharo expressions or code snippets are represented either in the text as `'Hello'` and `'Hello' reversed`, or for more substantial snippets, as follows:

```
'Hello'
```

When we want to show the result of evaluating an expression, we show the result after three chevrons >>> on the next line, like so:

```
'Hello' reversed
>>> 'olleH'
```

Whenever we feel the text makes a point that is important or technical enough to be highlighted, we will do so with a thick bar: Important! This is a point that is worth drawing some more attention.

1.3 Thanks

I would like to thank the following persons for their feedback: Roelof Wobben.

Part I

Getting started

CHAPTER 2

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-faces 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 operator 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).

2.1 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 subclass: #DieTest
  instanceVariableNames: ''
  classVariableNames: ''
  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 2-1 gives an overview of the messages.

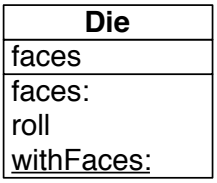

Figure 2-1 A single class with a couple of messages. Note that the method `withFaces:` is a class method.

```
Object subclass:
  ... 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).

2.2 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 2-2. With it you can interact with a die by writing expression in the bottom pane.

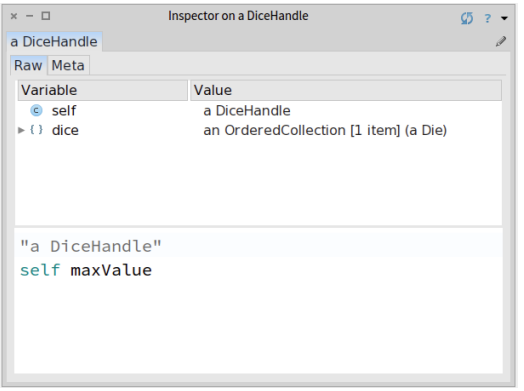

Figure 2-2 Inspecting and interacting with a die.

2.3 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 value comprised between 1 and 6. This is what the following test method is actually specifying.

```
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 illustrate their correct behavior. It helps you to program your solution.

2.4 Instance creation interface

We would like to get a simpler way to create `Die` instances. For example we want to create a 20-faces 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 sent 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 new 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 on `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.

2.5 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 2-3). We create a new handle then add some dice to it.

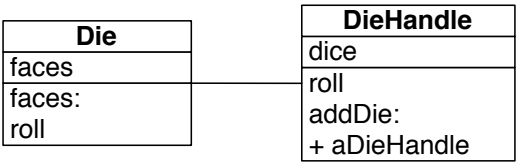

Figure 2-3 A die handle is composed of dice.

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

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

```
TestCase subclass: #DieHandleTest
  instanceVariableNames: ''
  classVariableNames: ''
  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!

2.6 Defining the DieHandle class

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

```
Object subclass: ...
  ... 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 2-4

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

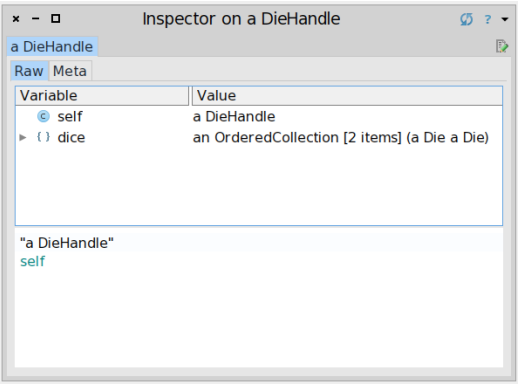

Figure 2-4 Inspecting a DieHandle.

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.

2.7 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 in 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 to provide 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 argument. It is in this stream that we should add information. We use the message `nextPutAll:` to add a number of 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 2-5 and it is now easier to check the dice contained inside a handle (See Figure 2-6).

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 argument. A better solution is to rewrite `printOn:` as follows:

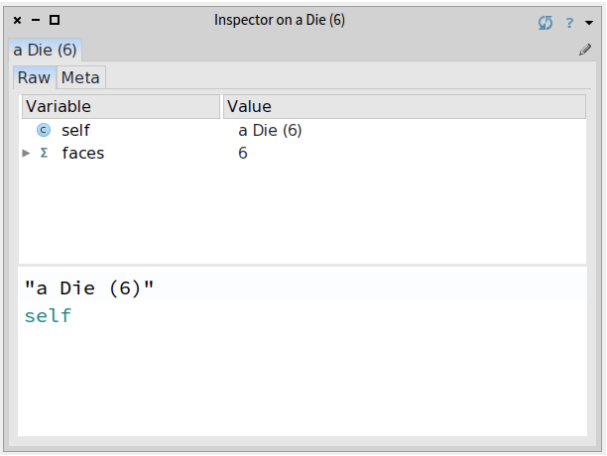

Figure 2-5 Die details.

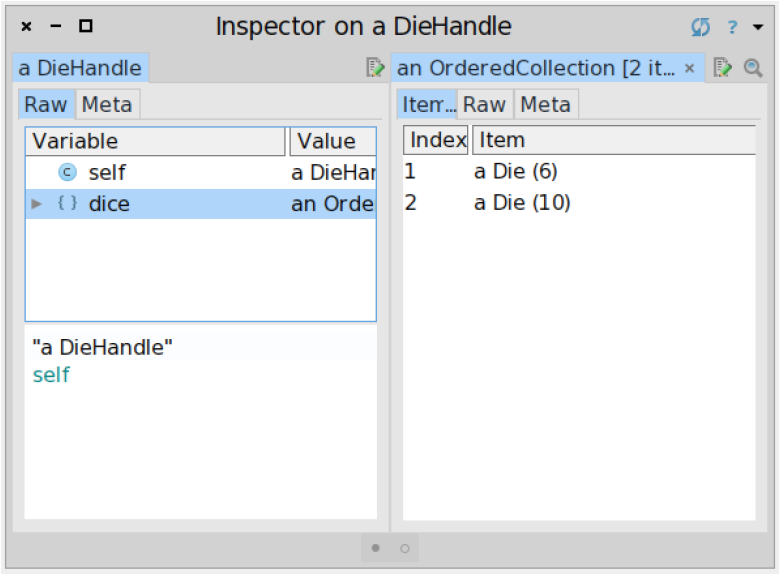

Figure 2-6 A die handle with more information.

```
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`.

2.8 Rolling a die handle

Now we can define the rolling of a die handle by simply summing 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 ...
```

2.9 About Dice and DieHandle API

It is worth to spend some times 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 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 2-7). The client is not concerned by the fact that the receiver

is composed out a simple object or a complex one. Such design decision supports the *Don't ask, tell* principle.

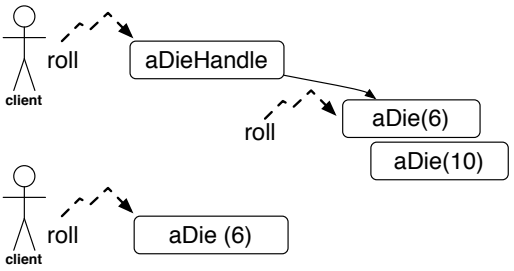

Figure 2-7 A polymorphic API supports the *Don't ask, tell* principle.

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).

```
| 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 3.

2.10 Handle's addition

Now what is missing is that 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 on the classes we want.

Now we should ask ourself 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 passed as 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.

2.11 Role playing syntax

Now we are ready to offer a syntax following practice of role playing game, 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 will be 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 faces number: 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.
```

2.12 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 class 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 3

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, in the volume two of this book, we will extend this example to present the Visitor Design Pattern.

3.1 Starting with constant expression and a test

We start with 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 subclass: #EConstantTest
  instanceVariableNames: ''
  classVariableNames: ''
  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 subclass: #EConstant
  instanceVariableNames: 'value'
  classVariableNames: ''
  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.

3.2 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 subclass: #ENegationTest
  instanceVariableNames: ''
  classVariableNames: ''
  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 subclass: #ENegation
  instanceVariableNames: 'expression'
  classVariableNames: ''
  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 ...
```

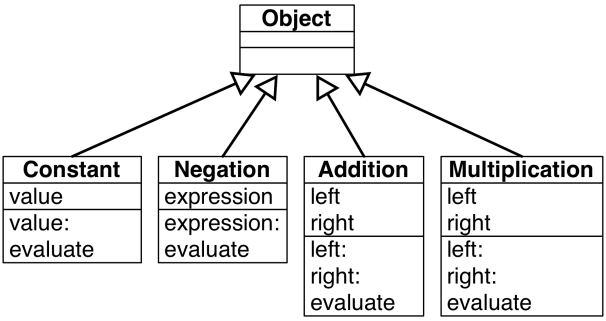

Figure 3-1 A flat collection of classes (with a suspect duplication).

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.

3.3 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 subclass: #EAdditionTest
  instanceVariableNames: ''
  classVariableNames: ''
  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 subclass: #EAddition
  instanceVariableNames: 'left right'
  classVariableNames: ''
  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
```

3.4 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 subclass: #EMultiplicationTest
  instanceVariableNames: ''
  classVariableNames: ''
  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
  instanceVariableNames: 'left right'
  classVariableNames: ''
  package: 'Expressions'
```

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

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

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

3.5 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 3-2. 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.

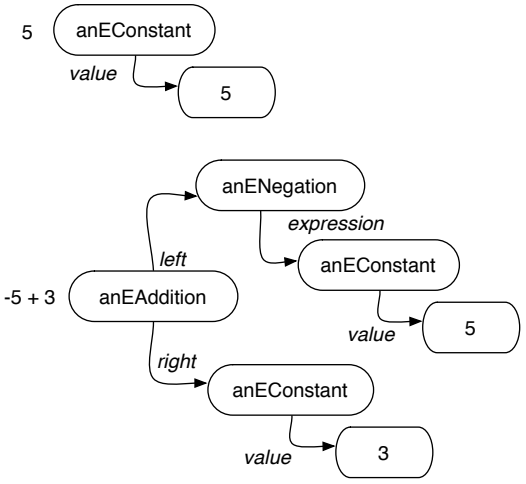

Figure 3-2 Expressions are composed of trees.

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 3-3. 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.

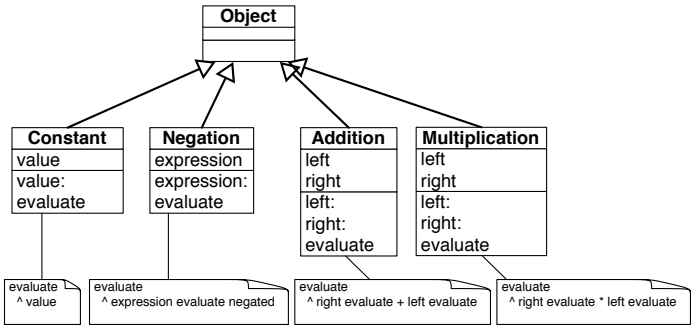

Figure 3-3 Evaluation: one message and multiple method implementations.

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

3.6 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.

3.7 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 3-4). 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.

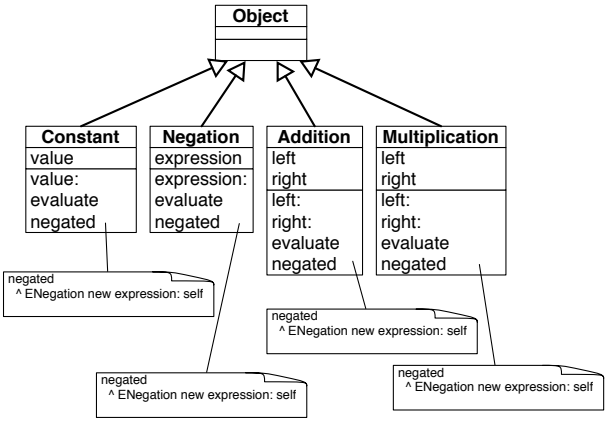

Figure 3-4 Code repetition is a bad smell.

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 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 3-5). This is what we do in the next section.

3.8 Introducing Expression class

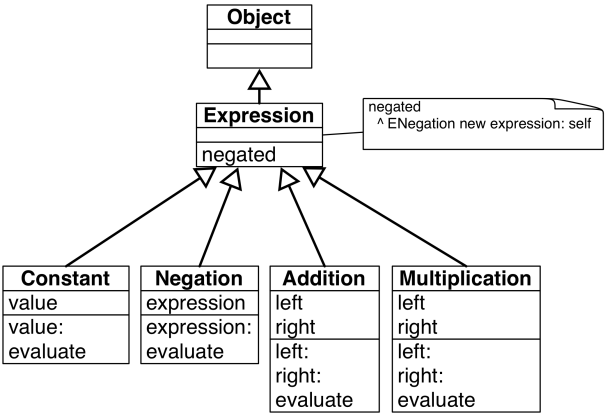

Figure 3-5 Introducing a common superclass.

Let us introduce a new class to obtain the situation depicted by Figure 3-5. We can simply do it by adding a new class:

```
Object subclass: #EExpression
  instanceVariableNames: ''
  classVariableNames: ''
  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 subclass: #EConstant
  instanceVariableNames: 'value'
  classVariableNames: ''
  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 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 really 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 3-5. 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 ArithmeticExpression 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.

3.9 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 constant5.
  ep2 := EConstant constant3.
  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.

3.10 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 with such examples is that

- they help documenting 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 to use them.

3.11 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 customisation 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 which is shared by other objects and it hosts hook methods which are program customisation points. `printOn:` is a hook method. The term hook comes from the fact that code of subclasses are invoked in the hook place (see Figure 3-6).

The default definition of the method `printOn:` as defined on the class `Object` is the following: it grabs the class name and checks if it starts with a vowel or not and write 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
```

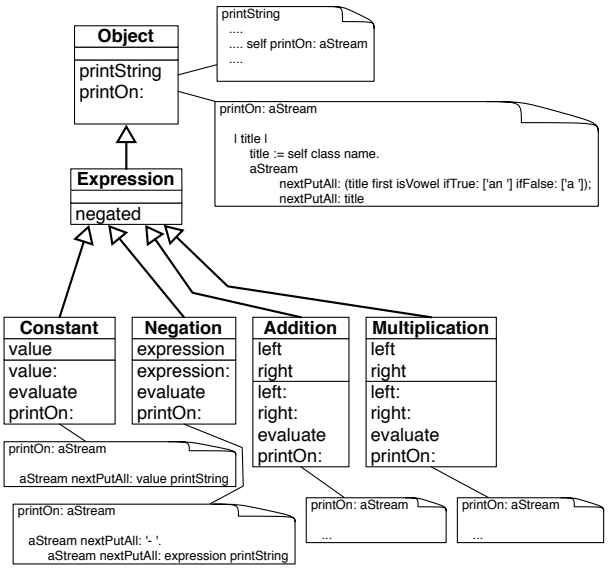

Figure 3-6 printOn: and printString a "hooks and template" in action.

A word about streams

A stream is basically 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 value: 5) printString equals: '5'
```

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 method 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 ...
```

3.12 Revisiting negated message for Negation

Now we can go back on 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 minuses 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 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 3-7. We said that the message negated is overridden in ENegation 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.

3.13 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 subclass: #EBinaryExpression
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions'
```

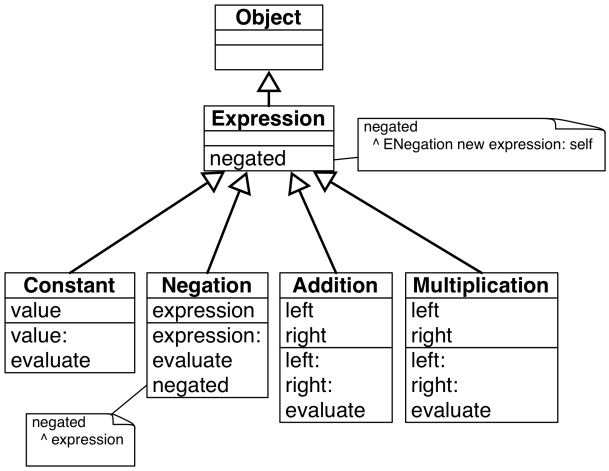

Figure 3-7 The message `negated` is overridden in the class `ENegation`.

```
EBinaryExpression subclass: #EAddition
  instanceVariableNames: 'left right'
  classVariableNames: ''
  package: 'Expressions'
```

```
EBinaryExpression subclass: #EMultiplication
  instanceVariableNames: 'left right'
  classVariableNames: ''
  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 subclass: #EBinaryExpression
  instanceVariableNames: 'left right'
  classVariableNames: ''
  package: 'Expressions'
```

```
EBinaryExpression subclass: #EAddition
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions'
```

```
EBinaryExpression subclass: #EMultiplication
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions'
```

We should get a situation similar to the one of Figure 3-8. All your tests should still pass.

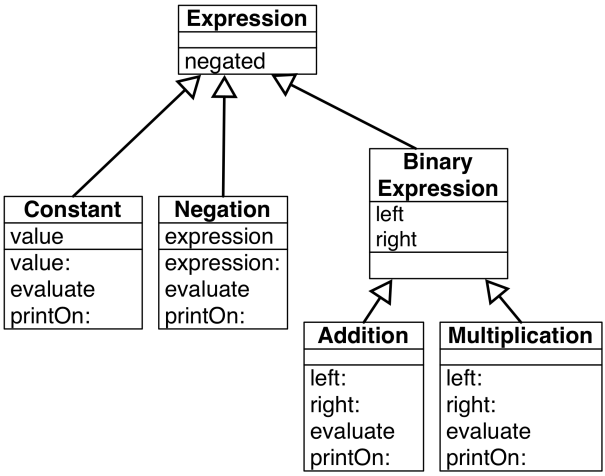

Figure 3-8 Factoring instance variables.

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 to apply the same design point that 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 give 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
  ^ ' * '
```

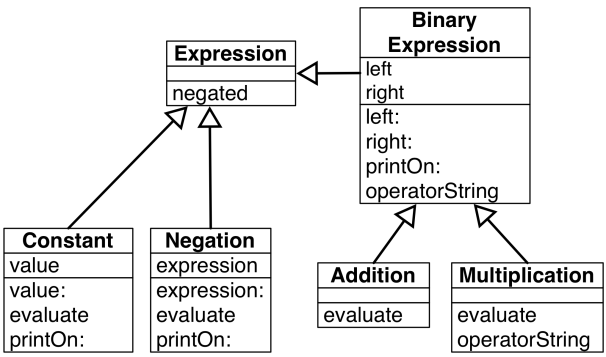

Figure 3-9 Factoring instance variables and behavior.

3.14 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 behav-

ior of programs. As we saw refactorings are powerful operations that really help doing complex operations in a few action.

- We saw that the introduction of a new superclass and moving instance variables or method 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.
- 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.

3.15 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 instance and cause a error because no method `operatorString` is found.

The code would have looked like the following:

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

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

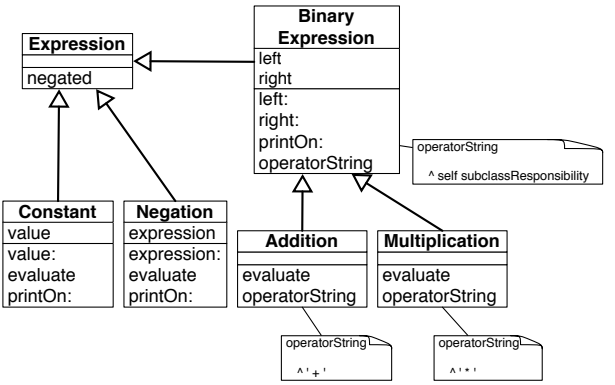

Figure 3-10 Better design: Declaring an abstract method as a way to document a hook method.

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 3-10.

In the solution presented before (section 3.13) 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`.

3.16 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 ma-

nipulate 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 subclass: #EVariable
  instanceVariableNames: 'id'
  classVariableNames: ''
  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 out 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 on 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 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 3-11 shows the final situation.

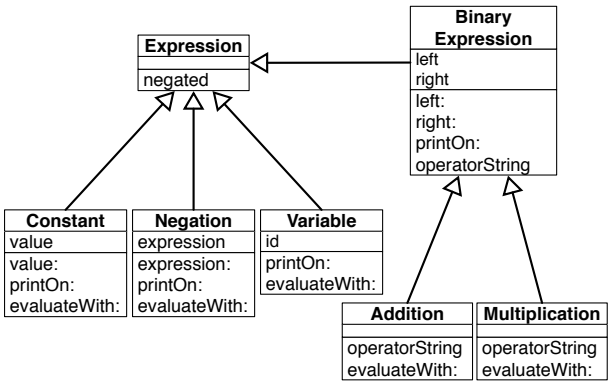

Figure 3-11 Variables and their evaluation.

3.17 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 building confidence in the changes we do by identifying fast errors.
- Refactorings are more than simple code transformations. Usually refactorings pay attention their application does not change the behavior of program. As we saw refactorings are powerful operations that really help doing complex operation in a few action.
- We saw that the introduction of a new superclass and moving instance variables or method 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.

Part II

Power of Messages

CHAPTER 4

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 in 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 think that choices are made using explicit conditionals. In this chapter we will show an example of double dispatch via the paper stone scissors game.

4.1 Starting with a couple of tests

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

```
TestCase subclass: #StonePaperScissorsTest
  instanceVariableNames: ''
  classVariableNames: ''
  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.

4.2 Creating the classes

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

```
Object subclass: #Paper
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'StonePaperScissors'
```

```
Object subclass: #Scissors
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'StonePaperScissors'
```

```
Object subclass: #Stone
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'StonePaperScissors'
```

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

4.3 With messages

We are read to make sure that a first test is passing. Let us work on `testPaperIsWinning`.

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

The first method that we define is `play:` and it takes another player as 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 imaging 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 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 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 read to define the corresponding methods. First we define the methods `playAgainstScissors:` in the correspoding 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
```

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

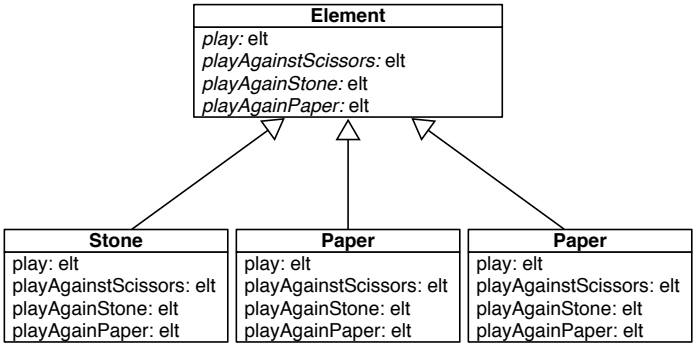

Figure 4-1 An overview of a possible solution using double dispatch.

4.4 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 symetrical. You play a stone against a paper or the inverse. Not all the double dispatch are symetrical. 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 example, 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.

4.5 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 that the object decide 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.

4.6 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 know what do do.

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

Here is a sketche 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
```

4.7 Conclusion

Sending a message is making a choice amongst 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 going 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 5

Stone Paper Scissors Solution

5.1 Stone

```
Stone >> play: anotherTool
  ^ anotherTool playAgainstStone: self
```

```
Paper >> playAgainstStone: aStone
  ^ #paper
```

```
Scissors >> playAgainstStone: aStone
  ^ #stone
```

```
Stone >> playAgainstStone: aStone
  ^ #draw
```

5.2 Scissors

```
Scissors >> playAgainstScissors: aScissors
  ^ #draw
```

```
Paper >> playAgainstScissors: aScissors
  ^ #scissors
```

```
Stone >> playAgainstScissors: aScissors
  ^ #stone
```

```
Scissors >> play: anotherTool
  ^ anotherTool playAgainstScissors: self
```

5.3 Paper

```
Scissors >> playAgainstPaper: aPaper
  ^ #scissors
```

```
Paper >> playAgainstPaper: aPaper
  ^ #draw
```

```
Stone >> playAgainstPaper: aPaper
  ^ #paper
```

```
Paper >> play: anotherTool
  ^ anotherTool playAgainstPaper: self
```

CHAPTER 6

Revisiting the Die DSL: a Case for Double Dispatch

In Chapter 2, 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 is able to 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 clearly too abstract so we will go over a full example to explain it.

6.1 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
```

6.2 [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.

6.3 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
```

6.4 Turning requirements as 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.

6.5 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 really 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.

6.6 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 of 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.

6.7 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` will be executed, we will know that the receiver is a die (instance of this class). Similarly when the method `+` defined on the class `DieHandle` will be 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 makes this really concrete.

6.8 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.

6.9 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 rewritting 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 6-1.

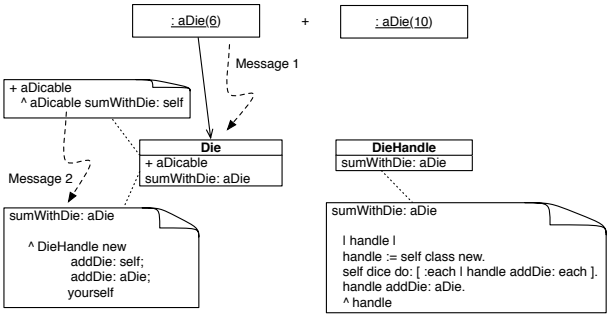

Figure 6-1 Summing two dice and be prepared for more.

6.10 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.

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 are able to sum a die with a die handle as shown in Figure 6-2. The test `testAddingADieAndHandle` should now pass.

6.11 Stepping back

You may ask why this is working. We defined two methods `sumWithDie:` one on class `Die` and one on 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 6-3.

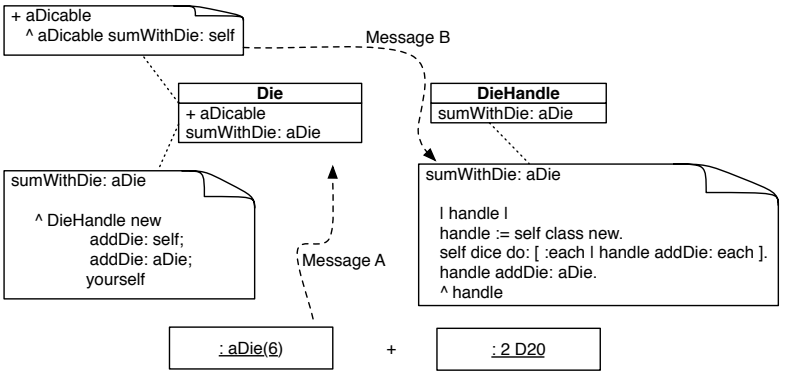

Figure 6-2 Summing a die and a dicable.

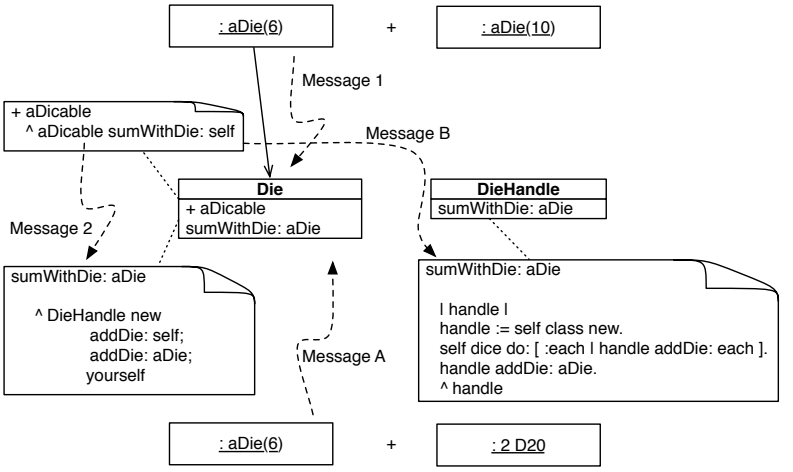

Figure 6-3 Summing a die and a dicable

6.12 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 than 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 is 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.

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

6.13 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 body of `sumWithHandle:` definition.

Figure 6-4 shows the full set up. We suggest to follow 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.

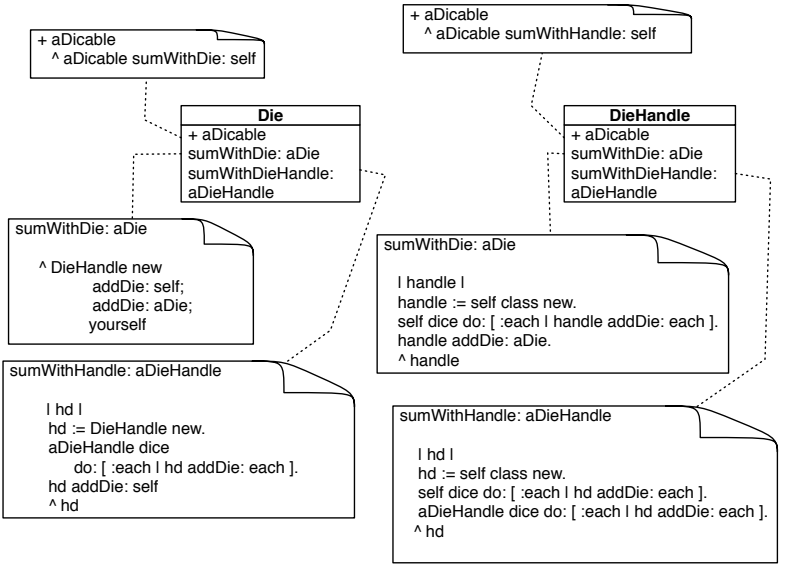

Figure 6-4 Handling all the cases: summing a die/die handle with a die/die handle .

```
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
```

6.14 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 method 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 structure such as documents, 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 7

Revisiting the Die DSL: a Case for Double Dispatch

```
Die >> sumWithDie: aDie

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

```
Die >> + aDicable
  ^ aDicable sumWithDie: self
```

```
DieHandle >> sumWithDie: aDie
  | handle |
  handle := self class new.
  self dice do: [ :each | handle addDie: each ].
  handle addDie: aDie.
  ^ handle
```

```
DieHandle >> sumWithHandle: aDieHandle
  | handle |
  handle := self class new.
  self dice do: [ :each | handle addDie: each ].
  aDieHandle dice do: [ :each | handle addDie: each ].
  ^ handle
```

```
DieHandle >> + aDicable
  ^ aDicable sumWithHandle: self
```

```
Die >> sumWithHandle: aDieHandle
  | handle |
  handle := DieHandle new.
  aDieHandle dice do: [ :each | handle addDie: each ].
  handle addDie: self.
  ^ handle
```

Part III

Playing with Visitors

Here I should take the slides flow from the design mooc lectures.

CHAPTER 8

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.

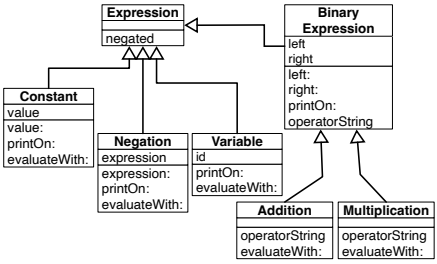

Figure 8-1 A simple hierarchy of expressions.

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

Let us start with the previous situation.

8.1 Existing situation: expression trees

Figure 8-1 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.

8.2 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.
```

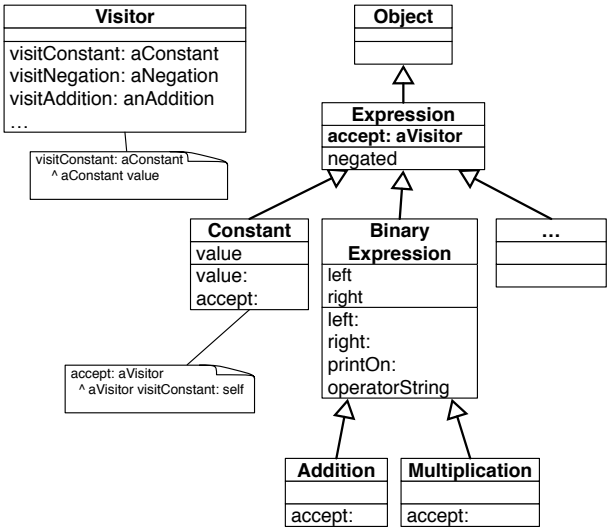

Figure 8-2 Visitor principle.

Let us step by step implement an evaluating visitor.

8.3 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.

8.4 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.

8.5 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
```

8.6 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 8-3 describes the situation.

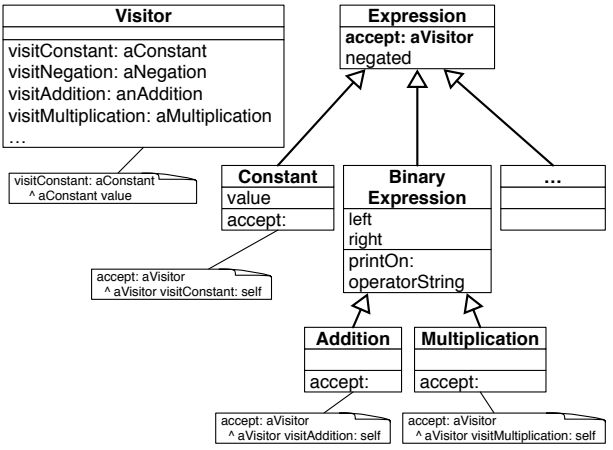

Figure 8-3 Visitor at work.

8.7 Supporting Division

As you can guess the logic is exaclty 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 ...
```

8.8 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
```

8.9 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 ...
```

8.10 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
```

8.11 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'
```

8.12 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 ...
```

8.13 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.

CHAPTER 9

Playing with Interpreters

Returning an integer is not really nice because we cross boundaries. Visitor then after I should make sure that we only return expressions of the same domain and with a stack.