Exploring Languages with Interpreters and Functional Programming Chapter 6

H. Conrad Cunningham

02 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

6 Procedural Abstraction

6.1 Chapter Introduction

Chapter 2 introduced the concepts of procedural and data abstraction. This chapter (6) focuses on procedural abstraction. Chapter 7focuses on data abstraction.

The goals of this chapter are to:

- illustrate use of procedural abstraction, in particular of the top-down, stepwise refinement approach to design
- introduce modular programming using Haskell modules

6.2 Procedural Abstraction Review

As defined in Chapter 2, *procedural abstraction* is the separation of the logical properties of an *action* from the details of how the action is implemented.

In general, we abstract an action into a Haskell function that takes zero or more arguments and returns a value but does not have other effects. In later chapters, we discuss how input, output, and other effects are handled in a purely functional manner. (For example, in Chapter 10 we examine simple input and output.)

We also collect one or more functions into a Haskell module with appropriate type definitions, data structures, and local functions. We can explicitly expose some of the features and hide others.

To illustrate the development of a group of related Haskell procedural abstractions in this chapter, we use top-down stepwise refinement.

6.3 Top-Down Stepwise Refinement

A useful and intuitive design process for a small program is to begin with a high-level solution and incrementally fill in the details. We call this process top-down stepwise refinement. Here we introduce it with an example.

6.3.1 Developing a square root package

Consider the problem of computing the nonnegative square root of a nonnegative number x. Mathematically, we want to find the number y such that

$$y \geq 0 \text{ and } y^2 = x.$$

A common algorithm in mathematics for computing the above y is to use Newton's method of successive approximations, which has the following steps for square root:

1. Guess at the value of y.

2. If the current approximation (guess) is sufficiently close (i.e., good enough), return it and stop; otherwise, continue.
3. Compute an improved guess by averaging the value of the guess y and x/y, then go back to step 2.

To encode this algorithm in Haskell, we work top down to decompose the problem into smaller parts until each part can be solved easily. We begin this *top-down stepwise refinement* by defining a function with the type signature:

```
sqrtIter :: Double -> Double -> Double
```

We choose type `Double` (double precision floating point) to approximate the real numbers. Thus we can encode step 2 of the above algorithm as the following Haskell function definition:

```
sqrtIter guess x                                    -- step 2
    | goodEnough guess x = guess
    | otherwise          = sqrtIter (improve guess x) x
```

We define function `sqrtIter` to take two arguments—the current approximation `guess` and nonnegative number `x` for which we need the square root. We have two cases:

- When the current approximation `guess` is sufficiently close to `x`, we return `guess`.

  We abstract this decision into a separate function `goodEnough` with type signature:

  ```
  goodEnough :: Double -> Double -> Bool
  ```

- When the approximation is not yet close enough, we continue by reducing the problem to the application of `sqrtIter` itself to an improved approximation.

  We abstract the improvement process into a separate function `improve` with type signature:

  ```
  improve :: Double -> Double -> Double
  ```

  To ensure termination of `sqrtIter`, the argument `(improve guess x)` on the recursive call must get closer to termination (i.e., to a value that satisfies its base case).

The function `improve` takes the current `guess` and `x` and carries out step 3 of the algorithm, thus averaging `guess` and `x/guess`, as follows:

```
improve :: Double -> Double -> Double      -- step 3
improve guess x = average guess (x/guess)
```

Function application `improve y x` assumes `x >= 0 && y > 0`. We call this a *precondition* of the `improve y x` function.

Because of the precondition of `improve`, we need to strengthen the precondition of `sqrtIter guess x` to `x >= 0 && guess > 0`.

In `improve`, we abstract `average` into a separate function as follows:

```
average :: Double -> Double -> Double
average x y = (x + y) / 2
```

The new guess is closer to the square root than the previous guess. Thus the algorithm will terminate assuming a good choice for function `goodEnough`, which guards the base case of the `sqrtIter` recursion.

How should we define `goodEnough`? Given that we are working with the limited precision of computer floating point arithmetic, it is not easy to choose an appropriate test for all situations. Here we simplify this and use a tolerance of 0.001.

We thus postulate the following definition for `goodEnough`:

```
goodEnough :: Double -> Double -> Bool
goodEnough guess x = abs (square guess - x) < 0.001
```

In the above, `abs` is the built-in absolute value function defined in the standard Prelude library. We define square as the following simple function (but could replace it by just `guess * guess`):

```
square :: Double -> Double
square x = x * x
```

What is a good initial guess? It is sufficient to just use 1. So we can define an overall square root function `sqrt'` as follows:

```
sqrt' :: Double -> Double
sqrt' x | x >= 0 = sqrtIter 1 x
```

(A square root function `sqrt` is defined in the Prelude library, so a different name is needed to avoid the name clash.)

Function `sqrt' x` has precondition `x >= 0`. This and the choice of `1` for the initial guess ensure that functions `sqrtIter` and `improve` are applied with arguments that satisfy their preconditions.

6.3.2 Making the package a Haskell module

We can make this package into a Haskell module by putting the definitions in a file (e.g., named `Sqrt`) and adding a module header at the beginning as follows:

```
module Sqrt
    ( sqrt' )
where
    -- give the definitions above for functions sqrt',
    --   sqrtIter, improve, average, and goodEnough
```

The header gives the module the name `Sqrt` and lists the names of the features being *exported* in the parenthesis that follows the name. In this case, only function `sqrt'` is exported.

Other Haskell modules that *import* the `Sqrt` module can access the features named in its export list. In the case of `Sqrt`, the other functions—`sqrtIter`, `goodEnough`, and `improve`)— are local to (i.e., hidden inside) the module.

In this book, we often call the exported features (e.g., functions and types) the module's *public* features and the ones not exported the *private* features.

We can import module `Sqrt` into a module such as module `TestSqrt` shown below. By default, the `import` makes all the definitions exported by `Sqrt` available within module `TestSqrt`. The importing module may select the features it wishes to export and may assign local names to the features it does import.

```
module TestSqrt
where

import Sqrt -- file Sqrt.hs, import all public names

main = do
    putStrLn (show (sqrt' 16))
    putStrLn (show (sqrt' 2))
```

In the above Haskell code, the symbol "`--`" denotes the beginning of an end-of-line comment. All text after that symbol is text ignored by the Haskell compiler.

The Haskell module for the Square root case study is in file `Sqrt.hs`. Limited, smoke-testing code is in file `SqrtTest.hs`.

6.3.3 Reviewing top-down stepwise refinement

The program design strategy known as *top-down stepwise refinement* is a relatively intuitive design process that has long been applied in the design of structured programs in imperative procedural languages. It is also useful in the functional setting.

In Haskell, we can apply top-down stepwise refinement as follows.

1. Start with a high-level solution to the problem consisting of one or more functions. For each function, identify its type signature and functional requirements (i.e., its inputs, outputs, and termination condition).

   Some parts of each function may be incomplete—expressed as "pseudocode" expressions or as-yet-undefined functions.

2. Choose one of the incomplete parts. Consider the specified type signature and functional requirements. Refine the incomplete part by breaking it into subparts or, if simple, defining it directly in terms of Haskell

expressions (including calls to the Prelude, other available library functions, or previously defined parts of the algorithm).

When refining an incomplete part, consider the various options according to the relevant design criteria (e.g., time, space, generality, understandability, and elegance).

The refinement of the function may require a refinement of the data being passed.

If it not possible to design an appropriate function or data refinement, back up in the refinement process and readdress previous design decisions.

3. Continue step 2 until all parts are fully defined in terms of Haskell code and data and the resulting set of functions meets all required criteria.

For as long as possible, we should stay with terminology and notation that is close to the problem being solved. We can do this by choosing appropriate function names and signatures and data types. (In other chapters, we examine Haskell's rich set of builtin and user-defined types.)

For stepwise refinement to work well, we must be willing to back up to earlier design decisions when appropriate. We should keep good documentation of the intermediate design steps.

The stepwise refinement method can work well for small programs , but it may not scale well to large, long-lived, general purpose programs. In particular, stepwise refinement may lead to a module structure in which modules are tightly coupled and not robust with respect to changes in requirements.

A combination of techniques may be needed to develop larger software systems. In the next section (6.4), we consider the use of modular design techniques.

6.4 Modular Design and Programming

In the previous section, we developed a Haskell module. In this section, let's consider what a module is more generally.

Software engineering pioneer David Parnas defines a *module* as "a work assignment given to a programmer or group of programmers" [17]. This is a *software engineering* view of a module.

In a programming language like Haskell, a `module` is also a program unit defined with a construct or convention. This is a *programming language* view of a module.

In a programming language, each module may be stored in a separate file in the computer's file system. It may also be the smallest external unit processed by the language's compiler or interpreter.

Ideally, a language's module features should support the software engineering module methods.

6.4.1 Information-hiding modules and secrets

According to Parnas, the goals of *modular design* are to [14]:

1. enable programmers to understand the system by focusing on one module at a time (i.e., *comprehensibility*).
2. shorten development time by minimizing required communication among groups (i.e., *independent development*).
3. make the software system flexible by limiting the number of modules affected by significant changes (i.e., *changeability*).

Parnas advocates the use of a principle he called *information hiding* to guide decomposition of a system into appropriate modules (i.e., work assignments). He points out that the connections among the modules should have as few information requirements as possible [14].

In the Parnas approach, an information-hiding module:

- forms a *cohesive* unit of functionality *separate* from other modules
- *hides* a design decision—its *secret*—from other modules
- *encapsulates* an aspect of system likely to change (its secret)

Aspects likely to change independently of each other should become secrets of separate modules. Aspects unlikely to change can become interactions (connections) among modules.

This approach supports the goal of changeability (goal 2). When care is taken to design the modules as clean abstractions with well-defined and documented interfaces, the approach also supports the goals of independent development (goal 1) and comprehensibility (goal 3).

Information hiding has been absorbed into the dogma of contemporary object-oriented programming. However, information hiding is often oversimplified as merely hiding the data and their representations [19].

The secret of a well-designed module may be much more than that. It may include such knowledge as a specific functional requirement stated in the requirements document, the processing algorithm used, the nature of external devices accessed, or even the presence or absence of other modules or programs in the system [14–16]. These are important aspects that may change as the system evolves.

Secret of square root module The secret of the `Sqrt` module in the previous section is the algorithm for computing the square root.

6.4.2 Contracts: Preconditions and postconditions

Now let's consider the semantics (meaning) of functions.

The *precondition* of a function is what the caller (i.e., the client of the function) must ensure holds when calling the function. A precondition may specify the valid combinations of values of the arguments. It may also record any constraints on any "global" state that the function accesses or modifies.

If the precondition holds, the supplier (i.e., developer) of the function must ensure that the function terminates with the *postcondition* satisfied. That is, the function returns the required values and/or alters the "global" state in the required manner.

We sometimes call the set of preconditions and postconditions for a function the *contract* for that function.

Contracts of square root module In the `Sqrt` module defined in the previous section, the exported function `sqrt' x` has the precondition:

```
x >= 0
```

Function `sqrt' x` is undefined for `x < 0`.

The postcondition of the function `sqrt' x` function is that the result returned is the correct mathematical value of the square root within the allowed tolerance. That is, for a tolerance of 0.001:

```
(sqrt x - 0.001)^2 < (sqrt x)^2 < (sqrt x + 0.001)^2
```

We can state preconditions and postconditions for the local functions `sqrtIter`, `improve`, `average`, and `goodEnough` in the `Sqrt` module. These are left as exercises.

The preconditions for functions `average` and `goodEnough` are just the assertion `True` (i.e., always satisfied).

Contracts of `Factorial` module Consider the factorial functions defined in Chapter 4. (These are in the source file `Factorial.hs`.)

What are the preconditions and postconditions?

Functions `fact1`, `fact2`, and `fact3` require that argument `n` be a natural number (i.e., nonnegative integer) value. If they are applied to a negative value for `n`, then the evaluation does not terminate. Operationally, they go into an "infinite loop" and likely will abort when the runtime stack overflows.

If function `fact4` is called with a negative argument, then all guards and pattern matches fail. Thus the function aborts with a standard error message.

Similarly, function `fact4'` terminates with a custom error message for negative arguments.

Thus to ensure normal termination, we impose the precondition

```
n >= 0
```

on all these factorial functions.

The postcondition of all six factorial functions is that the result returned is the correct mathematical value of `n` factorial. For `fact4`, that is:

```
fact4 n = fact'(n)
```

None of the six factorial functions access or modify any global data structures, so we do not include other items in the precondition or postcondition.

Function `fact5` is defined to be 1 for all arguments less than zero. So, if this is the desired result, we can weaken the precondition to allow all integer values, for example,

```
True
```

and strengthen the postcondition to give the results for negative arguments, for example:

```
fact5 n = if n >= 0 then fact'(n) else 1
```

Caveat: In this chapter, we ignore the limitations on the value of the factorial functions' argument `n` imposed by the finite precision of the computer's integer arithmetic. We readdress this issue somewhat in Chapter 12.

6.4.3 Interfaces for modules

It is important for an information-hiding module to have a well-defined and stable interface. What do we mean by interface?

According to Britton et al [2], an *interface* is a "set of assumptions ... each programmer needs to make about the other program ... to demonstrate the correctness of his own program".

A module interface includes the type signatures (i.e., names, arguments, and return values), preconditions, and postconditions of all public operations (e.g., functions).

As we see in Chapter 7, the interface also includes the *invariant* properties of the data values and structures manipulated by the module and the definitions of any new data types exported by the module. An invariant must be part of the precondition of public operations except operations that construct elements of the data type (i.e., constructors). It must also be part of the postcondition of public operations except operations that destroy elements of the data type (i.e., destructors).

As we have seen, in Haskell the `module` not provide direct syntactic or semantic support for preconditions, postconditions, or invariant assertions.

Interface of square root module The interface to the `Sqrt` module in the previous section consists of the function signature:

```
sqrt' :: Double -> Double
```

where `sqrt' x` has the precondition and postcondition defined above. None of the other functions are accessible outside the module `Sqrt` and, hence, are not part of the interface.

6.4.4 Abstract interfaces for modules

An *abstract interface* is an interface that does not change when one module implementation is substituted for another [2,17]. It concentrates on module's essential aspects and obscures incidental aspects that vary among implementations.

Information-hiding modules and abstract interfaces enable us to design and build software systems with multiple versions. The information-hiding approach seeks to identify aspects of a software design that might change from one version to another and to hide them within independent modules behind well-defined abstract interfaces.

We can reuse the software design across several similar systems. We can reuse an existing module implementation when appropriate. When we need a new implementation, we can create one by following the specification of the module's abstract interface.

Abstract interface of square root module For the `Sqrt` example, if we implemented a different module with the same interface (signatures, preconditions, postconditions, etc.), then we could substitute the new module for `Sqrt` and get the same result.

In this case, the interface is an abstract interface for the set of module implementations.

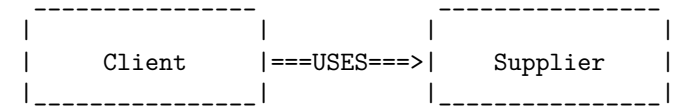
Caveats: Of course, the time and space performance of the alternative modules might differ. Also, because of the nature of floating point arithmetic, it may be difficult to ensure both algorithms have precisely the same termination conditions.

6.4.5 Client-supplier relationship

The design and implementation of information-hiding modules should be approached from two points of view simultaneously:

supplier: the developers of the module—the providers of the services
client: the users of the module—the users of the services (e.g., the designers of other modules)

The *client-supplier relationship* is as represented in the following diagram:



```
(module user)                    (module)
```

The supplier's concerns include:

- efficient and reliable algorithms and data structures
- convenient implementation
- easy maintenance

The clients' concerns include:

- accomplishing their own tasks
- using the supplier module without effort to understand its internal details
- having a sufficient, but not overwhelming, set of operations.

As we have noted previously, the *interface* of a module is the set of features (i.e., public operations) provided by a supplier to clients.

A precise description of a supplier's interface forms a *contract* between clients and supplier.

The client-supplier contract:

1. gives the responsibilities of the client

   These are the conditions under which the supplier must deliver results—when the *preconditions* of the operations are satisfied (i.e., the operations are called correctly).

2. gives the responsibilities of the supplier

   These are the benefits the supplier must deliver—make the *postconditions* hold at the end of the operation (i.e., the operations deliver the correct results).

The contract

- protects the client by specifying how much must be done by the supplier
- protects the supplier by specifying how little is acceptable to the client

If we are both the clients and suppliers in a design situation, we should consciously attempt to separate the two different areas of concern, switching back and forth between our supplier and client "hats".

6.4.6 Design criteria for interfaces

What else should we consider in designing a good interface for an information-hiding module?

In designing an interface for a module, we should also consider the following criteria. Of course, some of these criteria conflict with one another; a designer must carefully balance the criteria to achieve a good interface design.

Note: These are general principles; they are not limited to Haskell or functional programming. In object-oriented languages, these criteria apply to class interfaces.

- **Cohesion:** All operations must logically fit together to support a single, coherent purpose. The module should describe a single abstraction.
- **Simplicity:** Avoid needless features. The smaller the interface the easier it is to use the module.
- **No redundancy:** Avoid offering the same service in more than one way; eliminate redundant features.
- **Atomicity:** Do not combine several operations if they are needed individually. Keep independent features separate. All operations should be *primitive*, that is, not be decomposable into other operations also in the public interface.
- **Completeness:** All primitive operations that make sense for the abstraction should be supported by the module.
- **Consistency:** Provide a set of operations that are internally consistent in
  - naming convention (e.g.,, in use of prefixes like "set" or "get", in capitalization, in use of verbs/nouns/adjectives),
  - use of arguments and return values (e.g.,, order and type of arguments),
  - behavior (i.e., make operations work similarly).

  Avoid surprises and misunderstandings. Consistent interfaces make it easier to understand the rest of a system if part of it is already known.

  The operations should be consistent with good practices for the specific language being used.
- **Reusability:** Do not customize modules to specific clients, but make them general enough to be reusable in other contexts.
- **Robustness with respect to modifications:** Design the interface of an module so that it remains stable even if the implementation of the module changes. (That is, it should be an abstract interface for an information-hiding module as we discussed above.)
- **Convenience:** Where appropriate, provide additional operations (e.g.,, beyond the complete primitive set) for the convenience of users of the module. Add convenience operations only for frequently used combinations after careful study.

We must trade off conflicts among the criteria. For example, we must balance:

- completeness versus simplicity
- reusability versus simplicity
- convenience versus consistency, simplicity, no redundancy, and atomicity

We must also balance these design criteria against efficiency and functionality.

6.5 What Next?

In this chapter (6), we considered procedural abstraction and modularity in that context.

In Chapter 7, we consider data abstraction and modularity in that context.

6.6 Chapter Source Code

The Haskell source code for this chapter are in files:

- `Sqrt.hs` for the Square Root case study
- `SqrtTest.hs` for (limited) "smoke testing" of the `Sqrt` module
- `Factorial.hs` for the factorial source code from Chapter 4
- `TestFactorial.hs` is an extensive testing module developed in Chapter 12 for the factorial module

6.7 Exercises

1. State preconditions and postconditions for the following internal functions in the `Sqrt` module:
   a. `sqrtIter`
   b. `improve`
   c. `average`
   d. `goodEnough`
   e. `square`
2. Develop recursive and iterative (looping) versions of the square root function from this chapter in one or more primarily imperative languages (e.g., Java, C++, C#, Python 3, or Lua)

6.8 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work from my previous materials:

- Using Top-Down Stepwise Refinement (square root module), which is based on Section 1.1.7 of Abelson and Sussman's *Structure and Interpretation of*

Computer Programs [1] and my example implementations of this algorithm in Scala, Elixir, and Lua as well as Haskell.

- Modular Design and Programming from my Data Abstraction [4] and Modular Design [5] notes, which drew ideas over the past 25 years from a variety of sources [2,3,6–16,18,19].

In 2017, I continued to develop this work as sections 2.5-2.7 in Chapter 2, Basic Haskell Functional Programming), of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Basic Haskell Functional Programming chapter into four chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 2.1-2.3 became the basis for new Chapter 4, First Haskell Programs; previous Section 2.4 became Section 5.3 in the new Chapter 5, Types; and previous sections 2.5-2.7 were reorganized into new Chapter 6, Procedural Abstraction (this chapter), and Chapter 7, Data Abstraction. The discussion of contracts for the factorial functions was moved from the 2017 Evaluation and Efficiency chapter to this chapter.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), adding cross-references, and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

6.9 Terms and Concepts

TODO: Update

Procedural abstraction, top-down stepwise refinement, abstract code, termination condition for recursion, Newton's method, Haskell `module`, module exports and imports, information hiding, module secret, encapsulation, precondition, postcondtion, contract, invariant, interface, abstract interface, design criteria for interfaces, software reuse, use of Haskell modules to implement information-hiding modules, client-supplier contract.

6.10 References

[1] Harold Abelson and Gerald Jockay Sussman. 1996. *Structure and interpretation of computer programs (SICP)* (Second ed.). MIT Press, Cambridge, Massachusetts, USA. Retrieved from https://mitpress.mit.edu/sicp/

[2] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. 1981. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th international conference on software engineering*, IEEE, San Diego, California, USA, 195–204.

[3] Timothy Budd. 2000. *Understanding object-oriented programming with Java* (Updated ed.). Addison-Wesley, Boston, Massachusetts, USA.

[4] H. Conrad Cunningham. 2019. *Notes on data abstraction.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/DataAbstraction/DataAbstraction.html

[5] H. Conrad Cunningham. 2019. *Notes on modular design.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ModularDesign/ModularDesign.html

[6] H. Conrad Cunningham, Yi Liu, and Jingyi Wang. 2010. Designing a flexible framework for a table abstraction. In *Data engineering: Mining, information, and intelligence*, Yupo Chan, John Talburt and Terry M. Talley (eds.). Springer, New York, New York, USA, 279–314.

[7] H. Conrad Cunningham and Jingyi Wang. 2001. Building a layered framework for the table abstraction. In *Proceedings of the ACM symposium on applied computing*, Las Vegas, Nevada, USA.

[8] H. Conrad Cunningham, Cuihua Zhang, and Yi Liu. 2004. Keeping secrets within a family: Rediscovering Parnas. In *Proceedings of the international conference on software engineering research and practice (SERP)*, CSREA Press, Las Vegas, Nevada, USA, 712–718.

[9] Nell Dale and Henry M. Walker. 1996. *Abstract data types: Specifications, implementations, and applications.* D. C. Heath, Lexington, Massachusetts, USA.

[10] Cay S. Horstmann. 1995. *Mastering object-oriented design in C++.* Wiley, Indianapolis, Indiana, USA.

[11] Cay S. Horstmann and Gary Cornell. 1999. *Core Java 1.2: Volume I—Fundamentals.* Prentice Hall, Englewood Cliffs, New Jersey, USA.

[12] Bertrand Meyer. 1997. *Object-oriented program construction* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[13] Hanspeter Mossenbock. 1995. *Object-oriented programming in Oberon-2.* Springer, Berlin, Germany.

[14] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (December 1972), 1053–1058.

[15] David L. Parnas. 1979. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering* SE-5, 1 (1979), 128–138.

[16] David L. Parnas. 1979. The modular structure of complex systems. *IEEE Transactions on Software Engineering* SE-11, 13 (1979), 128–138.

[17] David L. Parnas. 2001. Some software engineering principles. In *Software fundamentals: Collected papers by David L. Parnas*, Daneil M. Hoffman and David M. Weiss (eds.). Addison-Wesley, Boston, Massachusetts, USA.

[18] Pete Thomas and Ray Weedom. 1995. *Object-oriented programming in Eiffel*. Addison-Wesley, Boston, Massachusetts, USA.

[19] David M. Weiss. 2001. Introduction: On the criteria to be used in decomposing systems into modules. In *Software fundamentals: Collected papers by David L. Parnas*, Daniel M. Hoffman and David M. Weiss (eds.). Addison-Wesley, Boston, Massachusetts, USA.