CHAPTER 1

A little expression interpreter

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

1.1 Starting with constant expression and a test

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

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

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

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

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

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

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

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

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

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

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

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

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

Your test should pass.

1.2 Negation

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

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

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

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

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

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

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

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

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

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

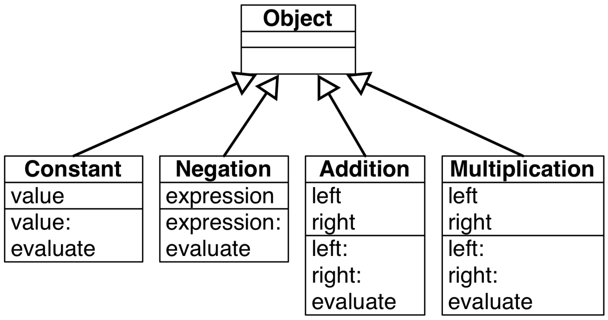

Figure 1-1 A flat collection of classes (with a suspect duplication).

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.

1.3 Adding expression addition

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

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

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

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

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

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

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

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

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

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

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

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

1.4 Multiplication

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

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

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

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

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

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

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

1.5 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 1-2. 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.

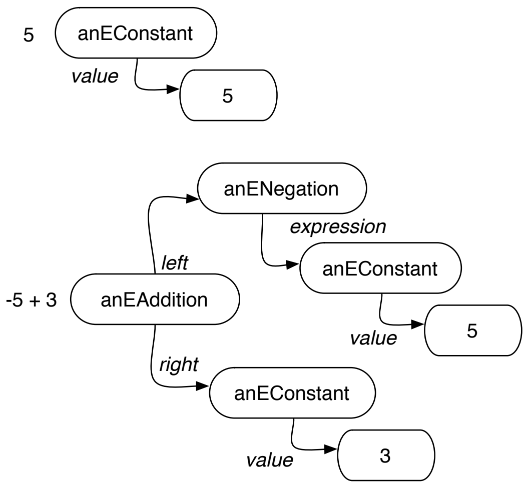

Figure 1-2 Expressions are composed of trees.

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 1-3.

> **Note** 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.

> **Note** 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.

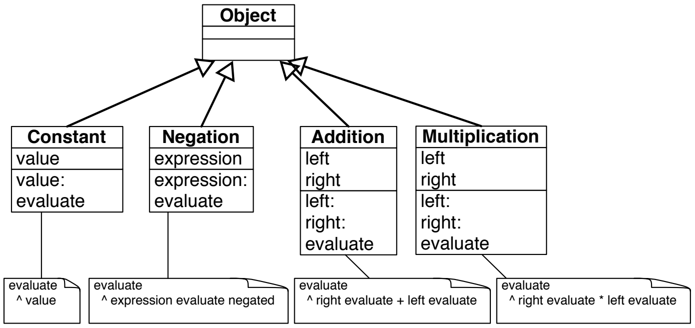

Figure 1-3 Evaluation: one message and multiple method implementations.

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.

Note 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

1.6 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.

1.7 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 1-4). This is not good because if tomorrow we want to change the behavior of negation we will have to change it four times while in fact one time should be enough.

What are the solutions?

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

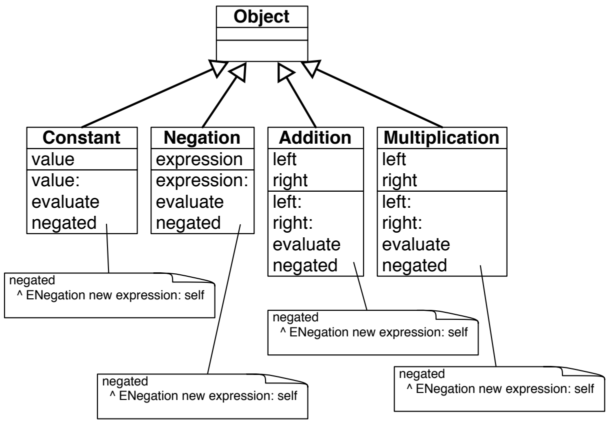

Figure 1-4 Code repetition is a bad smell.

sage `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 1-5). This is what we do in the next section.

1.8 Introducing Expression class

Let us introduce a new class to obtain the situation depicted by Figure 1-5. We can simply do it by adding a new class:

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

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

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

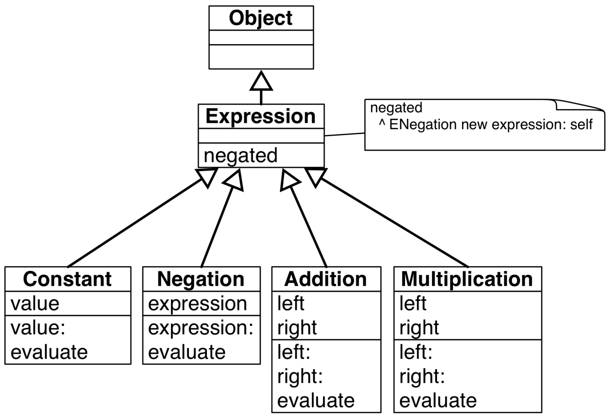

Figure 1-5 Introducing a common superclass.

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 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 1-5. It is a good moment to run all your tests again. They should all pass.

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

1.9 Class creation messages

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

```
EConstant new value: 5
```

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

```
EConstant value: 5
```

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

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

Better instance creation for constants

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

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

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

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

Better instance creation for negations

We do the same for the class `ENegation`.

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

We write of course a new test as follows:

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

Better instance creation for additions

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

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

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

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

Better instance creation for multiplications

We let you do the same for the multiplication.

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

And another test to check that everything is ok.

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

Run your tests! They should all pass.

1.10 Introducing examples as class messages

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

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

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

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

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

This way we can define the test as follows:

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

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

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

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

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

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

What is nice about such examples is that

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

So think about using them.

1.11 Printing

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

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

Note that we do not redefine the method `printString` because it is more complex and `printString` is reused for all the objects in the system. We just have to implement the part that is specific to a given class. In object-oriented design jargon, `printString` is a template method in the sense that it sets up a context that is shared by other objects and it hosts hook methods which are program customization points. `printOn:` is a hook method. The term hook comes from the fact that code of subclasses is invoked in the hook place (see Figure 1-6).

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

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

A word about streams

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

Printing constant

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

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

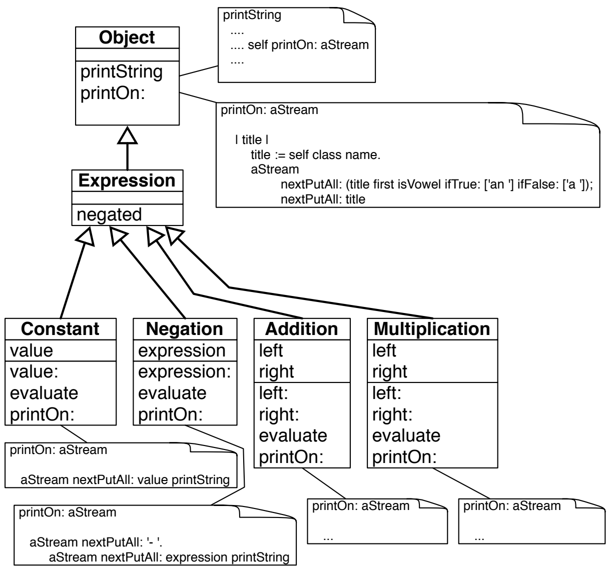

Figure 1-6 printOn: and printString a "hooks and template" in action.

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

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

Printing negation

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

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

Here is a possible definition

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

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

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

We can also define it as follows:

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

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

Printing addition

Now let us write yet another test for addition printing.

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

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

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

Printing multiplication

And now we do the same for multiplication.

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

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

1.12 Revisiting negated message for Negation

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

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

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

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

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

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

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

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

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

Understanding method override

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

By adding a method in the class `ENegation`, we created the situation shown in Figure 1-7. 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.

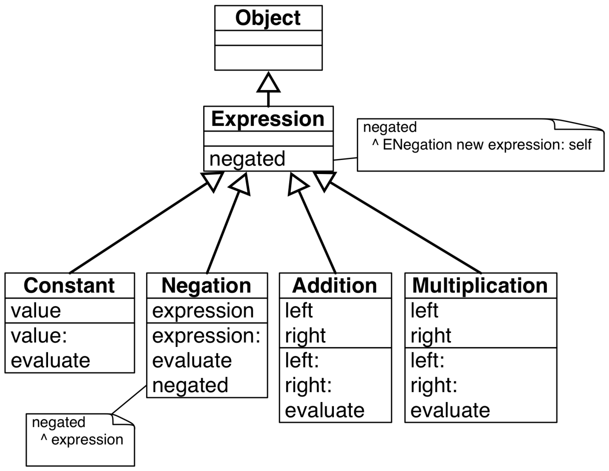

Figure 1-7 The message negated is overridden in the class ENegation.

1.13 Introducing BinaryExpression class

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

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

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

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

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

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

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

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

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

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

We should get a situation similar to the one in Figure 1-8. All your tests should still pass.

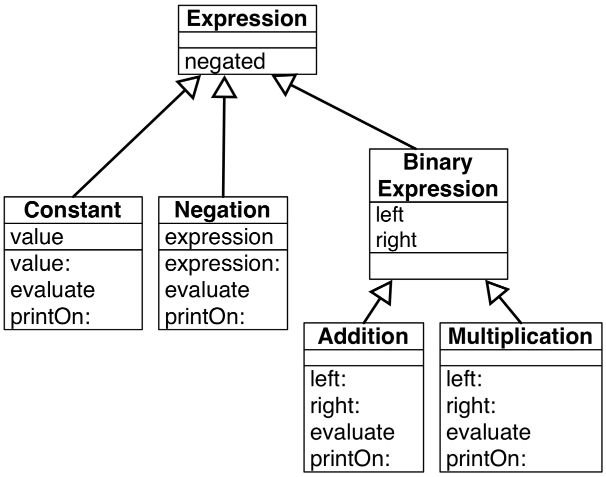

Figure 1-8 Factoring instance variables.

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

Creating a template and hook method

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

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

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

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

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

Here is the result you should get:

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

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

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

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

1.14 What did we learn

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

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

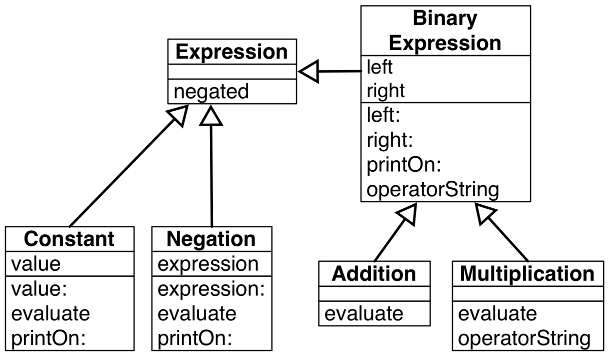

Figure 1-9 Factoring instance variables and behavior.

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

1.15 About hook methods

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

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

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

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

Creating hooks is always good

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

Avoid not documenting hooks

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

The code would have looked like the following:

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

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

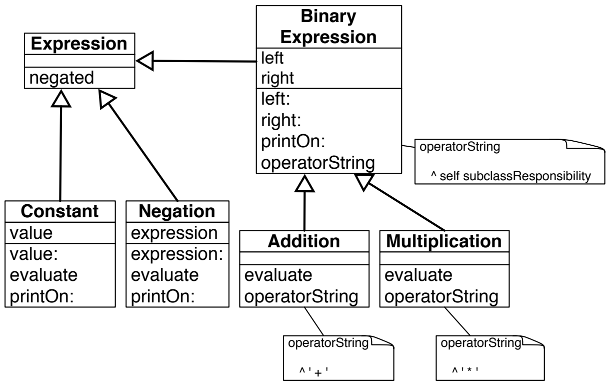

Figure 1-10 Better design: Declaring an abstract method as a way to document a hook method.

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 1-10.

In the solution presented before (section 1.13) 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`.

1.16 Variables

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

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

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

Some technical points

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

Dictionaries

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

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

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

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

Dynamic Arrays

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

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

Pairs

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

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

Back to variable expressions

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

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

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

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

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

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

Your tests should all pass at this point.

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

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

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

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

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

Non working approaches

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

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

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

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

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

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

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

The solution: adding evaluateWith:

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

So here are the steps that we should follow.

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

A test like the following one:

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

is transformed as follows:

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

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

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

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

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

Do the same for the multiplications.

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

Figure 1-11 shows the final situation.

1.17 Conclusion

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

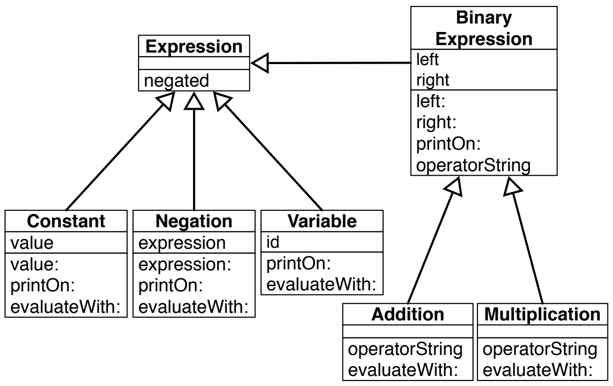

Figure 1-11 Variables and their evaluation.

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