Chapter 6

Making Programs and Documenting Them

Abstract Programs are more than a set of instructions, which when executed produces the desired result. Programming is an activity, and a program is the result of a process, in which a problem has been expressed, analyzed, subdivided, implemented, tested, and possibly rephrased. And often the process and its result are to be wrapped and documented for it to be useful by the programmer or others. In this chapter, we will zoom out, and focus on some of these surrounding processes. The chapter will describe:

- How to design functions.
- How and why to document programs using in-code documentation.

6.1 The 8-step Guide to Writing Functions

Pólya's problem-solving technique described in Section 1.2 is a useful starting point for solving problems, and for the object-oriented programming paradigm to be discussed in later chapters Chapter 17, approaches such as Pólya's have been put into systems. Regardless of the origin, there is always a point, where a programmer has to focus on small-scale problems such as: Which functions, should be used, and how should the functions be designed? This is not an area heavily investigated in the literature, but often a skill that programmers pick up by actively engaging in programming alone or with other programmers. However, here I will venture a recipe for designing functions, which I and my colleagues call The 7-step guide to writing functions. It is not meant as the ultimate guide or as required steps, but it is our experience that these steps contain essential elements that consciously or perhaps unconsciously always take part in designing useful and reusable functions.

To decide which functions to write and how to write them:

1. Note: Write a short note on what a function should do.
2. Name: Invent a name for the function. Semantically meaningful names should be preferred.
3. External test: Write a small test program, which uses the yet-to-be-written function.
4. Type: Decide what type, the function should have, e.g., by how you used it in your test program.
5. Implement: Write the function and possibly its helper functions.
6. Internal test: Extend your test program with more examples, where you use the function and based on its implementation.
7. Run: Run the test program
8. Document: Write brief in-code documentation of the function, see Section 6.2.

As an example, let us revisit the problem of solving a quadratic equation: The task is to find the zero-crossings of a second-degree polynomial, i.e., $f(x) = ax^2+bx+c = 0$. The process could be as follows:

1. Note: We decide to stick to the mathematical description:

> Given parameters a, b, and c, the function should return the 0, 1, or possibly 2 locations x, where $f(x) = 0$.

2. Name: This function may be used together with solvers for other equations, so we decide to give it the rather long name

   `solveQuadraticEquation`.

3. External test: We decide on a single test to get a feeling of how the function is to be used:

   Listing 6.1: Defining the function sum
   ```
   let p = solveQuadraticEquation 1.0 0.3 -1.0
   printfn "0=1.0x^2+0.3x-1.0 => x = %1" p;;
   ```

4. Type: Since there may be 0-2 points x, where $f(x) = 0$, the output answers could be a tuple. Further, since $a, b, c, x \in \mathbb{R}$, we will use floats. Hence,

   `solveQuadraticEquation -> float -> float -> float -> float*float`.

5. Implement: Thinking about how to write `solveQuadraticEquation`, we decide that since the calculation of the discriminant is done twice, we will add it as a helper function. Our resulting code is:

   Listing 6.2: Defining the function sum
   ```
   let discriminant a b c = b ** 2.0 - 4.0 * a * c

   let solveQuadraticEquation a b c =
     let d = discriminant a b c
     ((-b + sqrt d) / (2.0 * a),
      (-b - sqrt d) / (2.0 * a))
   ```

6. Internal test: Working with the code, we realize that it is unclear, what happens, when there are 0 og 1 solutions, so we update the external test and add more tests:

   Listing 6.3: Defining the function sum
   ```
   let p1 = solveQuadraticEquation 1.0 0.3 -1.0
   printfn "0=1.0x^2+0.3x-1.0 => x = %A" p1
   let p2 = solveQuadraticEquation 1.0 0.0 0.0
   printfn "0=1.0x^2+0.3x-1.0 => x = %A" p2
   let p3 = solveQuadraticEquation 1.0 0.0 1.0
   printfn "0=1.0x^2+0.3x-1.0 => x = %A" p3
   ```

7. Run: The complete code with examples and its output, when executed is shown in Listing 6.4

**Listing 6.4 solveQuadraticEquation.fsx:
Solving quadratic equations**

```
let discriminant a b c = b ** 2.0 - 4.0 * a * c

let solveQuadraticEquation a b c =
  let d = discriminant a b c
  ((-b + sqrt d) / (2.0 * a),
   (-b - sqrt d) / (2.0 * a))

let p1 = solveQuadraticEquation 1.0 0.3 -1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p1
let p2 = solveQuadraticEquation 1.0 0.0 0.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p2
let p3 = solveQuadraticEquation 1.0 0.0 1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p3
```

```
$ dotnet fsi solveQuadraticEquation.fsx
0=1.0x^2+0.3x-1.0 => x = (0.8611874208, -1.161187421)
0=1.0x^2+0.3x-1.0 => x = (0.0, -0.0)
0=1.0x^2+0.3x-1.0 => x = (nan, nan)
```

8. Document: The following section will discuss how to perform in-code documentation and use Listing 6.4 as an example.

6.2 Programming as a Communication Activity

Documentation is a very important part of writing programs since it is most unlikely that you will be writing really obvious code. Moreover, what seems obvious at the point of writing may be mystifying months later to the author and to others. Documentation serves several purposes:

1. Communicate to the user of the code, what it does and how to use it. In this book, we will emphasize the XML-standard for this purpose.

2. Highlight big insights essential for the code, which is important for other programmers to understand and maintain the code.

3. Highlight possible conflicts and/or areas where the code could be changed later, which is also targeted programmers rather than users of the code.

The essential point is that coding is a journey in problem-solving, and proper documentation is an aid in understanding the solution and the journey that lead to it. Documentation is most often a mixture of in-code documentation and accompanying documents. Here, we will focus on in-code documentation which arguably causes problems in multi-language environments and run the risk of bloating code. Since documentation is about human-to-human communication, there is no correct documentation. However, as in all things, documentation can both be too little and too much, and the ability to produce documentation is best learned by example and by doing.

F# has two different syntaxes for comments. Comments can be block comments:

Listing 6.5: Block comments.

```
(*<any text>*)
```

The comment text `(<any text>)` can be any text and is stilled parsed by F# as keywords and basic types, implying that `(* a comment (* in a comment *) *)` and `(* "*)" *)` are valid comments, while `(* " *)` is invalid.

Alternatively, comments may also be line comments,

Listing 6.6: Line comments.

```
//<any text>
```

where the comment text ends after the first newline.

The block and line comments are used principally for communicating insights and comments into the code between programmers who want to understand and/or maintain the code.

Users of the code, are most likely also programmers but have an outside perspective. They are more interested in what the code does, and how it is to be used. For this we recommend the *Extensible Markup Language* documentation standard (*XML-standard*) [1]. All lines of the XML-standard start with a triple-slash `///`. Thus, it is a line-comment, where an extra slash has been added for visual flair. XML consists of tags which always appear in pairs, e.g., the tag "tag" would look like `<tag> ... </tag>`. A subset of tags are listed in Table 6.1. If no tags are used, then it is automatically assumed to be a `<summary>`. An example of a documented script is shown in Listing 6.7. is:

Several tools exist that extract the comments from source code and reorder the comments into manual type structures, such as Doxygen. Popular output from such tools is both HTML and LaTeX. However, for this text, the usage of the XML-standard as a way to standardize comments will suffice.

---

[1] For specification of C# documentations comments see ECMA-334: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`

Tag	Description
`<c>`	Set text in a code-font.
`<code>`	Set one or more lines in code-font.
`<example>`	Set as an example.
`<exception>`	Describe the exceptions a function can throw.
`<list>`	Create a list or table.
`<para>`	Set text as a paragraph.
`<param>`	Describe a parameter for a function or constructor.
`<paramref>`	Identify that a word is a parameter name.
`<permission>`	Document the accessibility of a member.
`<remarks>`	Further describe a function.
`<returns>`	Describe the return value of a function.
`<see>`	Set as link to other functions.
`<seealso>`	Generate a See Also entry.
`<summary>`	Main description of a function or value.
`<typeparam>`	Describe a type parameter for a generic type or method.
`<typeparamref>`	Identify that a word is a type parameter name.
`<value>`	Describe a value.

Table 6.1 Recommended XML tags for documentation comments, from ECMA-334 3rd Edition, Annex E, Section 2.

6.3 Key Concepts and Terms in This Chapter

This chapter has considered elements that are an important part of the activity of programming, but to some extent complement the specific act of writing source code. You have seen:

- How to use the **7-step guide** to design functions, which emphasizes writing examples of function usage before implementing the function itself.
- Write **in-code** documentation to support the understanding of the code.
- Documentation is written for programmers and there are at least two different types: **users** and **maintainers**.
- The **XML standard** uses `///`, is for both types of programmers, and documents what a program does and how it is to be used.
- The **line** and **block** comments are for implementation-specific details and intended to be read by programmers who seek to understand and maintain the code.
- There is no such thing as the correct documentation, but you are well advised to follow the XML standard and to improve your skill by writing documentation and sharing it with others.

**Listing 6.7 commentExample.fsx:
Code with XML comments.**

```
/// The discriminant of a quadratic equation with
/// parameters a, b, and c
let discriminant a b c = b ** 2.0 - 4.0 * a * c

/// <summary>Find x when 0 = ax^2+bx+c.</summary>
/// <remarks>Negative discriminants are not checked.</remarks>
/// <example>
///   The following code:
///   <code>
//      let p = solveQuadraticEquation 1.0 0.3 -1.0
///     printfn "0=1.0x^2+0.3x-1.0 => x = %A" p
///   </code>
///   prints <c>0=1.0x^2+0.3x-1.0 => x = (0.9, -1.2)</c>.
/// </example>
/// <param name="a">Quadratic coefficient.</param>
/// <param name="b">Linear coefficient.</param>
/// <param name="c">Constant coefficient.</param>
/// <returns>The solution to x as a tuple.</returns>
let solveQuadraticEquation a b c =
  let d = discriminant a b c
  ((-b + sqrt d) / (2.0 * a),
   (-b - sqrt d) / (2.0 * a))

let p1 = solveQuadraticEquation 1.0 0.3 -1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p1
let p2 = solveQuadraticEquation 1.0 0.0 0.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p2
let p3 = solveQuadraticEquation 1.0 0.0 1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p3
```

Part II

Declarative Programming Paradigms

A programming problem may have many solutions, e.g., squaring a real value, x^2, can in F# be written as both `x*x` and `x**2.0`, and more complicated problems typically have many valid solutions. Particularly long programs can be complex and can have a high risk of programming errors. Different programming languages offer different structures to aid the programming in managing complex solutions, which is sometimes called a *programming paradigm*. Paradigms may be classified as either *declarative* or *imperative*. Some languages such as F# are multiparadigm, making the boundary between programming paradigms fuzzy, however, in their pure form, programs in declarative programming languages are a list of properties of the desired result without a specification on how to compute it, while programs in imperative programming languages is a specific set of instructions on how to change *states* on the computer in order to reach the desired result. This part will emphasize the *functional programming paradigm*, which is a declarative paradigm. In Part III, the *imperative* and the *object-oriented programming paradigms* will be emphasized.

Functional programming is a style of programming which performs computations by evaluating functions. Functional programming is declarative in nature, e.g., by the use of value- and function-bindings – `let`-bindings – and avoids statements – `do`-bindings. Thus, all values are constants, and the result of a function in functional programming depends only on its arguments. It is deterministic, i.e., repeated call to a function with the same arguments always gives the same result. In functional programming, data and functions are clearly separated, and hence data structures are dum as compared to objects in object-oriented programming paradigm, see Part III. Functional programs clearly separate behavior from data and subscribes to the view that *it is better to have 100 functions operate on one data structure than 10 functions on 10 data structures*. Simplifying the data structure has the advantage that it is much easier to communicate data than functions and procedures between programs and environments. The .Net, mono, and java's virtual machine are all examples of an attempt to rectify this, however, the argument still holds.

The functional programming paradigm can trace its roots to lambda calculus introduced by Alonzo Church in 1936 [1]. Church designed lambda calculus to discuss computability. Some of the forces of the functional programming paradigm are that it is often easier to prove the correctness of code, and since no states are involved, then functional programs are often also much easier to parallelize than other paradigms.

Functional programming has a number of features:

Pure functions
: Functional programming is performed with pure functions. A pure function always returns the same value, when given the same arguments, and it has no side-effects. A function in F# is an example of a pure function. Pure functions can be replaced by their result without changing the meaning of the program. This is known as *referential transparency*.

higher-order functions

Functional programming makes use of higher-order functions, where functions may be given as arguments and returned as results of a function application. higher-order functions and *first-class citizenship* are related concepts, where higher-order functions are the mathematical description of functions that operator on functions, while a first-class citizen is the computer science term for functions as values. F# implements higher-order functions. The `List.map` is and example of a higher-order functions.

Recursion

Functional programs use recursion instead of `for`- and `while`-loops. Recursion can make programs ineffective, but compilers are often designed to optimize *tail-recursion* calls. Common recursive programming structures are often available as optimized higher-order functions such as *iter*, *map*, *reduce*, *fold*, and *foldback*. F# has good support for all of these features.

Immutable states

Functional programs operate on values, not on *mutable values* also known as *variables*. This implies *lexicographical scope* in contrast to mutable values, which implies *dynamic scope*.

Strongly typed

Functional programs are often strongly typed, meaning that types are set no later than at *compile-time*. F# does have the ability to perform runtime type assertion, but for most parts it relies on explicit *type annotations* and *type inference* at compile-time. This means that type errors are caught at compile time instead of at runtime.

Lazy evaluation

Due to referential transparency, values can be computed any time up until the point when it is needed. Hence, they need not be computed at compilation time, which allows for infinite data structures. F# has support for lazy evaluations using the `lazy`-keyword, sequences using the `seq`-type, and computation expressions, all of which are advanced topics and not treated in this book.

Immutable states imply that data structures in functional programming are different than in imperative programming. E.g., in F# lists are immutable, so if an element of a list is to be changed, a new list must be created by copying all old values except that which is to be changed. Such an operation is therefore linear in computational complexity. In contrast, arrays are mutable values, and changing a value is done by reference to the value's position and changing the value at that location. This has constant computational complexity. While fast, mutable values give dynamic scope and makes reasoning about the correctness of a program harder, since mutable states do not have referential transparency.

Functional programming may be considered a subset of *imperative programming*, in the sense that functional programming does not include the concept of a state, or one may think of functional programming as only having one unchanging state. Functional programming also has a bigger focus on declaring rules for *what* should be solved, and not explicitly listing statements describing *how* these rules should be combined and executed in order to solve a given problem. Functional programming is often found to be less error-prone at runtime, making more stable, safer programs that are less open for, e.g., hacking.