Designing Algorithms

2

This chapter attempts to explain how one finds an algorithm for a particular problem. Unfortunately, there is no simple answer to this question, and it's utterly impossible to provide a "cookbook" that would allow one to advance with confidence from one problem to the next. Even today software engineering does not have the kind of well-defined design rules that are a given in more mature engineering sciences. Instead, there are various codes of procedure that support designers in their work and are meant to increase the probability that the resulting programs will fulfil particular quality criteria, such as *correctness* and *readability*. Section 2.4 provides more information.

One can divide the design of algorithms into two partial sets of tasks. The first step consists of finding a method to solve a particular expression of the problem. This solution method—the actual algorithm—can initially be formulated without regard to what language the machine that will eventually be employed to automatically solve the problem understands. In the next step, though, it is necessary to create a machine-readable version of the algorithm in the form of a program written in a programming language that the selected machine can understand. At first, we'll look at two simple examples in which the first step poses no difficulty. The first example attempts to solve the problem of converting a certain amount of money from one currency into another. In this case, the solution method is obvious. The second example considers the problem of solving quadratic equations, the solution to which is already provided by standard mathematics. In both cases, therefore, we can concentrate on the second step, namely how to formulate the solution in a machine-readable format.

2.1 Case Study: Currency Conversion

Before the introduction of the euro, tourists often travelled across Europe carrying little cards that showed a table for converting amounts from one currency to another. For example, one could convert—with a little effort—amounts from German marks into Austrian

J. Brauer, *Programming Smalltalk – Object-Orientation from the Beginning*,
DOI 10.1007/978-3-658-06823-3_2

schillings; naturally the exchange rate was not up-to-date. Nowadays when exchange rates can be found online wherever one happens to be, one is likely to reach for one's cell phone.

At this point, we will solve the specific problem of converting a certain amount in Swedish kronor into euros. Assume that the exchange rate is 1 Swedish krona (*sk*) equals 0.108 euros (*eu*).[1] In order to calculate the amount in euros, one can use the following simple formula:

F: euAmount = *skAmount* · 0.108

At this point, we need to know what the capabilities of our machine actually are or, more precisely, what expressions in a programming language can make these capabilities available. The only calculation operation we need, multiplication, is naturally part of our machine's repertoire.

One of our machine's peculiarities, though, is that it can only process programs written in the form of a text that contains nothing but characters found on a standard typewriter (or PC) keyboard. Standard keyboards, though, don't contain a multiplication symbol (·); instead we have to use an asterisk (*). Also, we have to be sure that a period is used as the decimal separator, and not another character as is sometimes the case in other countries. Our formula now looks like this:

F: *euAmount* = *skAmount* * 0.108

The "design" of the algorithm for the currency conversion is now complete. It is universally applicable, because it can be used to convert any amount of Swedish kronor into euros. To perform a specific calculation, one needs only to substitute a specific amount for the variable skAmount.

The follow section shows how this algorithm can be executed on a real machine.

2.2 The First Smalltalk Program

The machine that we are going to use to execute algorithms will initially be called *SmaViM*, which is an acronym for Smalltalk Virtual Machine. Why did we select this name?

In Chap. 1, it was pointed out that computers are only capable of interacting with binary strings of symbols. That means that programs themselves must be coded in a binary language. The binary code that a computer can directly interpret as an instruction to do something is called a machine language. Because binary codes are very difficult for humans to read, directly programming in a machine language is not merely extraordinarily cumbersome and prone to error, it also manifests the problem that every type of computer uses its own machine language. If one wanted to use a program that had originally been developed for one kind of computer on a different kind, it would first be necessary to translate the code. For that reason, as early as the 1950s a trend started to develop so-called *higher*

[1] Exchange rate for May 12, 2008.

or *problem-oriented* programming languages. Well-known examples of such languages include, besides ALGOL, which we already discussed, FORTRAN, COBOL, PASCAL, C and, of course, Java and Smalltalk. These languages were called problem-oriented because the programmer was no longer forced to write instructions in the language of a specific machine, but could instead write problem solutions in a more familiar language, perhaps using mathematical notation. The example of our Formula F shows this clearly.

Translating Programs

In order for programs written in a higher language to still be able to run on a specific computer, they must first be translated into that computer's machine language. This translation procedure can be automated; programs that perform this function are called *compilers*. (The German concept, *Übersetzer*, or translators, is not used very much.) A compiler must in turn exist in the specific machine language so that the machine can run it. One needs a specific compiler for each programming language and each type of machine.

Smalltalk and Java programs operate somewhat differently. In order to avoid having specific computers translate programs that were developed right from the start on different machines, these programs initially define a universal, machine-independent instruction set.[2] After that sentence occurs, programs are translated only into this quasi-machine language, regardless of which machine they will eventually run on. For each specific machine type, though, one needs a program that can interpret the commands in the quasi-machine language and create for each command a specific binary sequence for the individual machine. This program allows the computer to *seem* to understand the universal instruction set, and for that reason, these machines are called *virtual machines* (VMs). Because of the procedure that was just described, though, they are sometimes also called *interpreters*. SmaViM is thus the VM that can execute Smalltalk programs once they have been translated into byte code.

A Smalltalk system has always consisted of not just a programming language and a VM, but also of an integrated development environment. The development environment has a modern GUI, which allows programmers to write Smalltalk programs, as well as to translate and execute them, and also to perform debugging tasks. These Smalltalk development environments are available from several sources; they differ in the type and extent of their components as well as in how they operate. For our purposes, it is sufficient to use the basic functions, which are more or less the same in all development environments. The screen captures used in the following to illustrate how Smalltalk programming works were made using Cincom's *VisualWorks*,[3] which is available free-of-charge for instructional purposes.

Chapter 5 describes how to use VisualWorks. Section 5.3.2 explains the required basic settings to enable the student's copy of VisualWorks to resemble the screen captures shown in this book.

[2] Frequently called a byte code.

[3] Version 8.0.

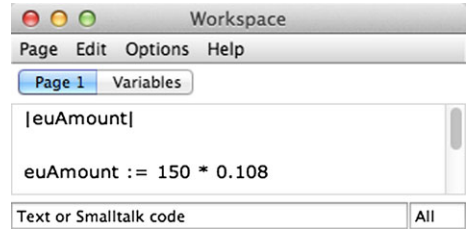


Fig. 2.1 Workspace (Mac OS version)

2.2.1 Entering Program Text

Every Smalltalk development environment makes a so-called *workspace* available. The workspace is a window that allows users to key in and edit Smalltalk programming text; it can be used like a window in a conventional text editor. Figure 2.1 shows an example of a workspace. The image represents what one sees when one runs VisualWorks on a Mac OS X platform.

For the sake of comparison, Fig. 2.2 shows the same workspace window under Microsoft Windows 7 OS. In addition, the tools and status-bar options are shown, which can be selected from the **Options** menu. The Mac OS variant will be shown from here on.

The text in the window shows the Smalltalk program for the currency conversion based on the algorithm described in the last section. In contrast to the text in the book, though, the image shows that the variable skAmount has been replaced with the numeral 150. In other words, the program is supposed to determine how many euros 150 Swedish kronor are equal to. In addition, there are a few other small changes, which are due to Smalltalk's syntax.

1. The first line contains, between two vertical lines, the name of the variable used in the program (`euAmount`). This is a so-called *declaration* of the variables. In general, you must declare variables before you can use them. Between the vertical lines, you can declare as many variables as you like, separated by spaces, tab spaces or carriage

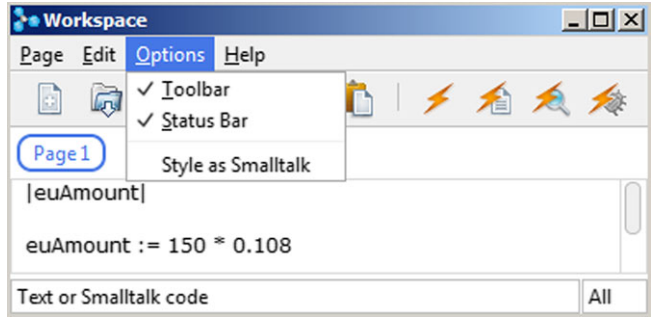


Fig. 2.2 Workspace (Windows version)

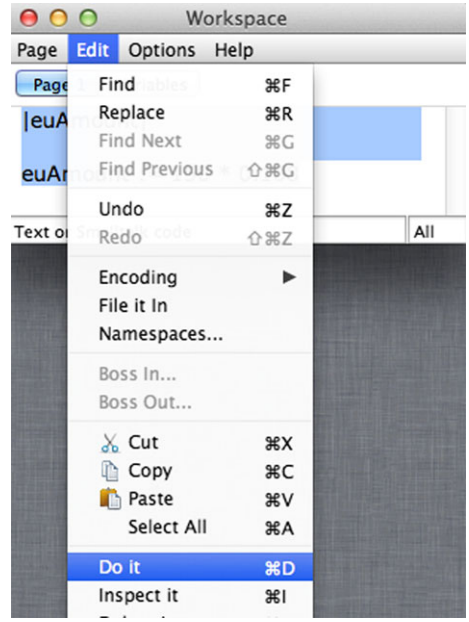


Fig. 2.3 The Edit menu in the workspace

returns. More specifically, the variables described here are *temporary* or *local* variables. We will get to know other kinds of variables further on in this manual.

Note for people who know conventional higher programming languages like PASCAL: Smalltalk variables do not have a type. Any random value can therefore be assigned to a variable.

2. In place of "`euAmount = ...`" we write here "`euAmount := ...`" The character string "`:=`" expresses the so-called *assignment*. The value of the expression on the right of the equal sign is allocated (assigned) to the variable shown on the left. The equal sign (without the preceding colon) serves in Smalltalk to compare expressions (see Sect. 2.3.3).

2.2.2 Executing Programs

Program text that has been keyed into the workspace can be immediately submitted for execution on SmaViM. First, though, you have to use the left mouse button to select the text. VisualWorks provides two methods for executing the selected text. The first is to left click **Do it** on the dropdown under the **Edit** menu in the workspace (Fig. 2.3).

You can achieve the same effect by clicking **Do it** on the context menu[4] (see Fig. 2.4).

[4]Open the context menu in Windows by right clicking the selected space. If the Mac OS mouse has only one button available, open the context by pressing **Ctrl** and clicking simultaneously.

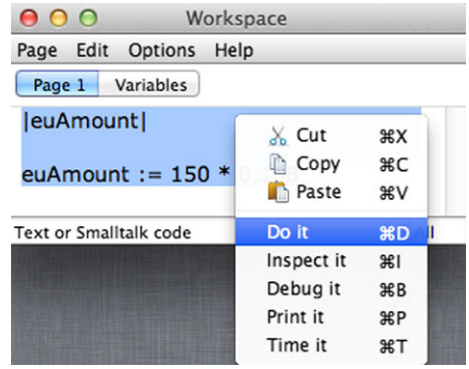


Fig. 2.4 Executing a program from the context menu

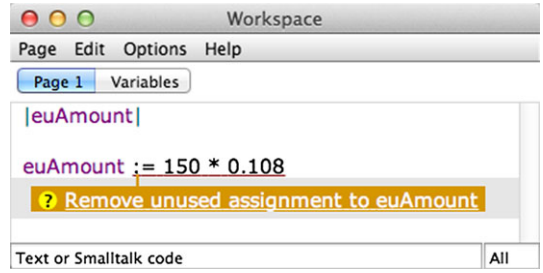


Fig. 2.5 Warning dialog

Whichever method one chooses, the warning dialogue shown in Fig. 2.5 appears, warning the program developer that a value is about to be calculated and assigned to the variable `euAmount`, but that the value assigned to this variable will not be used in the further course of the program. Clicking **remove** would remove the assignment, but that is not what we want at this point. If you hit the **Esc** key on the keyboard, the dialogue disappears.

In spite of the warning, the program is executed and the solution is assigned to the variable `euAmount`, but other steps are necessary before the value can be seen in the workspace. At first, one line is added to the program:

```
|euAmount|

euAmount := 150 * 0.108.
euAmount
```

Since we are now using the variable `euAmount`, the above dialogue will not appear anymore. Note that the second to last line has to be terminated by a full period.

Several methods are available for outputting the values of variables or expressions in SmaViM. The first method is to click **Print it** instead of **Do it** on one of the menus. Figure 2.6 shows the result.

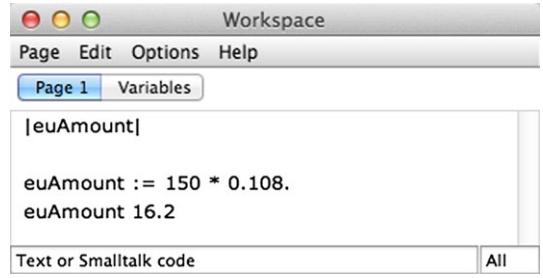


Fig. 2.6 Using **Print it** to execute a program

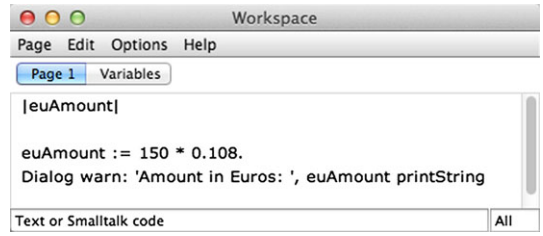


Fig. 2.7 Supplement to the program to display the result in a dialogue window

When you use **Print it** to execute a program, the value of the most recently calculated expression will be output in the workspace. Thus, Fig. 2.6 shows the value of the variable euAmount next to it.

Another possibility for SmaViM is to display the result in a small dialogue window. That possibility needs to be programmed, however. That is, an instruction to SmaViM must be added to our little program, which opens a dialogue window and displays the results in it. The instruction to do so might look like this:

```
Dialog warn: 'Amount in euros:', euAmount printString
```

At a later point, we will analyse in more detail the form and content of this instruction. To help you understand it, we'll say only this at this point: The instruction `Dialog warn:` opens a dialogue window and displays in it the text that appears after the colon. Texts that are to be displayed in this way as they are written can be any random character string enclosed between single quotes (for example, `'Amount in euros:'`). Numeric values, for example, the value of the variable `euAmount`, will be converted to text only when the command `printString` follows them. The comma between the two expressions serves to combine the two partial texts into one. This is necessary, because the instruction `Dialog warn:` expects a single string of characters as a parameter. Figure 2.7 shows our expanded program. In order to add the output instruction after the first instruction, it must be followed by a period. Stated more simply: two instructions following one another must be separated from one another by a period. If one selects the program text a second time and clicks **Do it**, the dialogue window shown in Fig. 2.8 appears on the screen.

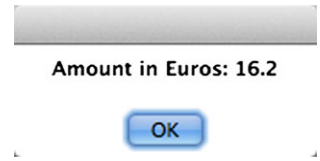


Fig. 2.8 Result output in a dialogue window

And so we have done it; we've solved the problem of converting an amount of currency with a static exchange rate and a specific amount in Swedish kronor. This solution is unsatisfactory, though, in that a change in the rate of exchange or in the amount to be converted requires a program change.

2.2.3 Adding Flexibility to the Currency Conversion

Quite obviously, a program that can convert a specific amount into another currency at an invariable exchange rate is not particularly useful. In order to make the program more flexible, let's first consider a generalised conversion formula:

`F': endAmount = startAmount · exchangeRate`

In this formula, it no longer matters which currencies are involved; all that's necessary is to use the appropriate rate of exchange. The result is the following Smalltalk program:

```
|endAmount startAmount exchangeRate|
endAmount := startAmount * exchangeRate.
Dialog warn: 'Converted amount: ', endAmount printString
```

Nevertheless, SmaViM cannot execute this program because it doesn't know the values for the `startAmount` and `exchangeRate` variables, and consequently cannot calculate their product. In the end, it's up to the user to say which amount is to be converted and at which rate of exchange. This requires instructions that allow the user to enter numeric values during execution of the program, and to allocate those values to the variables `startAmount` and `exchangeRate`. Once again, simple dialogue windows are available for data entry. The instruction

```
Dialog request: 'Amount to convert: ' initialAnswer: '0'
```

produces the dialogue window shown in Fig. 2.9. Any text (enclosed between single quotes) can occur after `request:`; the text appears at the top of the dialogue window (understood in this case as an entry prompt). The field beneath the text serves as a user input field. The value entered after the keyword `initialAnswer:` appears as a default

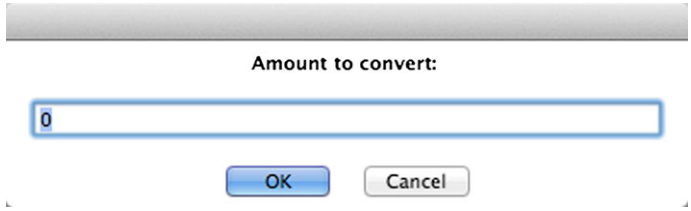


Fig. 2.9 Data-entry dialogue

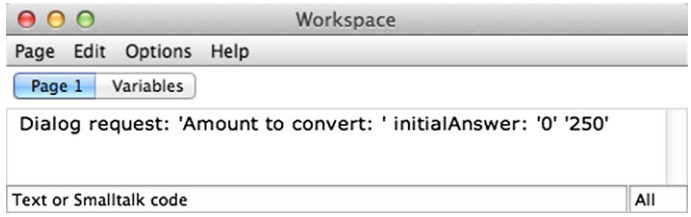


Fig. 2.10 Result of the entry in the dialogue window

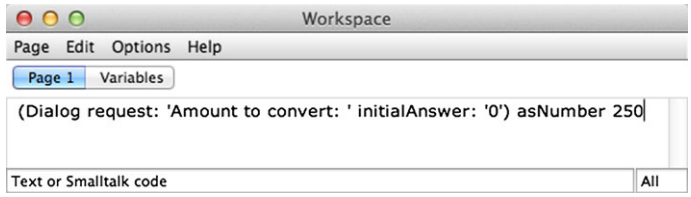


Fig. 2.11 Conversion of the dialogue entry into a number

value in the input field, and will be used as the value if the user enters nothing else. When one uses the command **Print it** to execute the above instruction in the workspace and enters a value of, say, 250 in the input field, once the dialogue entry has been confirmed by clicking the **OK** button, the text `'250'` appears as a result (see Fig. 2.10).

The result of an entry in a dialogue window is always text. Because, however, we can only use numbers to calculate, this text must now be converted into a number; this is, of course, possible only if the entered text can be treated as a number. The exact process involved, that is, which syntactic rules apply for using numbers, is considered in Sect. 8.1.2. The instruction `asNumber` is used to convert text into a number. Figure 2.11 shows both the expanded instruction and the result that appears after using **Print it** to execute the program. This result is now no longer text, but is instead the *number* 250—note the missing single quotes (compare Fig. 2.10).

Numbers entered in this way can now be assigned to the variables `startAmount` and `exchangeRate`. The following partial program accomplishes this:

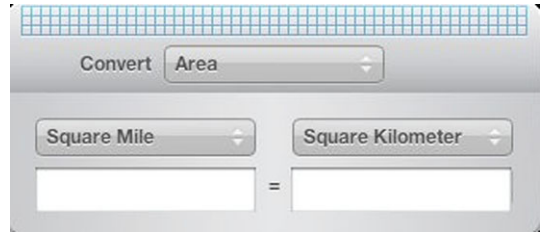


Fig. 2.12 Conversion program with GUI

```
startAmount := (Dialog request: 'Amount to convert: '
                          initialAnswer: '0') asNumber.
exchangeRate := (Dialog request: 'Exchange rate: '
                          initialAnswer: '1') asNumber
```

Note that the instructions shown here for the dialogue entries wrap over two lines each. This is not required, but is permitted. Spaces and line breaks usually have no meaning in Smalltalk programs.

Now we can complete our program for currency exchange:

```
|endAmount startAmount exchangeRate|
startAmount := (Dialog request: 'Amount to convert: '
                          initialAnswer: '0') asNumber.
exchangeRate := (Dialog request: 'Exchange rate: '
                          initialAnswer: '1') asNumber.
endAmount := startAmount * exchangeRate.
Dialog warn: 'Converted amount: ', endAmount printString
```

This program now lets any amount be converted at any rate of exchange. We should mention here that the process shown here, in which entries are made and displayed using primitive dialogue windows, has nothing to do with the kind of ergonomically acceptable GUI that one expects to find in modern application programs. The result of a "professionally" designed program would look something like Fig. 2.12. Such a program would of course search the Internet for the latest exchange rate and would also be capable of handling values in other units than currencies.

Of course, Smalltalk also allows you to design programs with GUIs that access the Internet or databases. Chapter 16 offers an introduction to those topics.

2.3 Case Study: Solving a Quadratic Equation

This section describes the design of algorithms in a somewhat more complex example. Here we are going to develop a program to solve quadratic equations.

2.3.1 The Algorithm

Let's start by looking at the following quadratic equation:

```
G: x^2 + 2.1x - 5.4 = 0
```

We look for all of G's solutions.

Mathematics teaches us that we can solve G by using the familiar "pq formula":

$$L: \quad x_{1,2} = -\frac{2.1}{2} \pm \sqrt{\frac{2.1^2}{4} + 5.4}$$

We must expand SmaViM's capabilities if we want to use it to solve this formula. If we had a machine that understood mathematical language (such as L uses), we would be finished, because the formula itself represents the machine-readable formulation of the problem-solving method. Let's assume, though, that SmaViM does not have such an ability. That is not necessary, though, because the formula itself shows how the quadratic equation can be solved using the four basic arithmetic calculations and the derivation of a square root. We can assume therefore that our machine is capable of performing those very operations, and that it can evaluate formulas that contain those operations. (The first case study already showed that the machine could multiply.)

- Addition, subtraction, multiplication and division
- Solving for the square root of a non-negative number
- Evaluation of expressions (formulas) using the operations named above, along with obeying parenthetical ordering (as is usual in mathematics)

In addition, we assume that the machine is capable of solving only one formula at any single moment. The formula L, though, actually contains two formulas for calculating the two solutions to the quadratic equation. For that reason, we note a specific formula for each solution, which the machine will solve in the specified order:

$$x_1 = \frac{-2.1 + \sqrt{2.1 \times 2.1 + 4 \times 5.4}}{2}$$

$$x_2 = \frac{-2.1 - \sqrt{2.1 \times 2.1 + 4 \times 5.4}}{2}$$

Using a simple algebraic redesign, the squaring was replaced by a multiplication.

As already explained in Sect. 2.1, when we create the text for the program, we have to limit ourselves to those characters that can be found on a standard typewriter or PC keyboard. Like the multiplication symbol, the radical and fraction bar are also not available on those keyboards. We can replace the fraction bar as a division sign with the forward

slash (/), and instead of a radical sign, we use the word symbol "sqrt."[5] Finally we again replace any decimal commas with decimal points; do not use indexed variables, which would not be possible on a standard keyboard; and again replace the equal sign with ":=." This results in:

```
x1:= (-2.1 + (2.1*2.1 + 4*5.4) sqrt)/2.
x2:= (-2.1 - (2.1*2.1 + 4*5.4) sqrt)/2
```

Notice that the word symbol sqrt is placed after the parenthetical expression from which the square root is to be derived. This writing convention accords with the rules of the Smalltalk language that we will use to write our algorithms.

► **Note for people who know conventional higher programming languages like PASCAL:** These languages customarily use a *functional* notation in which the word symbol `sqrt` (the name of the function) is written in front of the parenthetical expression:

```
x1:= (-2.1 + sqrt(2.1*2.1 + 4*5.4))/2.
x2:= (-2.1 - sqrt(2.1*2.1 + 4*5.4))/2
```

At this point, the algorithm is almost in a format that we could submit for execution to an actual machine. First, though, we should correct a little "blemish." The algorithm has to derive the same square root twice, which one should avoid. That is because the root is derived in yet another numerical procedure, an algorithm. This requires an expenditure of effort that should not have to unnecessarily occur twice. For that reason, we modify the algorithm so that the value of the square root is derived at the beginning of the program. This interim result becomes a variable that we call `root`, which we use subsequently to calculate the values of `x1` and `x2`:

```
root := (2.1*2.1 + 4*5.4) sqrt.
x1 := (-2.1 + root)/2.
x2 := (2.1 - root)/2
```

2.3.2 The Program

We will now expand the algorithm we wrote in the last section to include variable declarations for root, x1 and x2, as well as an output instruction. The program now reads:

[5] "Sqrt" means square root.

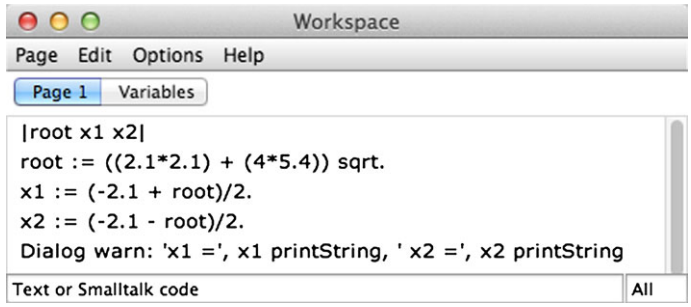
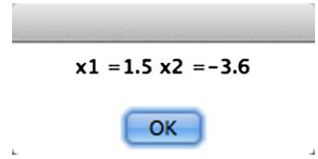


Fig. 2.13 Workspace display of a program to solve a quadratic equation

Fig. 2.14 Solution of a quadratic equation




```
|root x1 x2|
root := ((2.1*2.1) + (4*5.4)) sqrt.
x1 := (-2.1 + root)/2.
x2 := (-2.1 - root)/2.
Dialog warn: 'x1 =', x1 printString,
             'x2 =', x2 printString
```

Notice that the three commas in the output instruction combine the four lines of partial text into a single line of text. That is because, as explained above, the instruction `Dialog warn:` expects exactly one text string as a parameter.

If one enters the programming text in a workspace (as in Fig. 2.13) and then uses the **Do it** command from the **Smalltalk** menu to execute it, the result shown in Fig. 2.14 appears.

And so we've now used our SmaViM machine to solve a quadratic equation.

Naturally we can now change the program so that SmaViM can solve a different equation. But that's an exhausting process that's prone to error, as the following example shows. To solve the quadratic equation

`G':` $x^2 + 2.1x + 5.4 = 0$

the same procedure we followed above would yield the program shown in Fig. 2.15. If we attempted to execute the program, though, SmaViM would display the error message shown in Fig. 2.16. Since SmaViM cannot derive square roots from negative numbers, the program cannot be used to solve `G'`.

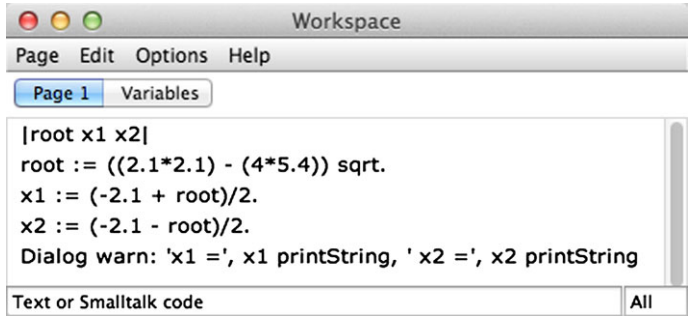


Fig. 2.15 (Incorrect) program for solving G'

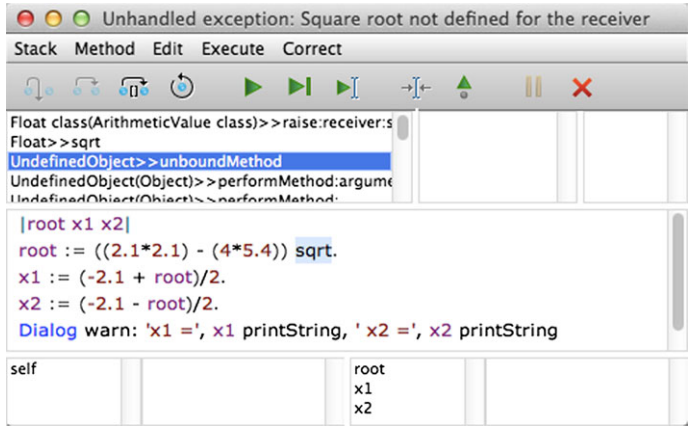


Fig. 2.16 Error message in SmaViM

In the next section, we will therefore again look at the concept of algorithms and systematically try to figure out how we can generalise the problem of solving quadratic equations in such a way that we do not need to change the program for each new calculation.

2.3.3 Generalising the Solving of Quadratic Equations

Mathematics customarily presents this generalised format for a quadratic equation:

$Q\colon\ ax^2 + bx + c = 0$

What we are looking for is a method that will solve Q for all real numbers (coefficients) a, b and c.

Using the pq formula on the equation yields this:

$$L_q\colon\quad x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

L_q cannot be entered into a program in the same way that we solved L in the previous example. What happens, for example, if $a = 0$? We already made reference in the last section to the problem that arises when the radicand (the expression within the radical) might be negative. If we are going to use SmaViM to solve this general equation, that is, to be able automatically to solve for all values of a, b and c, then SmaViM must master the concept of *case-by-case analysis*, because the same formula cannot be used to solve for all values of the coefficients. In other words, SmaViM must be able to make the execution of program steps dependent on conditions.

In order to use case-by-case analysis to automatically solve a problem, you must always be sure that the list of possible cases is complete. That means that, within a program, all cases of partial solutions that arise from the problem result in program steps that are tied to conditions.

In the case where $a = 0$, the equation Q "degenerates" to a linear equation:

```
bx + c = 0
```

It would appear that there is no problem in solving this equation. Taking into account the condition $a = 0$, we write:

```
if a = 0 then x = -c/b
```

In the case $a \neq 0$, the following subcases must be distinguished, which analysis of the radicand ($b^2 - 4ac$) yields:

3. *radicand* ≥ 0: Using the formula L_q, we get a standard quadratic equation with two real-number solutions. If *radicand* $= 0$, the two solutions are identical.
4. *radicand* < 0: There are no solutions in the set of real numbers. We will ignore at this point the possibility of complex numbers.

We could continue to describe the algorithm in the style we started above:

```
if a = 0 then x = -c/b
if a > 0 then x 1 =...
```

Instead of that, though, we'll write it in a form that uses Smalltalk syntax. Instead of

```
if a = 0 then x = -c/b
```

we write:

```
(a=0) ifTrue: [x := c negated / b]
```

First we write the condition (in this case `a = 0`), enclosed between parentheses, followed by the instruction `ifTrue: [...]`. This means that subsequent instructions written between square brackets (in this case, the single assignment `x := c negated / b`) will be executed if and only if the condition before `ifTrue:` is fulfilled. If the condition is not fulfilled, nothing happens. A series of instructions occurring between square brackets is called a *block*. In Smalltalk, the leading minus sign must be replaced with the word `negated` placed after the variable.

► **Note** Notice the various meanings of the equal sign, which can be used to formulate a condition or to assign a value to a variable (":="). Note also that Smalltalk programming is case-sensitive, and that the colon in `ifTrue:` is part of the instruction and must not be separated from the alphabetic characters. (In other words, `iftrue:` and `ifTrue:` are both invalid spellings.) In general, though, one can say, somewhat simplifying, that spaces and carriage returns can occur anywhere except within names or instructions.

Let's expand the case-by-case analysis based on the above considerations:

```
1  (a = 0) ifTrue: [x:= c negated / b].
2  (a ~= 0) ifTrue: [
3     radicand := (b * b) - (4 * a * c).
4     radicand >= 0)
5        ifTrue: [
6              root:= radicand sqrt.
7              x1:= (b negated + root) / (2 * a).
8              x2:= (b negated - root) / (2 * a)].
9     (radicand < 0)
10      ifTrue: ["no real-number solution"]]
```

We've numbered the lines of code to simplify the following explanations.

Line 1 deals with the familiar case of a linear equation. In line 2, the character string ~= means ≠. If $a \neq 0$, we are dealing with an "authentic" quadratic equation, and the subcases described above must be dealt with. This occurs in the `ifTrue:` block, which is part of the condition (`a ~= 0`), which starts with the initial square bracket in line 2 and ends with the final square bracket at the end of line 10. Since the solution of the quadratic equation depends on subcases that are determined by the value of the radicand, that value is calculated in line 3 and assigned to the variable `radicand`. Line 4 then checks whether

the value of the radicand is greater than or equal to 0. If that is true (that is, if the value is ≥ 0), the instructions in the associated `ifTrue:` block are executed, beginning with the opening square bracket in line 5 and ending with the final square bracket at the end of line 8.

► **Note** In Smalltalk, the period serves to separate successive instructions. For this reason, a period isn't needed at the end of a series of program steps. The same is true for a series of instructions contained within square brackets. For that reason, there is no period **before** the final square bracket in line 8.

In the event that the value of the radicand is negative—which is checked in line 9—we have initially not programmed a series of instructions. For now, the `ifTrue:` block (line 10) simply contains a comment. Comments are character strings enclosed between double quotation marks that can be inserted anywhere to explain the program text. When the program is executed, SmaViM simply ignores them.

Please notice the nesting of the individual cases. The `ifTrue:` block for the `(a ~= 0)` condition contains additional `ifTrue:` instructions for the subcases.

A thorough examination of the program, however, makes clear that the very first line already contains a violation of the precept that the case-by-case analysis covers every case. What happens if $b = 0$? The value of the expression `c negated / b` is not defined.

What we obviously need is a systematic list of all relevant cases. Which cases are relevant, that is, those which must be differentiated with regard to the posited solution method, is a question that depends on the problem being solved, and it is not always obvious. In our example, though, we are dealing with a problem sufficiently familiar from mathematics, where it is not particularly difficult to find the relevant cases:

$a = b = c = 0$: This is a trivial case, since every value of x is a possible solution to the equation.

$a = b = 0$ **and** $c \neq 0$: Because c cannot simultaneously be equal to and not equal to 0, there is a contradiction, and the equation has no solution.

$a = 0$ **and** $b \neq 0$: This is a linear equation with the solution $x = -c/b$.

$a \neq 0$: Now we have a quadratic equation. With regarded to possible real-number solutions, we must proceed to distinguish among the various radicands in the formula L_q, as described above. At the same time, one has to consider a case in which the radicand is equal to 0; in that case, one can avoid deriving the square root of 0. This leads to the following three subcases:

radicand $= 0$: The solution is $x = -b/2a$
radicand > 0: Two solutions based on formula L_q.
radicand < 0: No real-number solution

This yields the following algorithm to solve Q:

```
(a = 0)
   ifTrue:
     [(b = 0)
        ifTrue:
          [c = 0 ifTrue: ["Trivial solution"].
           c ~= 0 ifTrue: ["Equation unsolvable"]].
      b ~=0 ifTrue: [x := c negated / b]].
(a ~= 0)
   ifTrue:
      [radicand := b * b - 4 * a * c.
       radicand = 0 ifTrue: [x:= b negated / (2 * a)].
       (radicand > 0)
          ifTrue:
             [root := radicand sqrt.
              x1 := b negated + root / (2 * a).
              x2 := b negated - root / (2 * a)].
       (radicand < 0)
          ifTrue: ["No real-number solution"]]
```

► **Note** For now, in cases where no explicit solution can be calculated, the algorithm contains comments. Lines are indented in the above example to improve legibility of nested distinction of cases. Although SmaViM recognises instructions that are enclosed between square brackets as belonging together, it helps human readers to place alternative cases at the same indented position.

In the event that mutually exclusive cases appear in an algorithm (for example, $c = 0$ and $c \neq 0$), they can be combined in a single alternative in Smalltalk by using the instruction `ifTrue:ifFalse:`. Instead of writing:

```
(c = 0)
   ifTrue: ["Trivial solution"].
c ~= 0
   ifTrue: ["Equation unsolvable"]]
```

we can write:

```
(c = 0)
   ifTrue: ["Trivial solution"]
   ifFalse: ["Equation unsolvable"]]
```

This not only serves to make the program more readable to humans, but also saves SmaViM the effort of checking two times for the value of the variable c.

If we apply this technique to the entire algorithm, we end up with:

```
(a = 0)
   ifTrue:
     [(b = 0)
       ifTrue:
          [(c = 0)
            ifTrue: ["Trivial solution"]
            ifFalse: ["Contradiction"]]
       ifFalse: [x := c negated / b]]
   ifFalse:
     [radicand:= (b*b) - 4 * a * c).
     (radicand = 0)
        ifTrue: [x := b negated / (2 * a)]
        ifFalse:
           [(radicand > 0)
              ifTrue:
                 [root:= radicand sqrt.
                  x1:= b negated + root / (2 * a).
                  x2:= b negated - root / (2 * a)]
              ifFalse: ["No real-number solution"]]
```

Notice here how the case distinctions are nested. Readers should review the program until it is clear to them that—depending on the values for a, b and c—the algorithm always contains only one alternative path, that is, only one `ifTrue: / ifFalse:` branch of the program is executed. To help make this clearer, Fig. 2.17 shows a so-called decision tree.[6]

Each rhomboid node in the tree is a branch where decisions need to be made, until finally exactly one of the actions—shown as squares—must be chosen. One can think of these squares as the leaves of the decision tree.

When constructing a program, it's essential to use indents to make the structure of the algorithm clearly visible, since the structure is primarily determined by case distinctions. One important point is that the instances of the instructions `ifTrue:` and `ifFalse:` belonging to the same alternative should be aligned. At the same time, it's important to remember that the layout of the program text matters only to human readers; it is totally meaningless for SmaViM as it executes a program.

[6] In computer science, the root of a tree is usually shown at the top of a diagram.

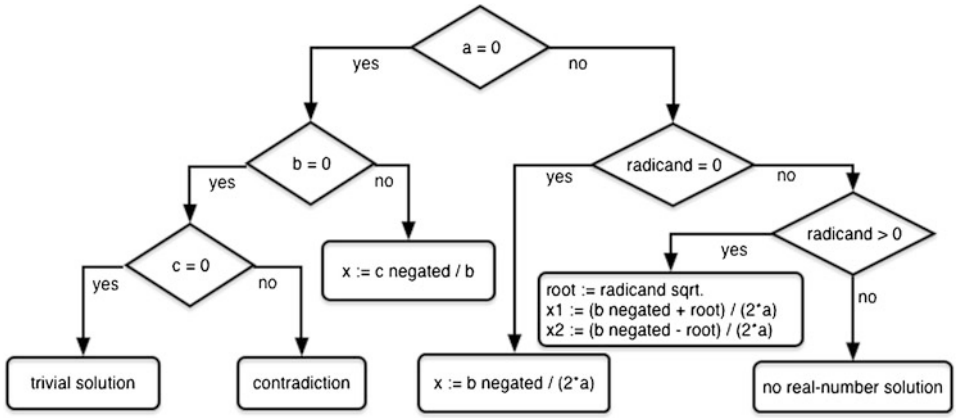


Fig. 2.17 Decision tree

In order to turn the algorithm into a usable program, you must add instructions to provide for the entry of coefficients and the output of results. For this purpose, use the dialogue windows described in Sects. 2.2.2 and 2.2.3. This yields the following program:

```
|a b c x radicand root x1 x2|
a := (Dialog request: 'a=' initialAnswer: '0') asNumber.
b := (Dialog request: 'b=' initialAnswer: '0') asNumber.
c := (Dialog request: 'c=' initialAnswer: '0') asNumber.
(a = 0)
   ifTrue:
      [(b = 0)
        ifTrue:
           [(c = 0)
               ifTrue: [Dialog warn: 'Trivial solution']
               ifFalse: [Dialog warn: 'Contradiction']
        ifFalse:
           [x := c negated / b.
            Dialog warn: 'x = ', x printString]]
   ifFalse:
      [radicand := (b*b) - 4 * a * c.
      (radicand = 0)
        ifTrue:
           [x := b negated / (2 * a).
            Dialog warn: 'x = ', x printString]
```

```
        ifFalse:
           [(radicand > 0)
              ifTrue:
                 [root := radicand sqrt.
                  x1 := b negated + root / (2 * a).
                  x2 := b negated - root / (2 * a).
                  Dialog warn: 'x1 = ', x1 printString,
                               'x2 = ', x2 printString]
              ifFalse:
                 [Dialog warn:'No real-number solution']]
```

As before, the first line contains the declaration of the required temporary variables.

2.4 Summary

If we contrast the two alternatives for solving a particular problem, using either human or mechanical effort, we reach the following conclusions:

The human must determine the method by which the solution can be discovered. This can occur either by coming upon the solution by analysing the problem, that is, by thinking about it, or else by finding an existing solution to the problem, either by researching it in a relevant text or by remembering that one had already solved the problem at an earlier date. At bottom, though, a human has to discover the method at least once.

If it's a familiar method, the concept behind the solution can be applied to the concrete problem.

The machine, on the other hand, doesn't have to bother with a method. A program supplies the machine with the method in the form of a program. Once the machine has the program, it can solve the problem by applying the method to supplied values. It executes the program.

In other words, a human is assigned, in the form of a problem description, the task of solving a problem. The description establishes what is to be accomplished. The way the problem is solved does not, in principle, matter to the person who assigns the problem. If the solution to the problem has an economic value, the only side issues are that the solution should be found as quickly as possible and that it require the work of as few people as possible. It should cost as little effort as possible.

The machine doesn't know what kind of a problem it is solving. The program tells the machine exactly what it needs to do.

On an economic level, therefore, it pays to employ a machine if its purchase and subsequent upkeep during its working life cost less than the labour it replaces by means of its capabilities, that is, its mechanical functioning. That will always be true when the problem solutions (program executions):

- Are required repeatedly (for example, payroll accounting)
- Require intensive calculation (for example, construction statics)

Machines also continue to conquer new fields of problem resolution, based on the following situations:

- Machines are fast and secure. They make possible, for example, the solution to problems involving space travel (travel to the moon and landing on it), military engineering and weather prediction.
- At base, machines like modern computers have universal applicability. In principle, they can solve any problem that's capable of a mechanical solution.

This final ability—computers' ability to solve all mechanically solvable problems—is called their programmability. It means that a machine can, in principle, master every problem-solving method that can be put into machine-solvable terms based on the computer's design and the mechanical functioning attendant on its design. In other words, a machine can execute any program. SmaViM also has that ability.

How Does One Find an Algorithm?

Back at the beginning of the chapter we discussed the difficulties involved in finding a correct method to solve a given problem and in turning that method into an error-free program. Computer science has so far been unable to develop a methodology for this process that can guarantee error-free programs. Anyone who spends time working with computers realises that software does not always satisfy the demands put on it, nor does it always function correctly. To name just one spectacular example, the military has been known to explode rockets shortly after take-off because a software error would have caused an even worse catastrophe. One should always keep such incidents in mind and carefully examine the use of computers, especially in cases where human safety depends on the correct functioning of a computer-controlled system.

One of the processes that can support programmers as they develop algorithms is so-called *step-by-step refinement* (Wirth 1971). An entire task is broken down into partial tasks. The partial tasks are in turn broken down into even smaller tasks until their solutions can be expressed in simple terms in the selected programming language. In Sect. 4.1, we will return to this process as we work through an example. Object-oriented software design uses a similar method, which is called the *composed-method* model (Beck 1997); we will discuss it in Sect. 14.3.

Separating Algorithms and User Interaction

The final version of the program used to solve quadratic equations, presented at the end of Sect. 2.3.3, contains both instructions that serve the actual calculation of solutions as well as others that accomplish interactivity with the user, that is, the instructions that allow the user to enter data or view results. The two types of instruction are only tangentially related

to one another. The solution method is independent of the source of the entries or what happens to the calculations after they have finished. There is a great deal to be said for separating the two program components. That would allow the solution process to be used in another context, where perhaps the input data is supplied by another part of the program and the results are processed by yet another, requiring user action in neither case.

The following chapters concentrate primarily on converting solution processes using the medium of an object-oriented programming language. The topic of user interactivity does not play a role until Chap. 16. Chapter 7 presents a version of the quadratic-equation program that does not contain the input/output instructions. Section 14.4 and Chap. 16 both discuss the separation of problem solution and user interactivity.

http://www.springer.com/978-3-658-06822-6

Programming Smalltalk – Object-Orientation from the Beginning
An introduction to the principles of programming
Brauer, J.
2015, XXI, 429 p. 277 illus., 1 illus. in color., Softcover
ISBN: 978-3-658-06822-6