1

Errors and Debugging

Now that you know how to define methods calling other methods, it is time to have a look at a really powerful tool that helps you to find your errors: the Squeak debugger. A debugger is a tool that presents the execution of a program and lets you inspect and changes the values of the variables or the methods of a program. Therefore in this chapter I will present some possible errors and what is a debugger. I start to present some errors you may get with variables then I show you how to use the debugger to identify and fix problems.

1 Default Value of a Variable

Using variables offers a lot of power, but they require a bit of attention. Indeed you can get errors by using variables not declared or with wrong values. Even experienced programmers make errors. The only difference is that an experienced programmer knows how to find his errors. Squeak provides some help by checking whether the variables have been declared. In addition, using a debugger can help to understand your errors.

First let's experiment a bit. Type, select and print the following script (1.1). You should obtain Figure 1.1 that shows that Squeak prompts you when a variable is not initialized. If you selected the choice yes when prompted when you print the script 1.1, you should get nil printed as shown by Figure 1.1 (right). The value nil that you just obtained is a special object assigned to any declared variable. First let us start to see what is the default value of a variable.

Script 1.1

```
| size |
size
```

When you do not initialize variables, the program has a higher chance of failing because it may happen that you use a variable having wrong or no value. By default, in Smalltalk the value of a

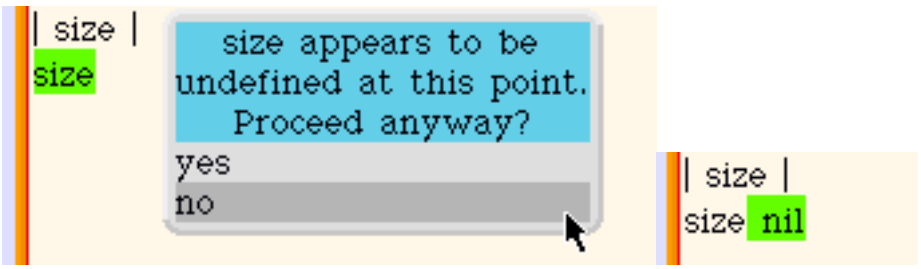


Figure 1.1: Left: Squeak prompts you when you use a variable that does not have been initialized. Right: the value nil is printed when you proceed.

uninitialized variable is nil. nil is an object that represents undefined values, that is objects with which we cannot do too much. Although it is an object like any other, nil does not understand many messages. In particular, it does not understand any of the number or robot messages. Therefore you will get an error when you send such messages to the object nil. Still the object nil is important because it allows you to know if a variable has been assigned or not.

Important!

By default, in Smalltalk the value of a uninitialized variable is nil. nil is an object that represents undefined values, that is objects with which we cannot do too much.

The script 1.2 shows that Squeak analyzes the script and detects before running it that I never assigned a value to the variable size.

Script 1.2

```
| size |
size + 10
```

The error occurring during the execution of the script above generates the message shown in Figure 1.1. Normally just answer no and initialize the variable. However Squeak cannot detect if you used a wrong value. In such a case it opens a debugger with which you can access the execution state (the receiver of the message, the message, the variables available...). This is what I will explain now.

2 Looking at Message Execution

I remind you the definition of the methods pattern and pattern4 if you forgot to save it, as I will use them to explain how the debugger works. What is important to remember is that the method pattern4 is using the method pattern and that the method pattern in its turn is invoking methods go: and turnRight: as show below.

Method 1.1

```
pattern
  "draws a pattern"

  self go: 100.
  self turnRight: 90.
  self go: 100.
  self turnRight: 90.
  self go: 50.
  self turnRight: 90.
  self go: 50.
  self turnRight: 90.
  self go: 100.
  self turnRight: 90.
  self go: 25.
  self turnRight: 90.
  self go: 25.
  self turnRight: 90.
  self go: 50
```

Method 1.2

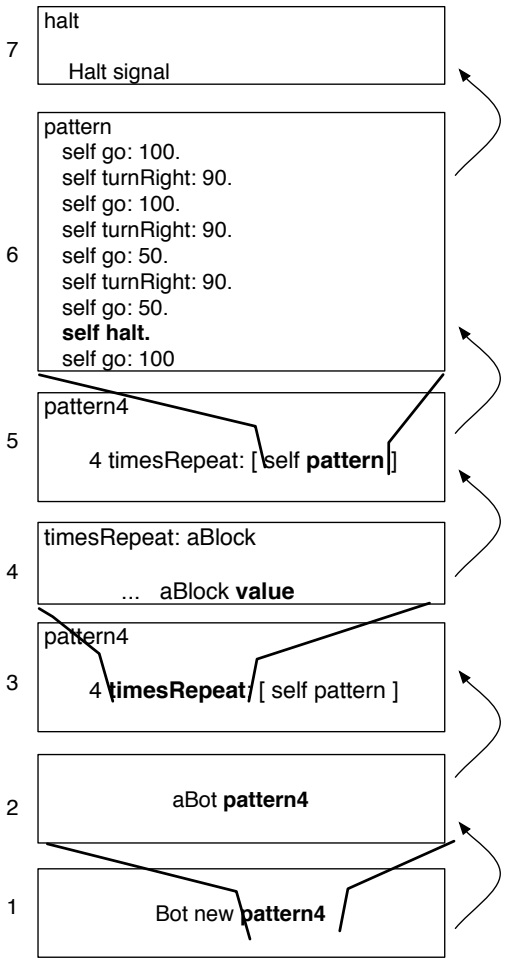
```
pattern4
  "draws 4 patterns"

  4 timesRepeat: [self pattern]
```

When Squeak executes the expression Bot new pattern4, several messages get invoked in a reaction chain that will makes the robot drawing the pattern on the screen. Let's have a look at this chain of messages: first Bot new creates a new robot to which the message pattern4 is sent. As a result the method pattern4 is executed. This execution send the message timesRepeat:, which in its turn send the message pattern. The execution of the method pattern sends several messages go: and turnRight:. In programming language jargon such a chain of messages is called an execution stack: it contains all the method executed in reaction to an initial message and the chain of messages that followed it.

A possible representation is shown in Figure **??** where the most recently executed methods are on the top, a method calling another one is below the called method, and I put in bold the part of the expression that led to the invocation of the method above it. Let's see how this is working:

1. In the bottom box, the message new is sent to the class Bot, this creates a new robot.
2. The message pattern4 is sent to the created robot.
3. The execution of the method pattern4 sends the message timesRepeat: [self pattern]. The message timesRepeat: is in bold as it is first sent.
4. The execution of the method timesRepeat: leads somehow to the block execution. This is done by sending the message value to the argument of the message timesRepeat:.
5. In the method pattern4 as result of the block execution by the method timesRepeat:, the message pattern is sent.
6. Similarly, the process continues, that is the messages of the method pattern are executed one after the other.

Figure **??** contains one other box that I will explain in a minute. What you should see is that one method call another one and that the called method is above the calling one.



@@Production: I want a number for the figure and the following caption: The execution stack containing all the messages and the methods invoked resulting of the execution of Bot new pattern4.@@

3 A First Look at the Debugger

Right now you have imagined the sequence of messages sent and methods executed resulting from the execution of a message, but Squeak has a debugger, a powerful tool that lets you see, navigate and change the chain of messages. The debugger is automatically invoked when a message is not understood by an object as I will show later, but you can invoke it explicitly by introducing the expression self halt in the body of a method. Introducing such an expression is useful to understand or find a bug in a program.

> A debugger is a tool that allow you to navigate through the executed methods. Using a debugger, you can print the values of the argument, modify the method definition and continue to execution.

To open a debugger, add the expression self halt in the method pattern as shown by method 1.3. Then execute the expression Bot new pattern4, you should obtain the situation depicted by Fig-

ure 1.2. A new robot is created, it started to draw the first pattern and got stopped and you get a chance to open a debugger.

Method 1.3

```
pattern
  "draws a pattern"

  self go: 100.
  self turnRight: 90.
  self go: 100.
  self turnRight: 90.
  self go: 50.
  self turnRight: 90.
  self go: 50.
  self halt.
  self turnRight: 90.
  self go: 100.
  self turnRight: 90.
  self go: 25.
  self turnRight: 90.
  self go: 25.
  self turnRight: 90.
  self go: 50
```

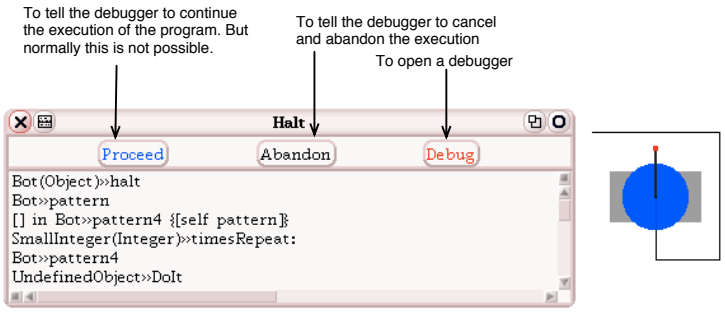


Figure 1.2: The robot started to draw the beginning of the first pattern then stopped and you get a chance to open a debugger.

> To raise a debugger insert the expression self halt in a method. The execution of the expression self halt raises a debugger.

The window presented in Figure 1.2 proposes three buttons: **Proceed**, **Abandon** and **Debug**.

- **Proceed.** This button tells the debugger to continue the execution of the message as if the self halt was not in the method. Note that this functionality is possible only if you provoked the error using self halt. When it is a real error that raises the debugger, using proceed does not help since Squeak cannot continue.

- **Abandon.** This button tells the debugger to simply close and it discards the actual execution.
- **Debug.** This button tells the debugger to open as shown by Figure 1.3

If you press the **debug** and select the second line of the top pane, you obtain Figure 1.3. The debugger is composed of several panes. The top pane represents the execution stack that I presented in the first section: you get all the messages that have been sent just before the halt occurs, the most recent message on top. The most recent message is halt therefore it is at the top of the stack and this is the the one that lead to the opening of the dialog box shown in Figure 1.3.

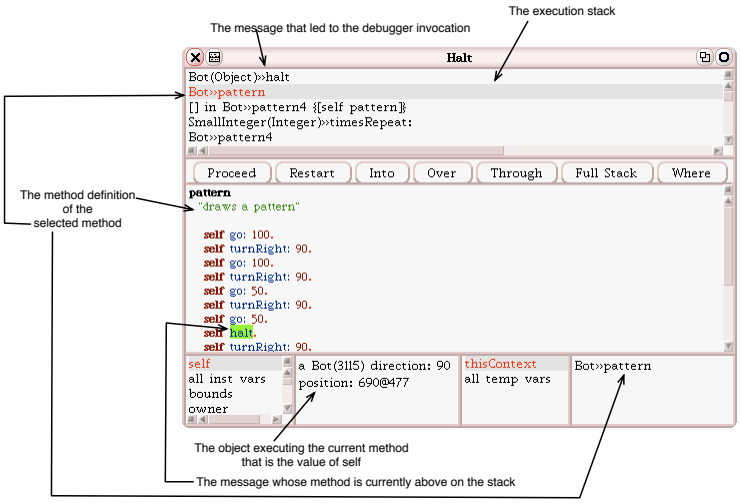


Figure 1.3: Selecting the method pattern.

Selecting one of the lines in the top pane shows the method definition in the second pane. Figure 1.3 shows that we selected the method pattern in the stack, and its definition is shown in the second large pane. The object that received the message (that is the object referred to by the variable self) and that executed the method pattern is shown in the left bottom part.

In the method body of the currently selected method, here pattern, the debugger highlights in green the method whose execution is currently stopped and that is above the selected method on the stack. Here self halt is currently stopped and then above pattern. This also means that the expression above the self halt in the method body have been executed, while the one below haven't. Here self go: 100. self turnRight: 90. self go: 100. self turnRight: 90. have already been executed.

If you click on the third line, that is the method pattern4 you see that the stack of method execution above is related to the execution of the expression self pattern of the method pattern4 (in Figure 1.4). Now if you click on the fifth line, you have a look again at the method pattern4 but before the loop is executed. The debugger indicates that the method above is related to the execution of the complete loop, timesRepeat: [self pattern].

If you select the fourth line as shown in Figure 1.4 you get the definition of the method timesRepeat: itself. Again the debugger shows the methods that are executed and lead to the method above in the stack. In addition notice that the object receiving the message is different, here this is not a robot but an integer as in the expression 4 timesRepeat: [self pattern] the receiver of the loop is the integer 4. This means that within this method the variable self is bound to an integer. In fact this is normal as the variable self *always* represents the object that received the current message.

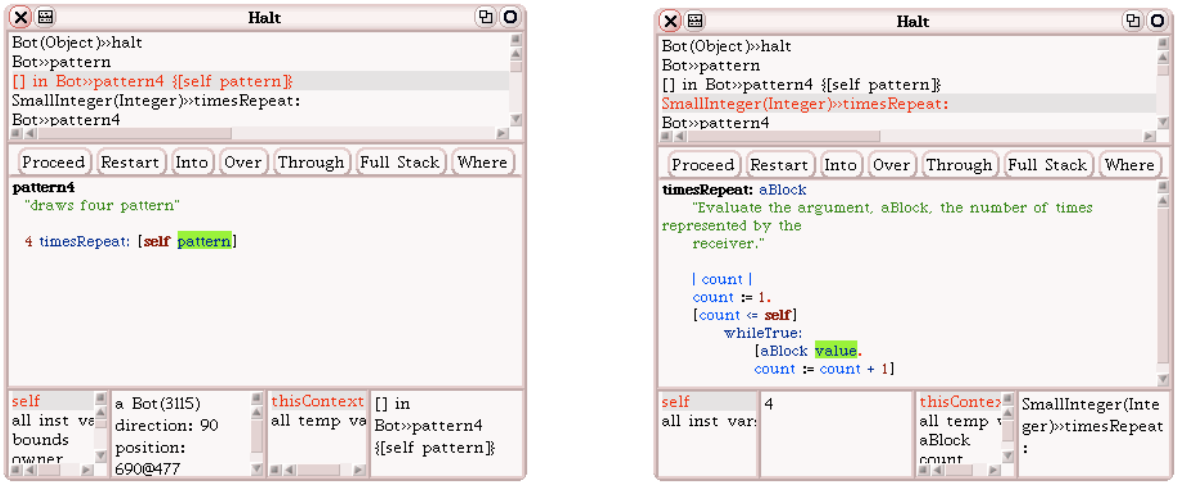


Figure 1.4: Left: Selecting the method pattern4. Right: Selecting the method timesRepeat:

4 Stepping in the Stack

The Squeak debugger not only lets you navigate the stack and identify the receiver of the messages, but it lets you execute step by step the method. You can ask the debugger to perform several actions using the buttons between the first and second pane. I describe the most useful ones from left to right.

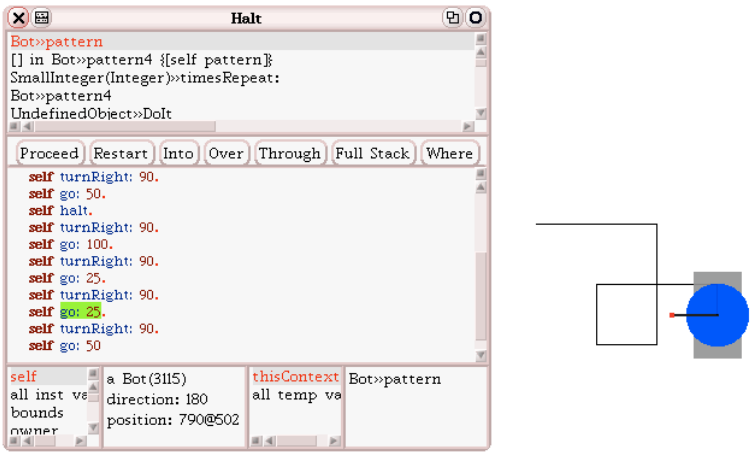


Figure 1.5: Stepping in the method pattern4 and seeing the robot executing all the stepped methods one by one.

- **Proceed.** Pressing the button proceed as the same effect as pressing the button proceed in the dialog box shown in Figure 1.2: it closes the debugger and continues the method execution if possible.
- **Restart.** You can ask the debugger to restart the execution of the current method. Note that sometimes this does not make sense since you may modify twice the same object and this may be incompatible.
- **Into.** Lets you go *into* the method currently selected without executing (See Figure 1.6).

- **Over.** This functionality is with the previous one the most used and useful of the debugger. It lets you execute the message currently selected but without going into the method execution. You simply execute the following expression and stop after. (See Figure 1.5). If you step over some expressions in the method pattern you should see the robot performing each action one after the other. Note that when you arrive at the end of the method, stepping over just return to the method that invokes the current one: you go down the stack. If you were stepping in pattern then you would end up in pattern4.

When you press the button **Into**, you ask the debugger to go into the method without executing it. For example, if the currently selected expression in the method pattern is self turnRight: 90, then pressing the button **Into** asks the debugger to go inside the method turnRight:, stopping at its first expression as shown by Figure 1.6. Here the first message to be sent is the message negated. Again here you have the choice to go into the first expression or to execute it. Again when you reach the end of a method, you come back to its calling method. Note that you can also see the value of the arguments passed to the method by selecting the argument name on the method body and using print it (see Figure 1.7).

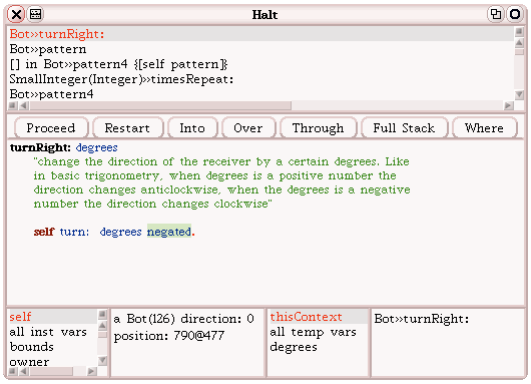


Figure 1.6: Stepping into the method turnRight:

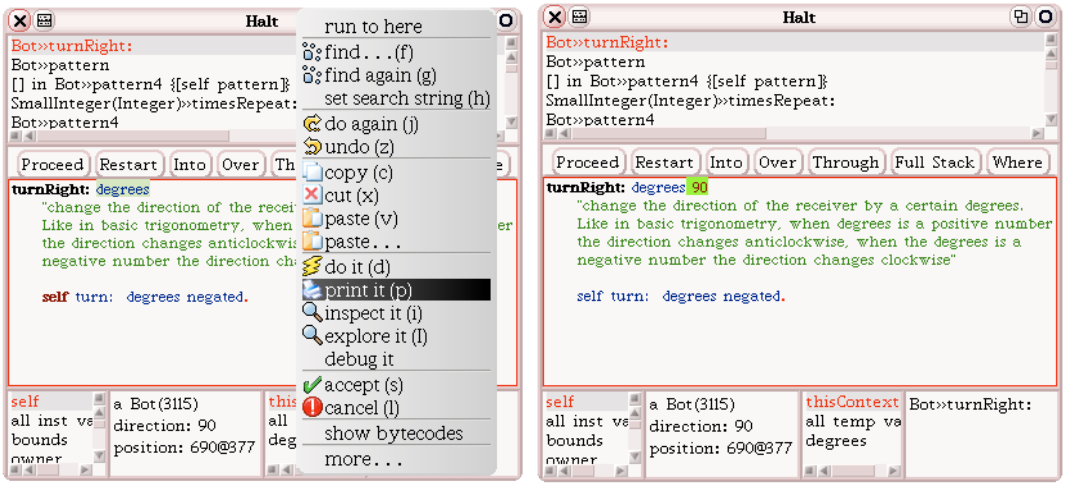


Figure 1.7: Right: Selecting an argument and printing. Left: Value printed.

In addition you can *modify* the method definition inside the debugger by editing the code in place and accepting it via the contextual menu item **accept** (see Figure 1.8). For example in Figure 1.6, I restarted the method by pressing the **Restart** button then edited the method in the debugger itself and replaced the 100 of the first go: message by 500. Then I recompiled the method by using the accept item choice of the menu. Now the robot will move 500 pixels if you press the **proceed** button.

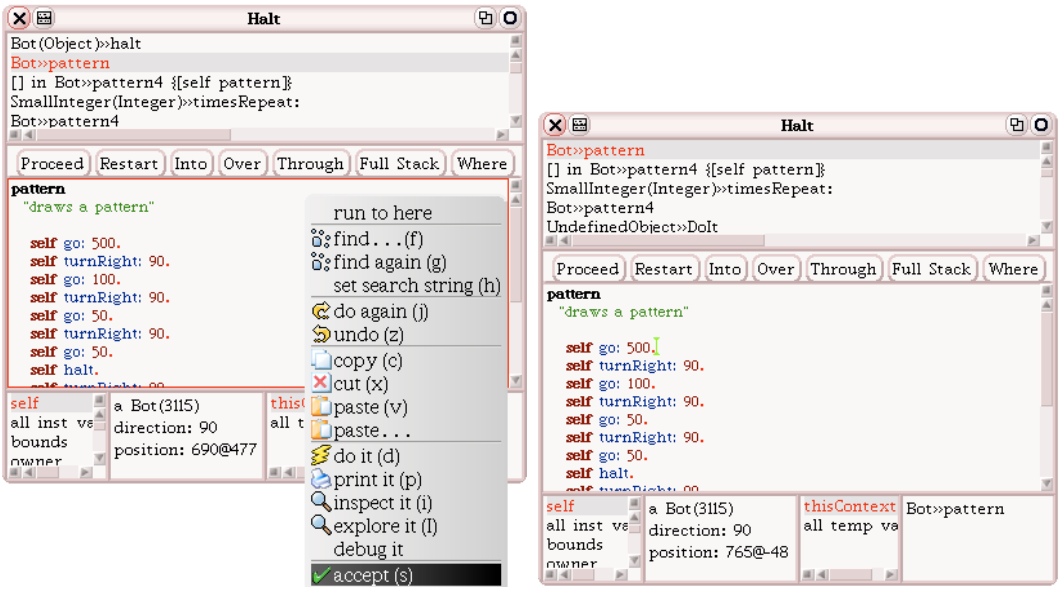


Figure 1.8: Right: Editing the method definition and recompiling it. Left: Method pattern recompiled

5 Fixing Errors

I show you that you can get a debugger by inserting the expression self halt in the method. However, you can also use the debugger to fix your errors. When an object receives a message that it does not understand you can get a debugger. In fact when an object does not understand a message, Squeak sends to this object the message doesNotUnderstand: with a representation of the message. By default, the method doesNotUnderstand: open a dialog box to ask you whether you want to get the debugger. You can use this debugger to navigate in the stack of executed methods and try to understand what went wrong.

Example 1. To illustrate the process, change the first line of the method pattern to be self go2: 100 and execute the expression Bot new pattern4. You should get the debugger dialog box as shown in Figure 1.9. The debugger dialog box indicates that the receiver, a robot created by the class Bot, does not understand the message go2:. When you press the button **Debug** of the dialog you get the debugger shown on the right of Figure 1.9.

The method on the top of the stack is the method doesNotUnderstand: which is sent to the receiver of a message, when the receiver does not understand a message. The method directly below it is then the method containing the message leading to the error and the call of the method doesNotUnderstand:. Here the method pattern contains the message go2: which is not understood by the robot as shown in Figure 1.9

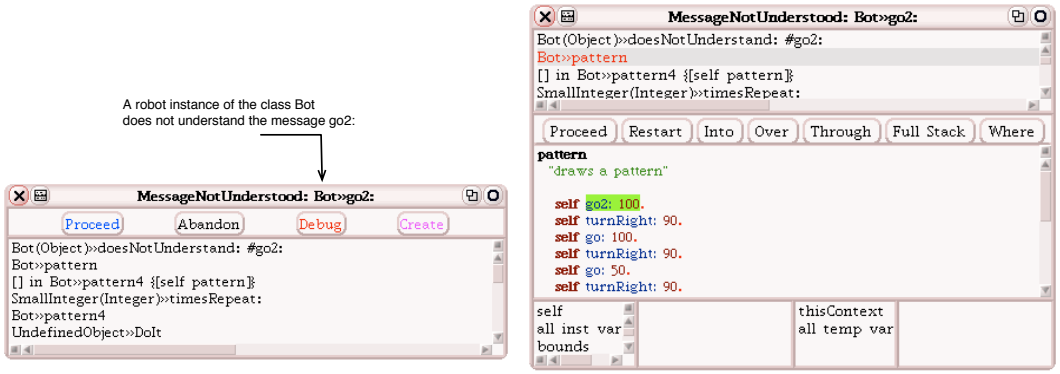


Figure 1.9: Left: Getting a doesNotUnderstand: error. Right: Identifying the problem.

Example 2. Now change the first line of the method pattern to be self go: nil and execute the expression Bot new pattern4. You should get the debugger dialog box as shown in Figure 1.10. The error is more difficult to spot. Here, the dialog box title MessageNotUnderstood: UndefinedObject indicates that there is a message not understood sent to nil, which is an instance of the class UndefinedObject.

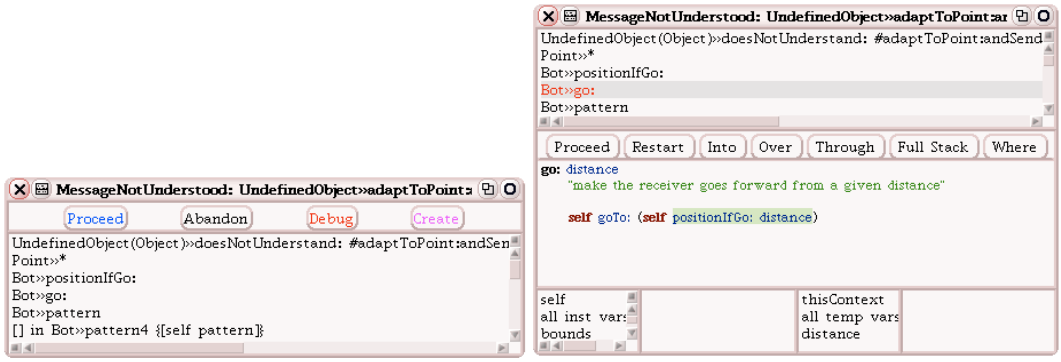


Figure 1.10: Left: A message not understood sent to nil. Right: Going down the execution stack.

The fact that nil was passed as value led to an error after several other method executions. Therefore you have to go down the stack to the point where you can understand your mistake and fix it. For example, in Figure 1.10 right, the second method from the top shows that something wrong happened with *, but the problem does not come from this method. This is the same situation in the following method. Select the method go:, you can see that the method go: just passes the argument it gets from the method pattern to the method positionIfGo:. If you select the parameter distance and print its value you get nil. This indicates that the problem comes from a place further down the stack.

Finally as shown in Figure 1.11 left, you see that nil is passed as argument instead of a number as expected by the method go:. You can now fix the bug by editing the method go: and replacing nil by the value, and press the button **Proceed** to let the execution runs.

Note that in this example, I use nil to get a really simple problem to identify. While you can get this kind of bugs, you will have to face all kind of unexpected situations leading to a bug. However, the process is the same, you will use the debugger, navigate through the stack, check the values of the arguments until you understand what went wrong. You may be forced to change your code and not only the arguments of messages sent.

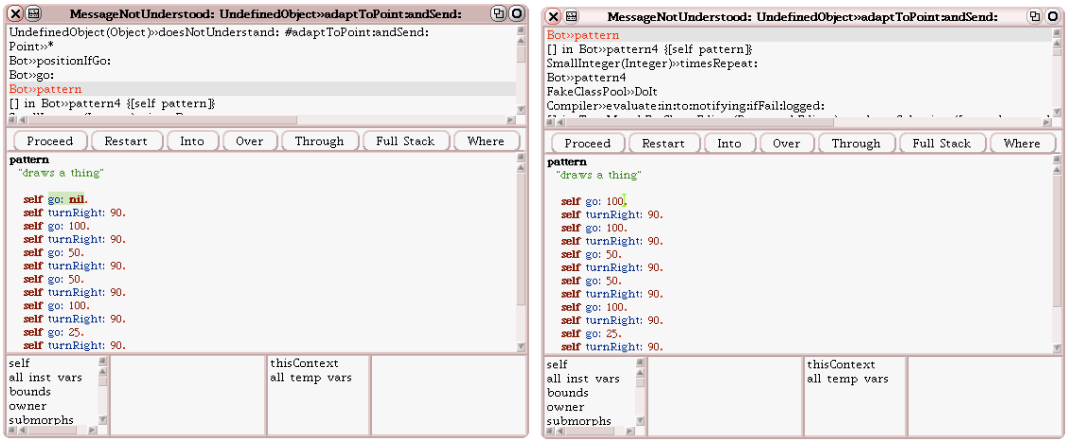


Figure 1.11: Left: Editing the method definition and recompiling it. Right: Method pattern recompiled.

Summary

- By default, in Smalltalk the value of a uninitialized variable is nil. nil is an object that represents undefined values, that is objects with which we cannot do too much, hence leading to error.
- A debugger is a tool that allow you to navigate through the executed methods. Using a debugger, you can print the values of the argument, modify the method definition and continue to execution.
- To raise a debugger insert the expression self halt in a method. The execution of the expression self halt raises a debugger.