Principles of Software Development

— an introduction for computational scientists —

Daniel Vedder

October 2019

Center for Computational and Theoretical Biology,
University of Würzburg

Contents

1. Principles of Software Development **4**
- 1.1. What to aim for . . . 4
- 1.2. How to get there . . . 5
- 1.3. Going on from here . . . 5

2. Understandable Software **7**
- 2.1. Layout . . . 7
- 2.2. Structure . . . 8
- 2.3. Documentation . . . 8

3. The Art of Abstraction **10**
- 3.1. Encapsulation . . . 10
- 3.2. Abstraction . . . 11
- 3.3. Modularisation . . . 12

4. Dealing with Errors **15**
- 4.1. Finding Errors . . . 15
- 4.2. Preventing Errors . . . 16
- 4.3. Anticipating Errors . . . 17

5. Programming tools: languages **18**
- 5.1. Fundamental differences between languages . . . 18
- 5.2. Some languages to choose from . . . 19
- 5.3. Working with libraries . . . 21

6. Programming tools: paradigms **22**
- 6.1. Procedural Programming . . . 22
- 6.2. Object Oriented Programming . . . 22
- 6.3. Functional Programming . . . 23
- 6.4. Conclusion . . . 24

7. Developing in a team **25**
- 7.1. Exchanging code: version control . . . 25
- 7.2. Improving code: collaborative development . . . 26
  - 7.2.1. Pair programming . . . 26
  - 7.2.2. Code inspections . . . 26

8. Some final thoughts on programming **28**
8.1. Optimisation 28
8.2. Licenses 29
8.3. Going on from here 30

A. Code Inspection Checklist **31**

B. Bibliography **32**

All chapters originally published on https://terranostra.one.

1. Principles of Software Development

Working at an institute for computational biology, my colleagues and I deal with computer code almost every day. Yet none of us is a trained software developer as such—we are biologists, physicists, and mathematicians who happen to have learnt a bit of programming on the side. Some of us (like myself) got into it as a hobby, most picked it up along the way. So over the past few weeks, I started to ask myself a question: "How can we become better at developing software?" The next question came naturally: "Well, what is good software development?"

I've been reading and thinking a lot about that since then. By now, I have seven years of programming experience to go on, but I have never thought through my craft in any systematic manner. So this is a bit of a voyage of discovery: drawing together shreds of experience and combining them with what I have learnt from others or read in blogs and books. It's fun, it's exciting, it's educational. And because writing is an excellent form of learning, I thought I would document my journey in a series of posts here on Terra Nostra.

So let's get started with the first part: "What, in essence, are the most important principles of software development?"

1.1. What to aim for

Elegance is something both artists and engineers aspire to. Since software development is both an art and a science, elegance often crops up when we talk about good code. Elegance has the connotation of efficient simplicity, of beauty paired with power. Describing something as "elegant code" is a high praise.

Nonetheless, it remains an elusive concept. It is hard to describe and even harder to quantify. Therefore, I prefer to talk about three qualities that are somewhat more concrete. You might call this the golden aim of software development: software that is *reliable, understandable,* and *extendable.*

Reliability. Software has to work, and it has to work correctly. Having some bugs is virtually unavoidable, but a single bad bug can render an entire program worthless. This is especially true in scientific computing, where much depends on the validity of our calculations. A bug in the wrong place can set research back by 15 years[1]. Beyond "mere" correctness, reliability also means that a software holds up well under stress and can deal with unexpected input, high computational loads, or a different system setup.

Understandability. "Programs must be written for people to read, and only incidentally for machines to execute", wrote Abelson & Sussman in the introduction to SICP[2]. If you've ever come back to one of your projects after having left it alone for a couple of months, you'll know how grateful one is for cleanly-written code. If you've ever had to come to grips with somebody else's codebase, that goes double. Regardless of whether you're debugging or extending a given piece of software—a good understanding of it is a necessary prerequisite to whatever you plan to do. When writing code, be kind to the person coming after you. (It might be you.)

Extensibility. A useful piece of software will frequently find itself having to adapt to new requirements. Whether it is support for new hardware in some imaging software, or a new factor to be included in a simulation model, living software grows over time. A well thought-out architecture is decisive in determining how hard or easy the implementation of new additions will be. Apart from making life easier for the original authors, software that is easily extendable is also more likely to be taken up and improved by others.

1.2. How to get there

There's many different aspects to achieving the three goals outlined above. Many of them play together, there is no one way to achieve perfection. Here, however, are two principles that crop up again and again in various forms. I'll be writing more about each of these some other time, but I did want to briefly mention them up front:

Reducing complexity. "Reducing complexity is arguably the most important key to being an effective programmer", says Steve McConnell (see below). Programming is complicated business—the bigger the program, the more complicated it becomes. Keeping this complexity at a level manageable by the human brain is a serious skill. But it pays off: less complex programs are more reliable, easier to understand, and easier to extend. In fact, one could call this the golden way of software development.

Using the right tools. Over the past seven decades, generations of computer scientists have invented countless tools to help us write better programs. From languages and libraries to paradigms and data structures, knowing the tools at your disposal and when to use which will make you faster, more efficient, and more effective.

1.3. Going on from here

Thus far with a very brief overview of some of the core principles of software development. In the coming weeks (and perhaps months) I will go more into detail on the things I have touched on above. Don't ask me yet what I'm going to write, I'm still figuring that out... But I look forward to it.

One last thought before I close. Becoming a better programmer is exactly that: a becoming, a process, a journey without end. The path, though, is simple: read, write, repeat. Read what other people have to say; in discussions, in books, in their source code. Write plenty of code of your own. And then start over.

2. Understandable Software

Having begun our series on software development with a broad look at the basic principles, let us now get down to the nitty-gritty. We said the three key aims of a developer should be software that is reliable, understandable, and extendable. As the second of these is probably the easiest, let us start with that. So how do you write software that is easy to understand?

Well, really it's very simple. All it takes is some good habits in three areas: code layout, code structure, and code documentation. Let's have a look.

2.1. Layout

Code layout is all about the visual appearance of your code on your screen. Hard-to-read code is often obvious at first glance. Likewise, eminently readable code has a certain elegance to its structure that is visible even before one has actually read it.

The most important thing here is proper indentation. (This won't be news to anybody who's been programming for a while, but I regularly have to point it out to students in our programming classes, so I'm including it anyway.) Indenting each line of code so that it is one tab (three or four spaces) further in than the opening line of its block makes a world of difference to legibility. That way, the reader knows at a glance what block the current line belongs to, like in this Java example:

```
// The function header is on the top level
public void longMultiply(int a, int b)
{
    int result = a; // now we are inside the function block
    while (b > 0) {
        result = result + a; // and this is the inner loop block
        b = b - 1;
    }
    return result; // now we're back to the function block
}
```

It gets a bit more complicated when we come to the subject of line breaks, such as when you have really long and complicated conditions in an if-clause. Here, you can use a combination of parentheses and line breaks to show which expressions belong together. Fortunately, almost all modern IDEs and editors have good indentation behaviour built in by default, so this really isn't hard. The key idea to remember is that layout should show logic.

Apart from indentation, there are a few other things to keep in mind. One is to limit the line length. Some projects have a strict line length limit of (traditionally) 80 characters. You don't need to be religious about this, but lines longer than 100 characters do become a lot harder to read.

Also, be liberal in your use of white space (spaces, tabs, and empty lines). Dense code is tough to wade through, so give your reader some breathing space. Of course, one can also overdo it, but you'll learn to find a good measure.

2.2. Structure

The principle way of structuring code is the use of functions. For some reason I have yet to understand, many biologists seem to be reluctant to put functions in their code—to their own detriment.

Functions are wonderful things. They enable code reuse, so you don't have to type that list of commands over for each analysis you do—you simply stick it in a function and call the function. Plus, they aid legibility. What's easier to understand in a source file, eight lines of gobble-de-gook R code or a single call to `doANOVA()`?

Functions are most useful when you keep them short. They help reduce complexity by breaking down a long program into small, manageable chunks that are easy to understand. If you need to scroll to finish reading your function, you've basically lost that advantage. An ideal length is probably no more than 20 lines. Short functions are easy to read, easy to understand, easy to test. So break up that 70-line-monster.

If you find your project growing well beyond the 500 lines-of-code mark, you should also seriously start to think about splitting it up into multiple files. Moving around inside very long files is cumbersome, and remembering what is where becomes increasingly more difficult. Better to group related functions in separate files. This makes it much easier to develop a mental model of the whole program, and finding code becomes quicker, too.

2.3. Documentation

Documenting your project sounds like a lot of work, but actually, it starts with the small things. Proper variable names, for instance. Calling your integer `no_carnivores` instead of `nc`, or using expressive function names like `competeHerbivores()` is really helpful. In a perfect-world ideal scenario, you won't even need comments, because your code will be so clean and readable that it's self-documenting code.

Of course, we don't live in a perfect world, so comments are necessary nonetheless. At the minimum, each file, class, and function ought to have a brief header comment describing what it does. (Some trivial functions can be skipped, but that should definitely be the exception.) You can also use comments as section headers, dividing up the functions in your files or the statements in your functions into subgroups. In some cases, you may also have some difficult code that needs to be explained in plain English. However, you

should first try to see whether you can rewrite the relevant section instead, so that it doesn't need to be commented anymore. (Usually, it is possible...)

Remember that comments should state intent, not mechanism. I can see what the program does by reading the code, I don't need the comment to repeat that for me. What the comment should tell me is why the code does what it does. I can see that the code checks whether `ttl == 0`, what I need to know is that this is used to avoid an infinity loop when an animal is searching for a free spot on a full landscape.

Comments are great when one is trawling through the source code itself, but when you first encounter a software project, you first want to get a higher-level perspective. Good practice is to include a text file called README in the top project directory. This can either be a brief explanation of the project in itself, or contain pointers to the rest of the documentation. Another good tradition in the Open Source world is to include a second document called HACKING (often in a folder called doc), with slightly more detailed information about the architecture and conventions of your software. (The bigger your project, the more useful this becomes.)

Lastly, having online/HTML documentation can be a big bonus. Many languages have libraries for generating this kind of documentation from the comments in your source code. (Some, like Java's javadoc, are already built in.) This will give you fancy-looking, interactive doc pages detailing every function in every file of your project—at virtually no extra cost to you. After all, you're already commenting your functions, aren't you?

3. The Art of Abstraction

"Software's Primary Imperative has to be managing complexity", says Steve McConnell in his book on software construction. In the first article of this series, I already said that reducing complexity makes software simultaneously more reliable, understandable, and extendable. Now, we are going to take a look at how that is possible.

The key concepts here are encapsulation, abstraction, and modularisation. Unfortunately, these terms are used semi-interchangeably in the literature, even though they have slightly different connotations. For the purposes of this article, I'm going to refer to encapsulation as the overarching principle, with the other two as more specific applications of the basic idea. Let's look at each in turn.

3.1. Encapsulation

The shown diagram (fig. 3.1) and the accompanying quote from the Zen of Python[3] illustrate what encapsulation does. Encapsulation is all about taking a complicated system (center) and splitting it up into a group of maximally self-contained subsystems (right).

There are two main reasons for doing this:

1. Each subsystem can be designed, developed, and debugged in isolation. You don't have to keep the entire program in mind while working on any one part.
2. Subsystems can be considered as "black boxes": as long as you know what they do, how they do it is irrelevant. This means you can think about the entire program

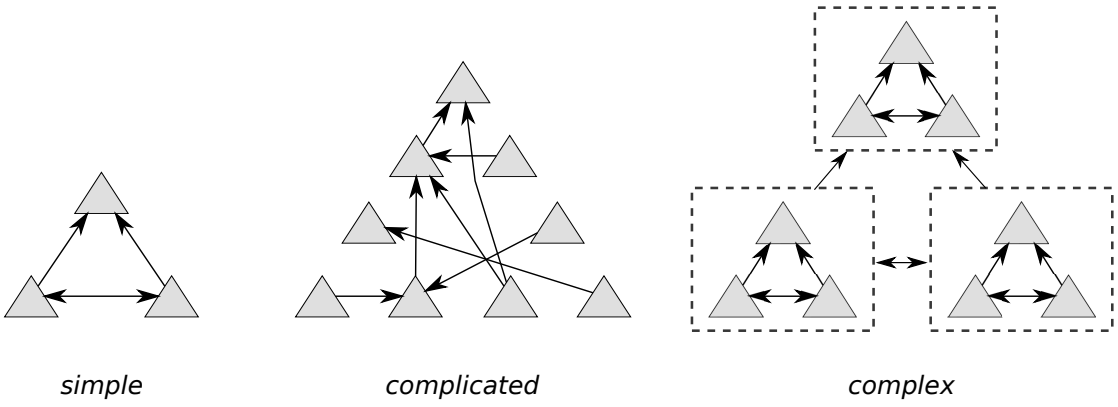

Figure 3.1.: "Simple is better than complex. Complex is better than complicated."

without having to keep in mind all the details of each individual component—it helps you see the big picture.

It should be fairly obvious how this makes thinking about a large system much easier. It is also worth noting that encapsulation can take place at multiple levels, such as the function, the class, or the package.

(Another positive side effect of encapsulation is that it makes code reuse easier. Ideally, you may even be able to take an old subsystem/class/function and copy-paste it straight into a new project.)

So how do you go about splitting up a system?

3.2. Abstraction

The first way is to cut it up into layers. This "vertical encapsulation" is usually referred to as abstraction.

It is something we do every day. Consider the sentence "Flocks of pigeons flew over the city." This statement treats a flock as a single object (although it's really composed of hundreds of individual birds), as it does the city (even though this too is composed of hundreds of buildings). The word "flock" abstracts the concept of a group of birds into a single term, making it easier to talk about. If you're a molecular biologist, the word "bird" may in itself already be an abstraction for a conglomerate of different organs, tissues, and cells—and so forth back to first principles.

But it is rather cumbersome to talk about "a loose association of avian organisms in powered aerial motion over a municipal collection of buildings". In natural language, as in programming, you want to be using an appropriate degree of abstraction. Programming is about solving problems, and problems are easiest to think and write about if you do so at the highest possible level of abstraction. Therefore, abstraction in programming is all about hiding the implementation details.

Consider the following diagram of the functions in a simple graphics library (fig. 3.2).

I wrote this library a year ago to help some colleagues of mine who were using LED panels to investigate butterfly navigation behaviour. The library itself (inside the bounded box) consists of a collection of shape functions (upper section), which are implemented using two core drawing functions (lower section). Most user code only has to use the extended functions, and the extended functions only ever call the core functions.

This is good for users of the library, because they can simply combine shape objects, and don't have to worry about the nitty-gritty, like how to draw a smooth circle on a gridded screen. But even better, this design makes the library highly portable. If you need to support a new type of display, literally the only thing you have to change are the core functions—because they are the only parts of the code that need to know anything about the hardware. Everything else will run just as it always has.

(In fact, I developed this library with a character-based screen, added support for the LED panels, ported it from Python to Common Lisp[4], and have just been asked to adapt it to a new generation of LED panels. Abstraction made it easy.)

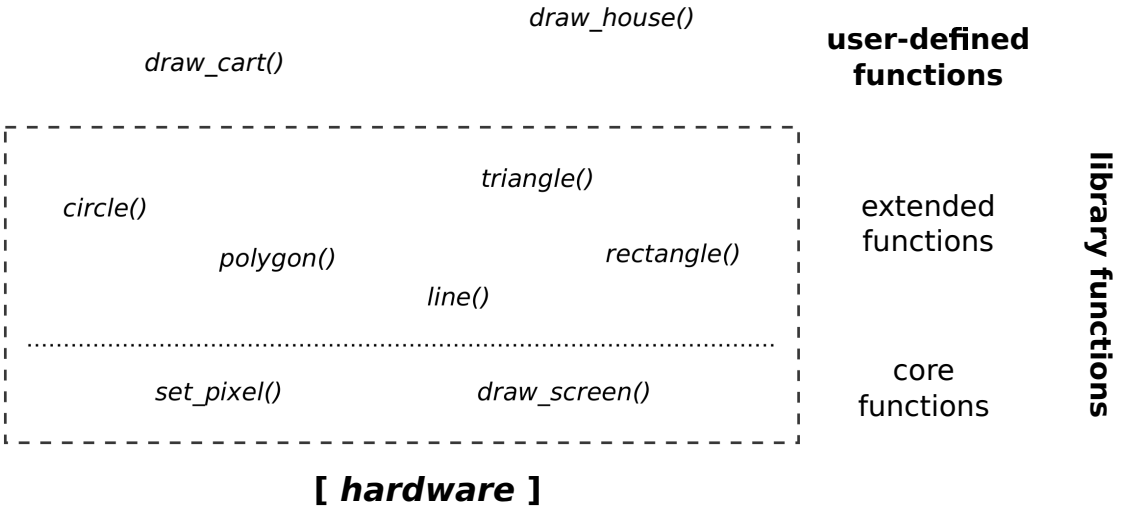

Figure 3.2.: Abstracting drawing routines.

One of the most advanced forms of building abstraction layers is to actually write your own computer language, targeted right at your problem area. (These are known as Domain-Specific Languages, or DSL.) I have used this once—to implement a text-adventure game, I created a simple language to describe game worlds in. In the right circumstances, DSL can be a powerful technique that makes for highly succinct and readable top-level code.

So, to sum up this section: the idea is to build "abstraction barriers" by hiding the implementation details. When each level of code only has to know about the level just below it, you can solve your problems more easily because you are working at the highest level of abstraction.

3.3. Modularisation

If abstraction is vertical encapsulation, then modularisation is encapsulation done horizontally. In this approach, the components of a software system are split up and grouped according to the tasks they perform.

The aim here is to make sure that code that is doing a similar job is close together. This means that it is quick to find as well as easily extendable, and even replaceable.

For example, it is common to have subsystems for file I/O (including things like logging or error handling), network code, the user interface, or the actual application logic.

One popular design concept is known as Model-View-Controller, illustrated in the simplified class diagram of my simulator Ecologia[5] (fig. 3.3).

The basic idea here is to keep the user interface (a GUI in this case) strictly separate from the logic. ("Logic" in this context refers to the rules governing a program's behaviour, in this instance the ecological entities and relationships that are being modelled.) To

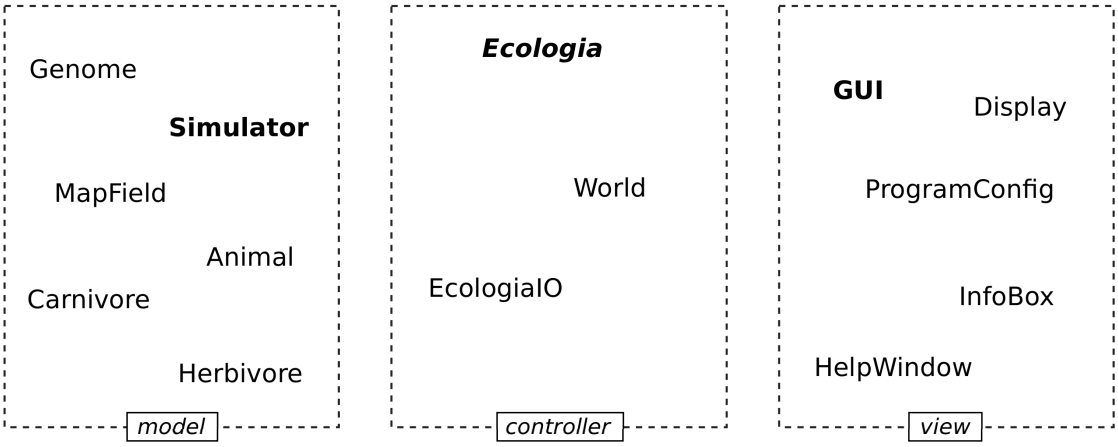

Figure 3.3.: The simplified class diagram of an ecosystem simulator.

do so, there are three packages involved: `model`, `view`, and `controller`. `controller` starts the program and initialises both `model` (the logic) and `view` (the user interface). It also offers some code that is needed by both sides—e.g. the `EcologiaIO` class. Lastly, it serves as an information exchange. Using the central data repository `World`, it passes user commands from `view` to `model` and information about the state of the simulation back the other way. Because of this set up, these two packages are actually not at all directly connected. Each only has to deal with the `controller`.

Separating `model` from `view` like this prevents the program code becoming an entangled mess of interleaved logic and interface code, which is very hard to read and understand. It has the added benefit that if you ever want to change the interface type (for example to a web or a commandline interface), you simply replace the `view` package and you are good to go. Nothing needs to be changed anywhere else.

The strict separation enforced here is a good example of the "loose coupling" concept, which states that there should be as few interactions between subsystems as possible. (Remember, encapsulation is about "maximally self-contained subsystems".) Connecting two subsystems only on a "need to know" basis is a requirement for the first advantage of encapsulation mentioned above: the ability to consider a subsystem in isolation from the rest of the program.

One last word about modularisation and coupling. One form of coupling that should be avoided if at all possible is global data (i.e. variables that are accessible from everywhere in the code). Global data is bad for at least three reasons. First, it often means that a lot of the code has to know implementation details (e.g. that a configuration table is stored as a `dict`). If the implementation ever changes, you are going to have to manually change a lot of code in a lot of places. Secondly, functions that depend on global state can be a right devil to test and debug, because global state can change in unexpected ways at runtime. (This gives rise to one of the key tenets of the Function Programming paradigm.) And thirdly, programs using global data have to be very careful that this is

always manipulated in the right order, and never in an invalid manner.

Of course, there will often be things that seem like they have to be known by every part of the code. Often, however, this can be avoided by passing these objects, or the relevant parts of them, down to the desired function via that function's parameter list. Alternatively, one can "hide" global variables behind access routines in a separate subsystem. This abstracts away the details of the implementation, and also helps to control how and when the data can be changed.

If you're tempted to use global data, remember McConnell's injunction: "The road to programming hell is paved with global variables." ("Although," as the Zen of Python goes on to say, "practicality beats purity.")

4. Dealing with Errors

The Jargon File[6] defines programming as: "A pastime similar to banging one's head against a wall, but with fewer opportunities for reward." Every programmer knows the frustration of looking for bugs that just won't be found. In fact, the majority of a software's development cycle is usually devoted not to the original writing, but to the subsequent debugging. Somebody who is good at finding and fixing mistakes therefore not only produces more reliable code, but is also a more efficient developer. So what techniques can we use to find bugs, or, if possible, prevent them occurring in the first place?

4.1. Finding Errors

The good news for computational scientists is that finding software errors has many parallels to something we all know how to do: scientific research. Both begin as you notice a strange phenomenon. Looking at this phenomenon, you formulate a hypothesis as to its possible cause. The hypothesis is tested, evaluated, adapted, or replaced, until you understand the causal mechanisms underlying your original phenomenon.

The scientific method as described above is of prime importance when looking for bugs. Although it can be tempting to dive in, modifying the program more or less at random, this never a good idea. (Said technique is known as "shotgun debugging" and frequently ends up creating more bugs than it solves...) Basically: don't guess! **Understand the problem, then solve its cause.**

Before you can fix a bug, you first have to find the code that causes it. This is best done with a **divide-and-conquer** approach. Analyse your program methodically, eliminating sections as you prove that they *cannot* contain the error. Iterate until you have pinpointed the exact class/function/line/statement that is the problem.

In some nasty cases, you may have to stabilise the error before you can find it. Not all bugs are always apparent—some only manifest themselves on a specific configuration of the input data or the system state. If you manage to reliably reproduce such a bug, you've often gone a long way towards nailing it down.

Perhaps the best-known debugging aid is a humble *print statement*. Especially in very small programs, all that may be needed to locate an error is to insert strategic print statements telling you what part of the program is currently executing, or the value of certain variables.

A slightly more advanced version of this is to utilise a *log file*. Instead of printing debug statements straight to the screen, have them saved to a file. This makes them easier

to work with afterwards and can also be used when a program is not run from the terminal. Actually, good logging practice can sometimes help diagnose an error without having to touch the code at all. (Side note: A good idea for logging is to introduce "verbosity levels" to govern how much information is emitted, depending on some user setting. Common values are: no logging/only errors/basic runtime information/extended debugging information.)

The most advanced debugging tools are known as (surprise surprise) *debuggers*. These are programs, usually language-specific, that can be used for in-depth inspections of other, running, programs. For example, one can use them to set break points at which program execution should pause. Using the debugger, one can then inspect the values of all current variables, or "step through" the continued execution of the source code until one finds the misbehaving statement. Thankfully, many modern development environments come with such capabilities out of the box.

4.2. Preventing Errors

Generally, preventing errors in the first place is much more efficient than having to find them after they've crept in. One concept that is important in this respect is that of **defensive programming**.

The basic idea here is to *actively ensure that things are as they should be*. Oftentimes, program crashes are caused by things external to the code itself: bad user input, a file that doesn't exist, a file that exists although it shouldn't, a network connection that has been interrupted. In defensive programming, the software checks for these possibilities before relying on the external resource. In a classic example, code that carries out some calculation with user input should check that what the user inputted was actually a number.

If a problem has been detected, the program then needs to decide what to do. Should it try to fix the problem by itself? (For example by substituting a known valid value, or by waiting for the connection to come back.) Or should it inform the user and either continue with the problem or quit completely? Which route the developer chooses to take depends on the kind of problem that is encountered, but also on the type of software. In some applications, it is of prime importance that a program be *correct*, i.e. that it never produces a result that is not spot on. In other applications, it may be more important to ensure that the program keeps running, even if things aren't perfect—i.e. that it is *robust*.

A second habit that helps reduce the number of errors is that of **refactoring**. Refactoring is about improving the quality of working code. As you work on your program, you should make it a habit to leave the code in a better state than you found it. If you see code duplication, move the duplicates into a single function. If you see ugly or difficult code, make it more readable. If you see global variables, hide them behind access routines. And so forth...

4.3. Anticipating Errors

The last big thing to do to decrease your bug count is to *anticipate that you will make mistakes*, and to **test** your software accordingly. (This is a huge topic, I will only skim it very briefly here.)

At the very least, you should test every component before you add it to your system, and then test the system as a whole. The easiest way of testing components is by using a REPL (read-eval-print loop). This is an environment provided especially by interpreted languages that allows the programmer to interactively execute code. For example, you can write a function and load it by copying it into the REPL. Then you can set about testing it. Does it produce correct output with typical input values? What happens when the input isn't typical? (Too large, too small, not present at all...) How do external conditions affect its working? Once you are confident the function does what it's supposed to, you can move on to the next one.

This is a very quick method of developing, and one well suited to exploratory programming (where you don't necessarily know the end result yet). However, as the stakes increase, your testing procedures have to become more thorough. Instead of the ad-hoc approach of the REPL, it is good practice to write some functions explicitly to test other functions—these are called **unit tests**. The more important correctness is for your program, the greater the percentage of it that should have tests written for it.

Over time, the test functions accrue into a testing framework, which should be run regularly to ensure recent changes haven't broken anything (a procedure known as *regression testing*).

In fact, there is a large school of thought that advocates writing your tests before you write your actual code. This *test driven development* (TDD) helps catch errors earlier and has the added benefit of getting developers to think more rigorously about the code they are going to write.

One final thought to keep in mind during all stages of the development cycle is that some areas of a program's code tend to be more error-prone than others. I'm not entirely sure why this is (perhaps they are the most complex), but numerous studies in industry attest to the fact. So it is probably a very good idea to try and identify these critical sections in your own code and pay special attention to them.

5. Programming tools: languages

Programmers love to debate about programming languages. Almost everyone has their favourite, so discussions as to the relative merits of each often degrade into "holy wars". Although no one will ever find the "perfect programming language" (despite numerous claims to the title), it is nonetheless instructive to compare different languages. After all, not all languages were created equal, and each have their individual strengths and weaknesses. Knowing about these can help you choose just the right language for just the right situation—in short, to use the right tools.

Of course, being able to choose the right language for the job assumes that you know more than one language. This is advisable anyway: knowing more than one language makes you more flexible as you can collaborate on a wider range of projects. Also, each language embodies a slightly different philosophy of programming—so knowing multiple languages gives you a broader perspective on software development in general. And you don't want the saying of "If all you have is a hammer, everything looks like a nail" to apply to you, do you?

Unfortunately, one is not always able to choose the language one would like for any given job. Perhaps you are joining an existing project, or you have to listen to what your boss and your collaborators say. But if you do have the choice, here are some things to keep in mind, and a selection of good languages for scientific computing:

5.1. Fundamental differences between languages

There are certain design decisions every programming language has to take. These will determine its general approach to programming, and make it more or less useful for different tasks. The most important of these decisions are:

Compiled vs. interpreted. In the end, the objective of any programming language is to take human-readable source code and turn it into computer-readable machine code. Interpreted languages do this "on the fly", as the program is executing. This means that they can be used interactively (with a REPL), and one doesn't have to recompile a whole system before testing a new version of it. This makes development in such a language easier and faster. However, because a program cannot be run without simultaneously running the interpreter, performance suffers. Compiled languages, on the other hand, separate the "translation" from the execution. A program is compiled once and can then be run on its own, without requiring further intervention from the compiler. This cuts out the middle man, generally making such languages

much faster. However, developing is somewhat more tedious. Also, a compiled executable will only run on the architecture and system it was compiled for, a portability problem interpreted languages can usually avoid.

Statically vs. dynamically typed. This is the question of how variable types (strings, integers, etc.) are recognized by the program. Dynamically typed languages don't care about variable types initially. A variable may be assigned values of various types over its lifetime without any problems. A type error is only raised when the programmer attempts to call an invalid type-specific operation on a value—such as indexing an int rather than a list. This makes the programming side of things easier and quicker, but can lead to some hard-to-diagnose bugs that ony manifest themselves at runtime. Statically typed languages avoid these bugs by forcing the programmer to assign a type to every variable, and throw an error if this variable is ever assigned to a value of a different type. Importantly, this type checking can be carried out during compilation, so that potential bugs are caught much sooner. The core rule is: in dynamic typing, type is something only values have, in static typing, variables have types too.

Memory management. All programs manipulate a computer's memory. While a program is executed, objects are created, stored in memory, and eventually have to be deleted again once they are no longer needed. In most modern languages, the task of deciding when an object can be deleted is carried out automatically by the "garbage collector", so the programmer doesn't have to worry about running out of memory inadvertently. In older (and especially compiled) languages, this was still a manual task; i.e. the assignment and deletion of an object in memory had to be explicitly included in the source code. Of course, this is a tricky task, and there are whole classes of bugs stemming from small mistakes in memory management. However, there are still some applications in which the overhead performance cost of garbage collection is so great (relatively speaking) that a manual memory management is the only viable choice. (This is especially the case in embedded systems programming.)

Other differences to be considered are the dominant paradigm of a language (more on this in the next post), and how large the "package ecosystem" of available third-party libraries is (this is usually a function of age and popularity).

5.2. Some languages to choose from

So, after this theoretical introduction, let's have a look at eight languages that are in some way important to computational biology. Of course, this is not an exhaustive list—amongst others, I have completely ignored web-based languages like PHP and JavaScript. But it does contain a brief introduction to a group of languages that are either already common in scientific computing, or have a great potential in this area.

- **C.** Most commonly used languages nowadays are in some way or another descended from C. C itself is a compiled, statically typed language with manual memory management. Executables produced with C can be very small, fast, and light on memory, making it a good choice for environments in which performance is critical (such as microcontrollers). However, it is infamous for being difficult to write well.
- **C++.** C++ is a superset of the C language that provides facilities for Object Oriented Programming (OOP) and expands the standard library. It is often used for performance-sensitive desktop software, such as games or simulations. Although it is somewhat easier to handle than pure C, it still retains the difficulty of manual memory management.
- **Java.** Java looks like C/C++, but isn't. It is a compiled, statically typed language, but includes a garbage collector (so has automatic memory management). It enforces strict OOP and has several features that make it well suited to development practices in large teams—though rather less so for an individual developer. A big bonus is its huge standard library (possibly the biggest of any language), which even includes a full GUI framework. An interesting feature is that it doesn't actually compile to native machine code, but to a "byte code" that needs the Java Virtual Machine (JVM) to execute. This means that it is slower than "true" compiled languages. On the upside, this byte code can be executed anywhere that has a JVM installed, making it extremely portable. Java is often used for desktop software.
- **Python.** Python is a dynamically typed, interpreted language with automatic memory management. The syntax is very clean and easy to learn, making it an excellent beginners' language. It's not in any way a toy language, though, enjoying immense popularity both for scripting and larger applications. The huge community means that there is a wealth of documentation and third-party libraries to be found online, including many for computational science. It is a pleasure to work in, not only because of its large standard library (it comes "batteries included", as they say). Unfortunately, it lacks the performance for computationally intensive tasks.
- **Perl.** Perl's popularity preceded that of Python, and it is a similar language in many respects. It is designed particularly with an eye for string manipulation, which has made it wide-spread in fields like bioinformatics that do a lot of string processing. It began to decline with the advent of Python, perhaps as the latter is much nicer to read (Perl's syntax is notoriously obscure).
- **R.** This is a scripting language designed specifically for data analysis and statistics. (Depending on which field you work in, you may use Matlab or SPSS for this purpose—biologists tend to go with R.) From a linguistic perspective, it is an excrutiatingly ugly language; its plethora of mutually incompatible data types are a right pain to work with. Its analysis and graph plotting libraries (both inbuilt and third-party) do however mean that it is a good choice for data analysis. Just don't do anything else with it. (It's performance is terrible anyways.)
- **Julia.** Julia is a very new language that is still in very active development, but

has begun to gain a sizeable following in the scientific computing community. It manages to combine the clean syntax of Python with the scientific library of R and a performance close to C—making it an excellent choice for many applications in scientific computing. (It achieves this performance by being just-in-time compiled: it has an interface like an interpreter, but actually produces machine code.) Its downsides are that the language is still evolving and changing very rapidly, and that there are few resources available online. Still very much worth the effort to learn, though.

- **Common Lisp.** This is perhaps a strange choice for inclusion in the list, as it is generally considered a pretty arcane language nowadays. However, it served as the standard language of AI research for several decades, and is still an advanced, powerful programming language. Notably, it was the first language to include a garbage collector, as well as originating a whole host of features that are slowly finding their way into more "mainstream" languages. (Julia has incorporated a surprisingly large number of these.) Common Lisp code can be both interpreted and compiled, making it both easy to develop and fast to run. Its biggest drawbacks are the small pool of potential collaborators (as hardly anybody knows it anymore) and a comparatively small number of available libraries.[7]

5.3. Working with libraries

Before I close, I'd like to add a thought on third-party libraries:

Using libraries is generally a great idea. You don't have to reinvent the wheel for every new project, instead building on what other (better?) developers have built before you. In many cases, one doesn't have the know-how to build an equivalent library anyway (perhaps when doing image manipulation). And if you want to use standard algorithms, such as for path-finding or compression, it is much better to go with a thoroughly tested public implementation than to hack up your own.

However, beware the dependency hell. The more libraries you use, the more problems you can have with unmet dependencies, out-of-date versions and similar troubles. This is doubly true if you're installing the libraries manually from source, instead of using a centralised package manager (like CRAN for R). Also, not all libraries are good libraries. In general, the more users a library has, the safer and easier it should be to use. Remember that although one should avoid reinventing the wheel, not every go-kart needs F1 wheels—in other words, the library you're thinking about using may really be much too advanced for the purposes you need it for. So use your discretion, and don't be afraid to not use a library, if doing so makes your program smaller, easier to read, and easier to port.

6. Programming tools: paradigms

When we talk about "using the right tools" in programming, that applies to a lot of different choices one can make. One is the choice of programming language, which we covered in the last article. Another important "tool set" to be aware of is that of programming *paradigms*.

Briefly, a programming paradigm is a way of thinking about programming. Specifically, it describes how we go about representing a real-world problem in computer code. A paradigm therefore includes both a mindset of how to analyse these problems, and a process for implementing the solution.

As computer science and programming practice advance, paradigms wax and wane in popularity. Here, I will briefly outline the three that are currently the most frequently used.

6.1. Procedural Programming

This could perhaps be considered the "default" paradigm. It is what is known as an *imperative* paradigm, meaning that programs are thought of as a series of commands for the computer. Programming, in the procedural paradigm, consists of step-by-step instructions to the computer about how to solve a given problem. ("First do this, next do that."[8])

On the implementation side of things, code is structured in blocks: loops, conditionals, and functions. (Hence the name of its historical predecessor, "structured programming".) Generally, functions are viewed not in their mathematical sense, but merely as a collection of instructions that are grouped together for convenience reasons. In short, they are not so much functions as "procedures".

If you write a simple script, chances are you're using this procedural paradigm. Its approach is also very prominent in the next paradigm, object-orientation.

6.2. Object Oriented Programming

If you're taught software development today, OOP is what you're taught. This paradigm was developed in the eighties, caught on in the nineties and was standard for the development of larger programs by the 2000s.

Its core idea is to view the world as a collection of interacting objects of various types. To do this, it uses *classes*, *objects*, and *methods*.

For example, imagine you wanted to model a city. A city consists of houses, so you would have a class `House`. Houses can do things, like `consumeElectricity()`—this would be a method. But not all houses are alike: there are appartment blocks, schools, factories, offices... Each of these subclasses inherit from the base class `House`, meaning they can all do that a basic house can (such as consume electricity). But they can each have more specialised methods, too. For example, the class `School` may have the method `teachChildren()`. (In the context of OOP, a method is defined as a function that can only be applied to values of a defined type; in other words, a function that is associated with a class.)

But there's not just one school in a city, neither is there just one appartment block. Classes, therefore, are instantiated to produce objects. An object is a specific instance of a class—not just any school, but "Loughby Secondary School". Each object has all the methods and attributes defined by its class and any superclasses, but has its own attribute values. (All schools have a name, but every school has a different name.)

This is a very intuitive approach to many programming problems, especially those that involve modelling real-world phenomena. It also encourages strong encapsulation and abstraction. Because of these strengths, it is currently the most popular paradigm for anything bigger than a 300-line script.

It does have its downsides, though. For starters, large object-oriented systems have a propensity to turn into a complicated tangle of classes and hierarchies—veritable phylogenies that are about as easy to understand as that of the prokaryotes. More significantly, not every programming problem lends itself to this way of thinking.

6.3. Functional Programming

Data processing, for example. Although one can use OOP to represent a data pipeline (such as that from DNA sequencing data over alignment to genome analysis), it is more natural to think about it as a set of functions applied in sequence.

That is just what functional programming does. In FP, the real world is not abstracted into a bunch of classes and objects, but into series of interlocking functions, each representing one process. Functions here are viewed in their mathematical sense, as entities that take some input and provide some output. Another way of putting it is that FP is about "evaluating an expression and using the resulting value for something"[9].

This means that functional code uses a lot less assignment operations than the imperative paradigms (it is considered a declarative paradigm), as expressions can easily be nested. Less assignments mean less variables being used, which avoids bugs caused by the improper/unexpected modification of variable values. Indeed, "pure" functional programming dictates that functions should be side-effect free, meaning they must not modify any global state at all. Given identical input, a function should always return identical output—regardless of the conditions elsewhere in the program.

The major advantage of this is that such functions are almost trivially easy to test and debug, and can thereafter be considered "black boxes". This greatly aids reliability and

understandability. On the downside, deeply nested function calls can be troublesome to read, and completely pure FP is hard to achieve.

A second aspect of FP is that because functions are so important, they are made "first-class objects". That means that the programming language treats them like any other variable type: you can pass functions as parameters to other functions, or have functions that return functions. This opens up a whole world of possibilities. Amongst others, it enables the `map` function, which takes in a function and applies it to each element of a given list (a quick alternative to a for loop).

Historically, FP has been around a lot longer than OOP. (Once again, Lisp was a pioneer in this department.) During the height of the "OOP-craze" it very much diminished in importance, but is now being rediscovered as a sensible solution for many scenarios.

6.4. Conclusion

Of course, functional programming isn't the Holy Grail either (even if some of its proponents are at least as zealous as many of those of OOP). Remember, the key message here is to **Use the Right Tools**.

There are many programming problems for which object-orientation is the most obvious solution, and there are many others for which a functional approach is a lot better. Equally, there are some programs that are so simple that you don't need either.

Use whichever is appropriate, and feel free to mix-and-match. Fortunately, most languages today are multi-paradigm and support more than one possibility. (Though, it has to be stressed, not all do equally well in all paradigms!) Just remember: not everything is a nail.

7. Developing in a team

Although a lot of scientists who write code usually work on their own, there will always be occasions when one becomes part of a development team. This could be because a junior colleague is joining your project (or vice versa), or because the software you are working on is so large and complex it requires the joint efforts of several people to complete. These scenarios not only make the principles we have already discussed more important (readable code, good architecture, etc.), they also necessitate a whole new set of procedures.

7.1. Exchanging code: version control

(If you already know about git, you can skip this section.)

The first issue you'll face is how to pass your source files around. Everybody needs an up-to-date version and you mustn't break anything if two people have changed the same file—how do you do that?

For this purpose, there is what is called *version control software*, the most popular of which is git. In a normal git setup, the code is hosted in a central repository. Each individual developer can check out this code (i.e. download it to his own machine) and change it at will. Each batch of changes is committed—this creates a "snapshot" of the project in the current condition. Finally, the developer pushes his changes back up to the master repository, from were others can pull them into their own local copies. Git merges all changes from all developers so that nothing is accidentally lost or overwritten. (As there are plenty of excellent tutorials out there on how to use git, I won't go into details. See the bibliography for more.[10] [11] [12] [13])

The most popular git hosting site by far is Github, which you have almost certainly heard of before. It is free and allows you to easily share your code with just about anyone. A popular alternative is Gitlab, which has the advantage that you can host it yourself onsite.

An important point to make is that you should be using git anyway, even if you're just working by yourself. The series of commits creates a great timeline to follow your project's progress, and the ability to jump back to a given commit means you've always got backups in case you screw something up. Also, sharing your work with others is dead easy if you've already got it sitting on Github.

7.2. Improving code: collaborative development

Developing software in a team offers enormous benefits, if done right. For a start, the quality of the software itself can increase dramatically, because there are more minds thinking about it and more eyes looking out for mistakes. At the same time, the team benefits too, as its members exchange ideas and share their experience and knowledge. This effect is especially strong for junior team members, who can learn rapidly from observing the way more experienced developers work.

Over the last few decades, industry practice has identified various techniques to maximise these benefits. Two of the most efficient are pair programming and code inspections, which I will briefly outline here. Both of these have been found to have an error-detection rate of around 40-70%, which is even higher than good testing achieves. (Disclaimer: I've never actually had the chance to try these out myself, sadly. I'm mostly going by McConnell's Code Complete 2 and various online sources[14].)

7.2.1. Pair programming

Pair programming is a bit of an unusual approach in that it entails two programmers working on the same computer. One, the driver, is the one actually typing into the keyboard. The other, the navigator, watches for mistakes and thinks ahead about what must be done next. The two partners keep up a conversation on what they are doing or thinking and regularly switch roles.

This technique is especially effective when working on difficult sections of the code. Pairs keep each other focussed, spot more mistakes and think of more possible approaches than an individual developer would. Together, they deliver higher quality code in a shorter time than alone.

However, it takes a bit of practice to get used to this style of coding. Not all pairs combine well, and not all code is amenable to the technique's specific benefits. When it works, though, it works well; and developers often enjoy the team-oriented approach.

7.2.2. Code inspections

Code inspections are a highly formalised version of code reviews that are intended to find errors during the development phase. An inspection will be chaired by a designated moderator (not the author of the inspected code!). This team member sends out a copy of the code to a small number of reviewers (about one to four) and arranges a meeting for the actual discussion. Each reviewer prepares by reading the code himself and annotating any problems he finds, based on a checklist of common mistakes/good practice. During the meeting, the code is read through section by section as the reviewers give and discuss their comments. Importantly, the aim of the meeting is not to find solutions, but purely to identify problems. Each problem is recorded and the complete list is later handed to the author for fixing.

Inspections are a very thorough tool that has proved to be highly effective at finding errors. As McConnell writes:

> *A study of large programs found that each hour spent on inspections avoided an average of 33 hours of maintenance work and that inspections were up to 20 times more efficient than testing (Russell 1991).*

Although they might seem overdone in their formality, experience has shown that less structured code reviews usually perform significantly worse. One challenge in the inspection is to keep the discussion on point and technical. This is where the moderator comes in, as it is his task to ensure that the discussion doesn't drift off-topic or become personal (attacking the author).

1st Postscript. The NASA space shuttle software is renowned as being the most bug-free software ever created. Through stringent development practices, its engineering team managed to reach a rate of just one error in half a million lines of code. (Industry average is 1-25 errors per 1000 LOC!) For an instructive write-up of how they do this, see here[15].

2nd Postscript. For a simple code inspection checklist I created for our institute, see appendix A, or download it from the website[16].

8. Some final thoughts on programming

In the past seven posts of this series, we've looked at how to make software understandable, reliable, and extendable. We've seen techniques for dealing with errors, reducing complexity, and developing in teams. We've touched on different programming languages and paradigms, and conventions for documenting code. Of course, these have been very cursory glances; but hopefully enough to give a brief overview of what to think about when developing software in a scientific context. Now, in closing, I want to mention two last topics and give a few pointers on where to go from here.

8.1. Optimisation

Thankfully, we are no longer in an era where every byte of memory is precious[17] and programmer time is cheaper than computer cycles[18]. Today, most programmers rarely have to worry about performance issues when writing their code.

Unfortunately, computational scientists often still do. Whether we deal with huge data sets (such as genomic sequences or imaging data) or construct complicated mechanistic models, we are often pushing the boundaries of what our computers can do. So if we find our program's memory consumption exceeds the RAM we have available, or the run-time starts to be measured in weeks instead of minutes, we need to optimise. This takes some skill.

The first rule of optimisation is: do it last! In the words of Donald Knuth: "Premature optimisation is the root of all evil." *First* make sure your program is correct, *then* figure out where the performance bottlenecks are, *then* try and make them more efficient.

Figuring out performance bottlenecks first, before you start tweaking your program, is incredibly important. Usually, one small part of a software is responsible for the lion's share of its run-time or memory consumption. Improving this bottleneck can get you orders of magnitude more efficiency. And as long as the bottleneck remains, improving all the other pieces is virtually useless.

To figure out where these bottlenecks are, you can use a program known as a "profiler" (every language has at least one available). This will tell you in detail which function calls use how much time and memory. Alternatively, if you just need a quick-and-dirty overview, you can use your language's `time()` call to calculate function run-times yourself (although profilers are very much worth getting to know!).

(Keep in mind that you can optimise for processing speed and a low memory footprint—but at some point, you are usually going to have to settle for a trade-off between the

two.)

Three common types of bottlenecks that you will face are those related to algorithms, data structures, and I/O (input/output) calls.

Algorithms and data structures are a huge component of computer science as a discipline that I won't even try to cover here. Generally, you don't need an in-depth knowledge of the various algorithms anyway, but it does help to at least be aware of tried-and-tested solutions to common problems like sorting or searching. Data structures is something you should try to learn more about, as the right choice of data structure can make a world of difference to your program's performance. You can have a look at this online resource[19], or check your local library for a relevant text book on the topic.

Whereas algorithmic optimisation is a very mathematical endeavour, I/O optimisation is much more about the hardware you're working on. Often, it isn't the actual calculations that make a program slow, but the time it takes the computer to move the bits and bytes around. Common operations that take a lot of time are network connections, disk reads/writes, or screen output. If you do a lot of these successively, the effect will be noticeable. Here, it often helps to cache data in memory (instead of re-reading it every time it's needed), or buffering output data and flushing it out in one go (instead of setting up a new connection for every little bit).

Overall, optimisation is a huge topic that quickly leads into highly advanced techniques based on very specialised knowledge. Therefore, I will say no more about it, except to encourage you to keep looking and learning, to find out what works in your scenario in your language on your machine.

8.2. Licenses

Strictly speaking, licenses are not an aspect of software development, but if you work with or create open source software, you'll have to know at least a bit about them.

"Open source" means that, unlike for most commercial software, the source code for a program may be freely inspected, modified, and redistributed. There's a whole range of licenses that allow this, each with slightly differing conditions. It can all be a bit confusing, but fortunately, there are good overviews[20] available that can be consulted.

Perhaps the two most important are the GNU General Public License (GPL) and the MIT license. The GPL is what is known as a "copy-left" license—not only must the source code for the work itself be made freely available, but all future derivative works must be published under the same license. The MIT license is less restrictive: it only stipulates that any redistribution of the software must retain the original license notice, but derivative works may choose a different license model.

But don't be too worried: if you're choosing an license for your own software, it basically boils down to a matter of preference and software-political opinion—unless your employer has a stated policy in the matter. And if you're joining an existing project, that choice has already been made anyway.

8.3. Going on from here

Slowly, the importance of good software development practices is being recognised by the scientific community. Recently, the topic was even covered by Nature[21]. Various groups are forming to train, support, and connect scientists who write software—examples being the Software Sustainability Institute or the Research Software Engineer Association[22].

As I have previously said, the purpose of this series of articles is to give an overview of topics that must be considered when developing software, especially in a scientific context. I hope this in itself proves useful, but must stress that everything I have touched I have touched only very briefly. Much more could and perhaps should be said—but diving deeper remains an exercise for the reader.

In closing, I can only repeat my mantra from the beginning: to become a better programmer, you must *read*, *write*, and *repeat*. In that spirit, happy hacking!

A. Code Inspection Checklist

These questions can be used to guide code reviews:

- Is the code properly formatted?
- Are variables clearly and succinctly named?
- Is the code split up into functions and files of appropriate lengths?
- Is code understandability aided by suitable comments?
- Are there no repetitive elements that should be refactored?
- Are there no hard-coded values that should be either named constants or configurable parameters?
- Does the software check for bad input (defensive programming)?
- Are libraries employed where useful, but not unnecessarily?
- Does the design make good use of encapsulation and abstraction?
- Is the user interface easy to understand and use?
- Is there sufficient documentation to allow an external person to use and modify the software?
- Can the program be automated?
- Are there mechanisms for verifying correctness? (Error messages, logging, unit tests...)

B. Bibliography

These are the two books I primarily relied on for this project:

- *Structure and Interpretation of Computer Programs*, Harold Abelson and Gerald Sussman, with Julie Sussman (MIT Press 1996, 2nd edition). The classic textbook developed for and used for years in the MIT introductory programming class. Simply known as SICP, it is an elegant introduction to many important concepts of computer science.
- *Code Complete 2*, Steve McConnell (Microsoft Press, 2004). A 900-page magnum opus on all the practical aspects of software development, covering everything from indentation to debugging and team management.

Other useful sources for further reading are:

- Wilson et al. (2014) *"Best Practices for Scientific Computing"* – Things to keep in mind when developing code (largely similar to the topics covered in this series). adf
- Wilson et al. (2016) *"Good Enough Practices in Scientific Computing"* – Things to keep in mind when working with computers in science generally. Includes a brief section on writing code, but also touches on things like data management and project organisation.
- *Netherlands eScience Center guide*[23]– A comprehensive guide to software development produced by the Dutch expertise center for research software. Goes into a lot more detail on many topics covered in this series.

Notes

1. https://www.sciencealert.com/a-bug-in-fmri-software-could-invalidate-decades-of-brain-research-scientists-discover
2. “Structure and Interpretation of Computer Programs”, see above
3. https://www.python.org/dev/peps/pep-0020/
4. https://terranostra.one/posts/ASCII-Art-Animations-in-Lisp.html
5. https://terranostra.one/posts/Project-Ecologia.html
6. http://www.catb.org/jargon/html/P/programming.html
7. https://terranostra.one/posts/An-Impression-of-Common-Lisp.html
8. Kurt Nørmark, "Overview of the four main programming paradigms", Aalborg University
9. *ibid.*
10. https://www.sitepoint.com/git-for-beginners/
11. https://www.tutorialspoint.com/git/index.htm
12. https://git-scm.com/book/en/v1/Getting-Started-Git-Basics
13. https://help.github.com/en/articles/set-up-git
14. e.g. https://www.cs.cmu.edu/∼aldrich/courses/654-sp07/slides/2-inspection.pdf
15. https://www.fastcompany.com/28121/they-write-right-stuff
16. Download the standard version or the self-explanatory version.
17. https://www.freecodecamp.org/news/where-do-all-the-bytes-come-from-f51586690fd0/
18. http://www.catb.org/jargon/html/story-of-mel.html
19. https://www.tutorialspoint.com/data_structures_algorithms/index.htm
20. https://choosealicense.com/licenses/
21. https://www.nature.com/articles/d41586-019-02046-0
22. https://www.software.ac.uk/, https://rse.ac.uk/who/
23. https://guide.esciencecenter.nl/best_practices/overview.html