THE CHALLENGE OF SOFTWARE DEVELOPMENT FOR SCIENCE

Félix Bertoni

14.11.2020

ABSTRACT

Transparency note: this paper serves both as an informative medium and as a way to inform about and promote my Software Development for Scientists course and related teaching services. The course book and other materials are published as free access resources and are sufficient for autonomous learning. Only my teaching services, as course supervision, are possibly paid, and are for entities wanting to get support for their members to learn software development and/or to support my course writing work. More on all that in section 6

Software development can take significant space in a scientist's life, would it be to display data, perform simulations or calculations, or even to develop tools for other scientists. Unfortunately, not all scientists had the chance to be taught software development basics, the time to explore this wide and unfriendly field of knowledge by themselves, nor consciousness of the underlying challenge of developing software for science.
In scientific field, software tools have requirements and constraints quite different from others, to ensure precision of computing and reproducibility of results. This white paper is meant to raise awareness of scientists on objectives they may want to achieve when developing software, and to give them bribes of solutions to approach them. It only gives sparse advice that I consider to be both extremely beneficial and not intuitive or unlikely to be encountered in autonomous learning. In order to keep this paper short, deep explanations and discussions about certain advice were skipped, which can make them a bit hard for complete beginners to understand. Reading through this document should be still beneficial even for complete beginner, especially sections 1, 5 and the start of 6.

ABOUT DOCUMENT

Keywords: Software development, reliability, reproducibility, course

Est. reading time: 20min — 40min

Type, version: white paper, 1.0

Date: 14.11.2020 (generated: 11.01.2021)

Author: Félix Bertoni, contact@felix-bertoni.fr

Sources: https://gitlab.com/feloxyde/courses

License: CC BY-SA

CONTENTS

1 Efficiency **2**
- 1.1 Editor . . . 2
- 1.2 Automation . . . 2
- 1.3 Version Control . . . 3

2 Readability **4**
- 2.1 Coherence . . . 4
- 2.2 Expressiveness . . . 4

3 Maintainability **5**
- 3.1 Decoupling . . . 5
- 3.2 Polymorphism . . . 6

4 Reliability **7**
- 4.1 Anatomy of a test . . . 7
- 4.2 Testing levels . . . 7
- 4.3 Test reliability . . . 8
- 4.4 Test Driven Development . . . 8

5 Learning software development **9**
- 5.1 Skills portability . . . 9
- 5.2 Do it ! . . . 9
- 5.3 Small focused steps . . . 9
- 5.4 Finding knowledge . . . 9

6 My course and services **10**
- 6.1 Course . . . 10
- 6.2 My services . . . 10

EFFICIENCY

The goal of nearly any software is to help its user to perform a task with ease. That's *efficiency*. Why would one spend hours to trace a curve with 1 000 data points while a software could do the same in seconds for 1 000 000, more accurately and reliably ?

Efficiency prevents frustration: no effort was wasted if no effort was put in failed task at first. If you need 1 key press and 10 seconds of waiting for plotting 1 000 000 data points, you not feel bad if your data appear to be erroneous: just correct it and re plot.

Software development is extremely frustrating, because it involves a lot of trivial, repetitive and attention demanding tasks, as compiling, testing, or collaborative work synchronization. Therefore, being efficient is also important when *developing* software. Fortunately, there are plenty of tools to get these tasks from *painful and long* to *easy, fluid and quick*.

Please note that advice presented here are also applicable to any text-based work, as for example LaTeX document editing.

EDITOR

Source code editor, as *VSCode*, *Vim*, or *Emacs*, are software to... well... edit source code. This may seem really simplistic said like that, but most of them pack a ton of features to help developers code with ease.

Syntax highlighting displays words in different colors and font to ease code reading.

Autocompletion tries to predict what one intend to write when typing, and proposes completion or corrections.

Code snippets are a wider autocompletion feature, able to automatically write entire blocks of code. They usually require a bit of setup to be used.

Block collapsing hides blocks of code under a one-line summary, making wide or complex documents easier to read and navigate.

Refactoring allows quick and generalized changes in the code. In most editor it takes the form of a *find and replace* tool, however advanced refactoring features can take language's syntax in account and work on thousands of files at once.

Code navigation enables quick navigation through the code, for example to find all uses of a component.

When working, keyboard shortcuts are not to be neglected. Try to force yourself configuring and using them for tasks you perform often.
Finally, editors are highly customizable, often through the use of extensions, and if a feature is not in it from start, it is likely implemented as a plugin. Some can even interface themselves with tools presented in the rest of this section.

AUTOMATION

In software development, most tasks can be partially or fully automated. It includes testing, building the software, and creating documentation. Automation is a keystone in software development, making it much more simple and pleasant, by turning time or attention consuming tasks into simple and reliable ones. Ideally, every possible set of task you need can be triggered using a single command or action, as `make test`.

Scripting tools, as *Bash*, *Expect*, or *Python*, are convenient for easy or unusual task automation, as for example manipulating files.

Build automation tools, as *Make* or *Meson*, allow defining a hierarchy of actions to perform, some dependent from others, and execute them in the right order when needed. While they are intended for building software, they are extremely powerful tools, for both software development and other projects, as writing LaTeX document. Some of them, as *CMake*, are specialized for a programming language or family of languages.

Analyzers are automatic software that analyze your code either by reading it or running it, and give you all kinds of reports. For example, a *linter* will give you report on structure and style of your code, while tools like *Valgrind* can spot memory management errors. Analyzers are usually bound to a specific language, but are fairly easy to use.

Debuggers allow you to run your code step by step, and observe what happens inside, or even modify state of the program on the fly. They are really convenient to understand and fix bugs, as their names imply. They are unfriendly at first, because they have simplistic, often command-line, interfaces. However, learning about ones corresponding to your main languages will prove to be a great help along all your projects.

There exists automatic or nearly automatic tools for any software development task. Some are fairly difficult to set up, while others are trivial. If a task is performed often in the project, it is wise to look for a tool and automate it if possible, and if such tool exists, to take time to set it up or rearrange an existing setup.

VERSION CONTROL

Version control software, as *Git*, or *SVN*, are the most important tools in software development. They maintain a history of changes happening in the software, and make difficult operations, illustrated figure 1, nearly trivial.

Zero loss: always keep a trace of what happened in the code in history.

Commit: take a snapshot of your project and save it in history.

Revert: come back to an older version (commit) of the code.

Branch: create two parallel histories. Extremely convenient for teamwork or experimental versions.

Merge: bring two parallel histories together. Extremely convenient for teamwork. Changes introduced in one history are applied to the other.

Conflict resolution: If a merge operation results in incoherent changes, version control software precisely points them out to help to get back into a coherent state.

Version control acts as a backbone for both programming and project management, as well as a code sharing medium. They can be used for much more than software development, as LaTeX document writing, art, or design. Version control tools all work quite similarly, so when you know one, you know roughly all of them. As they are simple to set up, and consume close to no resources, while acting as a safety net for the project, using one is mandatory for any software development project.

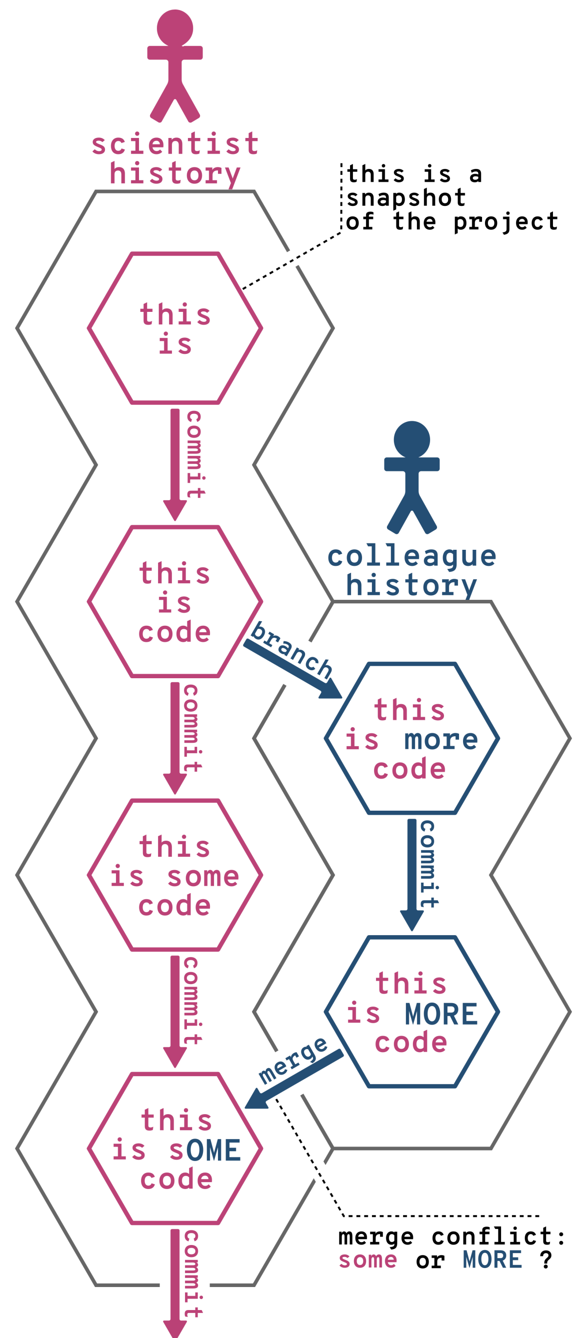

Figure 1: Version control workflow (git)

READABILITY

One of the key difference in science software versus other software is that science requires criticism and reproducibility. In general software development, readability of the code is simply a mean of achieving other points, as maintainability. In science, it can be considered as a requirement by its own.

Readability is part of a larger matter, code quality. In this section, we discuss practices that can enhance both code quality and readability. Please keep in mind that readability and code quality have subjective parts. Also, we need to think about human readers, but also about software that will parse your code, as interpreter, compiler or analyzers.

COHERENCE

Exactly like for a natural language or dialect, humans get used to the code they read. In that reason we want our code to be *coherent*: whatever choices we make regarding our code, we want them to be applied to our entire project. This way, readers of our code will have to learn only once how to read our code, and get used continuously to it through their reading.

Your programming language probably has official guidelines regarding code presentation, and possibly software design. If so, it is wise to apply them, at least partially, so anybody used to the language will also be used to your code's look. Keep in mind that not only your close colleagues will read your code: other researchers will, and maybe people from help forums.

In case no guideline exists, or they are insufficient, just create some, for yourself or your institution, write them somewhere and apply them. Software as linters and formatters can assist us in enforcing a set of rules in our code.

EXPRESSIVENESS

Formulate your code to express what you *intend* to do rather than *how* you do it. Ideally, your intent is expressed only through syntax and semantics of the language you use, in order to allow software as compiler to understand it as well. For example, one can use different types for different roles.

Listing 1: C++ intent through types

```
class Person {
  /*... defining a student */
};

class Wall {
  /*... defining a wall */
}

void paint(Wall w){
  /*... how we paint a wall */
}

void hug(Person p){
  /*... how we hug a person */

//labWall is a variable of type Wall
Wall labWall = Wall();
pain(labWall); //OK
hug(labWall); //NOPE, compiler will
    ↪ complain
}
```

In the previous listing, we defined two types, `Wall` and `Person`, and corresponding operations, respectively `paint` and `hug`. If we try to hug a wall, the compiler will raise an error: through types we have defined, it understood that it isn't logical to *intend* to hug a wall

Unfortunately, it is not always possible to express everything we want in a way the compiler can understand it. In that case we can use names, as for example variable names, as a complement of syntactical expression. Name functions, types, classes in regard of what they *are*, and variables in regard of what they are *used for*.

Listing 2: C++ intent through naming

```
class Wall {
  /*... defining a wall */
}

//here, our wall is used as a lab wall
Wall labWall = Wall();

//here, our wall is used as decoration
Wall decoration = Wall();
//(note: compiler is OK with that)
```

We can see that compiler will not complain about using a wall as a decoration, even if it is quite discussable. However, for a human reader, it will maybe look a bit strange and trigger thinking regarding the correctness of the code.

When naming is also insufficient, for example if some context or an overview of software structure is needed, a comment can make code clearer.

Listing 3: C++ intent through comments

```
//that's ok for modern art !
Wall decoration = Wall();
```

Comments are to be used wisely, as too many comments can obfuscate code rather than clarifying it. The general rule is: do not use comments if you can express things with either syntax or naming.

MAINTAINABILITY

Maintainability of a software pictures how easy and safe it is to introduce changes in its behavior or code. Such changes can happen for various reasons.

- Fix a bug
- Adapt software to new environment, for example enable scaling on a supercomputer, or supporting new libraries.
- Add or change functionalities of the software. For example, allow an image analysis software to process video.
- Change misleading code structures, as a misnamed function.

Ease of introducing a change is how much code we need to write or rewrite to actually implement the change. Safety of such change is related to how many bugs may create while doing so.

Readability and testing are important for maintainability, but its foundation lies in how the software is structured and organized, *software architecture*. Let's see some architecture principles for maintainability.

DECOUPLING

The rule of thumb in software architecture is to express components, as classes or functions, as isolated elements independent of each other and coordinated through minimal *interfaces*. Two components are independent if none needs to know *how* the other works for them to work together. That practice is called *decoupling*.

Let's imagine we want to write a program computing the average of a list of integers. It will be made of three components, *average* function computing average, *list* type, and *integer* types.
To achieve decoupling, we will first focus on *how* we want to manipulate components rather than focusing on *what* they are. This will allow us to define required interfaces for our components to interact with each other.
Computing average of a set of integer requires to add them together and divide their sum by an integer. These two operations, `add` and `integer_divide`, define our `Value` interface.
Computing average of a list of values requires to access each element of the list, sequentially. This can be done through a `next_element` operation. It will define our `Collection` interface.
Now, when defining our average function, `avg`, we will work on a *collection of values* instead of a *list of integers*. Finally, when implementing `List` and `Integer` types, we will ensure they support operations defined in `Collection` and `Value` respectively, for them to be compatible with `avg` function, as shown figure 2.

Means of interface definition depend on the programming language used. As for example, in *Java*, programmer can explicitly declare interfaces,

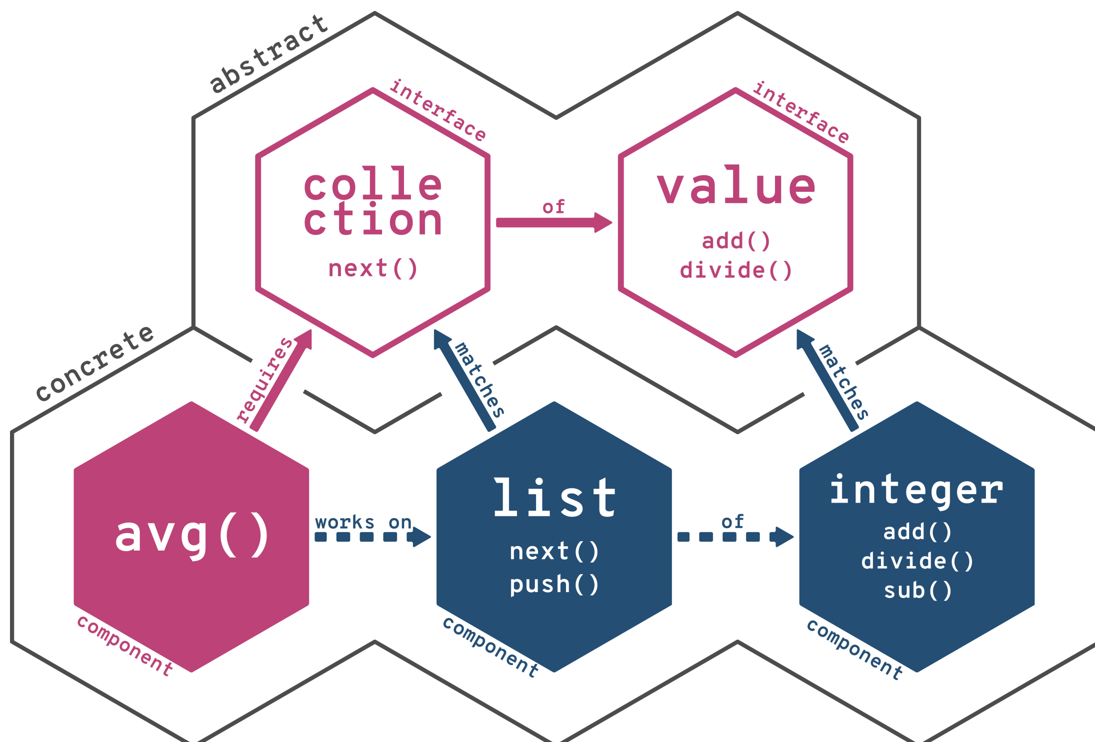

Figure 2: Example of decoupling

and then components have to explicitly implement it.

Listing 4: Java interface declaration

```
/* declaration of Collection*/
interface Collection {
  element next();
}

/* implementation */
class List implements Collection {
  element next() {
    /* some code here */
  }
}
```

The opposite example of Java interfaces would be *Python ducktyping*. Ducktyping is an implicit way of declaring an interface. Interfaces are never declared, and as long as a component has all required operations for a manipulation, it is compatible with this manipulation.

Listing 5: Python ducktyping implicit interface

```
class Integer:
  def add(other):
    #some code here
  def divide(other):
    #some code here
  def sub(other):
    #some code here

#here, Integer implicitely
#implements interface "Value"
#from our previous examples:
#it has add and divide operations
```

Implicit interface declaration offers more flexibility, while explicit interface declaration is much clearer.

POLYMORPHISM

Decoupling synergize extremely well with another feature of most major languages: *polymorphism*. Polymorphism allows manipulating component of different types with their respective operations, given they share a common interface. Here is a quick Python example. First, let's define two different types with a common interface.

Listing 6: Python polymorphism example

```
#defining two different types of components
class A:
  def scream(self):
    #printing AAAA on terminal
    print("AAAA")

class B:
  def scream(self):
    print("BBBB")

#they share a common, implicit
#interface, which is the scream operation
```

Then, we create a list of mixed components from both types, and, for each element of the list, we call the `scream` operation.

Listing 7: Python polymorphism example

```
#thats a list: A, B, B, A (it rocks)
lst = [A(), B(), B(), A()]

#making all elements scream
for element in lst:
  element.scream()
```

The output we can expect from executing this script is as follows:

```
AAAA
BBBB
BBBB
AAAA
```

We can see that operation corresponding to the type is used each time: `A` elements print `AAAA`, while `B` elements print `BBBB`.

Polymorphism can take various forms, happening either at compile time or at runtime. We can cite *operation overloading*, *genericity*, and *method overriding*, as being commonly used ones. Polymorphism, in synergy with decoupling, allows writing extremely flexible and extensible software with relatively small efforts.

RELIABILITY

Science needs reliable data and thus reliable software to process it. Ensuring a software is reliable is a complex matter. Having a clean software, from both code and structural point of view, is a big help. But at some point one has to test the software.

Testing is beneficial in every aspect of software development, and is ideally automated using testing frameworks, automation tools, testing software, or a combination of those. In the initial development of a feature, tests ensure that the feature is correctly implemented. In the rest of the lifetime of the software, those tests will ensure the feature has not been broken. Being able to verify if software is still properly working after some changes increases maintainability.

Testing may feel like a hassle. If it does, it indicates a deeper problem: tests are not automated enough. The simpler tests are to run, the more often they will be run, and the more developers are encouraged to write more tests.

ANATOMY OF A TEST

A test has three main elements, that can be specified with more or less flexibility.

Context is everything that is not part of the target, software or component, of the test. It can be another software, a server on a distant machine, a configuration file...

Precondition is the state in which the target shall be before starting the test. For example, in order to test a *save file* functionality, the precondition would be that a file is already loaded in the software.

Execution and verification describes manipulations of the target and what is their expected results. That is the actual verification of the test, and it can focus either on output of the target or its state after manipulations.

As in a scientific experiment, when testing software we usually want our test environment to be *controlled*. For example, if we test a webpage communicating with a server, we want our test to fail only if the webpage fails, and not if the server fails. Regarding this concern, we try to replace everything that have a fair amount of chances to fail by *test doubles*. In the case of our webpage, a server test double would not do any computing and only send predetermined answers, reducing the risk of failure due to its simplicity. Test doubles can be used as probes to verify that the target had the correct behavior, in this case, they are called *mocks*.

Writing tests is time-consuming. Context and preconditions are often shared, partially or fully, between several tests. It is convenient to turn them into reusable elements, known as *fixtures*.

TESTING LEVELS

To test a software, one can have different approaches, or *levels*. Or a combination of them.

System testing handles the software as a whole, caring only about end functionalities and not about implementation. It is usually quite hard to automate. It answers the question *Is my software properly working ?* It is convenient to track errors at a macro scale.

Unit testing breaks down software into as small as possible component, units, as functions and classes, and test them in isolation by extensively relying on test doubles and mocks. It answers the question *is this component properly implemented ?* Unit testing allow fine granularity bug detection and faster debugging. It is mandatory for complicated or widely used components, but can be time-consuming if test doubles and mocks are necessary. Decoupling eases unit testing.

Integration testing tests sets of components together. It acts as an in-between for unit and system testing. It answers the question *Are these components compatible ?*. Integration testing checks that components required and matching interfaces are coherent. Also, integration testing can act as unit testing in case all but one component have been extensively unit tested and therefore serves as mocks.

It is often hard to mock interactions between a software and its environment, inputs and outputs, as for example user key press. To mitigate this problem, it is advised to decouple as much as possible environmental code with inner logic code.

Listing 8: Decoupling print in Python

```
#creating a console IO class that can be
    ↪ mocked if needed
#it simply wraps code
class UserInterface:
  def send_output(message):
    print(message)
  def get_input(message):
    return input(message)

#example of a function using it
#that can receive a mock instead
def hello_world(ui):
  ui.send_output("hello world !")

#using it with concrete component
hello_world(UserInterface())
#or mock (defined elsewhere)
hello_world(MockUI())
```

TEST RELIABILITY

Tests need to be as reliable as possible, as we want to trust them when evaluating quality of our software. We first want to keep tests as simple and straightforward as possible. For example, use plain values for expected result instead of computing them. Using a testing library or framework helps since it provides battle-tested test functions and constructs.

We also want to ensure our code is *covered* by tests as much as possible, so every line of our software is run during tests execution. It is done by *coverage report* tools, that are fairly easy to use and setup. Coverage does not point out whether our tests are good or not, but warns about what has not been tested for sure.

Finally, a failing test is better than no test at all. If you do not have time to implement tests for components, but you know you need to in the future, it could be wise to create empty, always failing tests to remember that you need to test this particular aspect.

TEST DRIVEN DEVELOPMENT

Writing tests is a critical activity in software development, especially for science. You may be tempted to skip it to save time if the program *feels functional*, exposing yourself and colleagues to latter, vicious failures of your software. To prevent this situation to happen, developers can either rely on self-discipline and rigor, or they can use a development method favoring testing, as Test Driven Development.

The idea behind TDD is *write tests for something before coding it*. TDD works with tiny successive iterations, each implementing part of a functionality. An iteration, summarized figure 3, is conducted as follows.

1. Write a really simple and tiny test, usually a unit test, for a functionality or part of a functionality.
2. Check that test is failing. Compilation failure is considered as failure. If test pass, rewrite it so it does not pass.
3. Write just enough code to have the test passing.
4. Check that all tests written up to know pass, otherwise not correct the software or update tests. Check that code quality is sufficient, otherwise refactor code that needs to be. Once all tests are passing and code quality is satisfying, move onto the next iteration.

Such practice encourages decoupling and ensures you do not write untested code. However, it can be difficult to apply in certain situations. For example, when writing code directly tied to inputs and outputs of the software, writing tests can be complicated and time-costly, making TDD slow or counterproductive.

Keep in mind that Test Driven Development, as any software development method, is open to adaptation to your specific needs and combination with other methods. For example, one may want to write more than one tiny test per iteration. Purpose of a development method is to increase work efficiency, and, indirectly, software quality. In regard of this objective, whatever method you choose, it is important to regularly evaluate whether the method is fit to your case or not, and change it if need be.

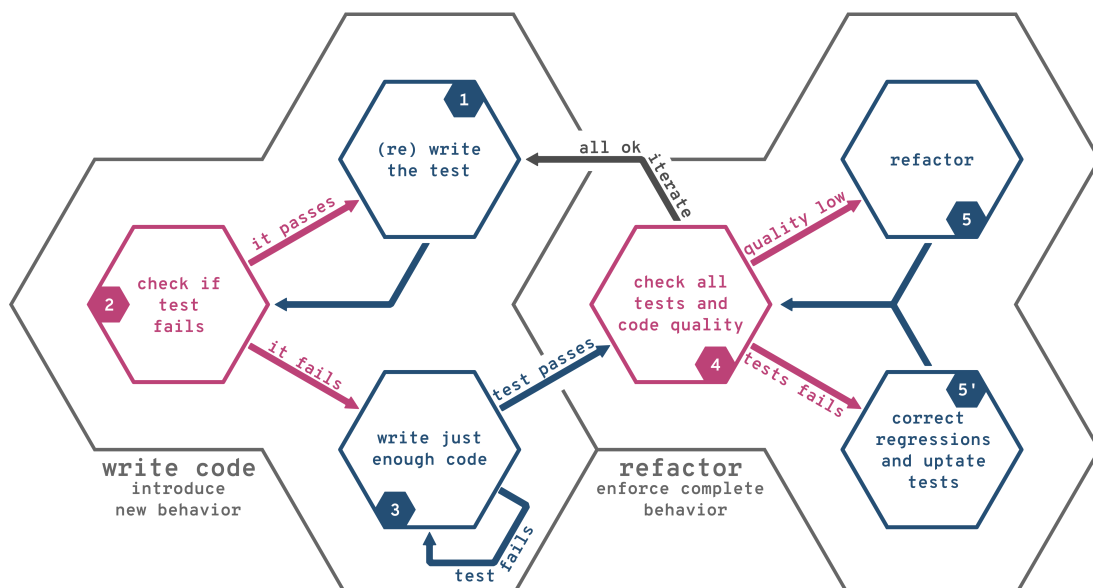

Figure 3: Test Driven Development overview

LEARNING SOFTWARE DEVELOPMENT

Elements presented before in this document are briefs selection and summary from all concerns, good practices and tools one shall have in order to produce high quality software with low efforts. Therefore, a lot is left for readers to learn, either by themselves or through taking courses. Regarding this matter, this section tries to give some directions on how to efficiently learn software development knowledge.

SKILLS PORTABILITY

In software development field, all tools or practices for a certain purpose tend to be very similar. Which means, once you've learned one from a category, learning others will be extremely easy. It also applies to programming languages. Once you know a programming language, most other languages will be trivial for you to learn.

Additionally, some tools and practices, as automation tools and version control, can be used outside of software development field.

Do not be afraid to spend a bit of time learning new tools when you wonder if you need them: it is rarely wasted time.

DO IT !

A lot of programming tools are language-based. Of course, programming languages are included in this category, but also *CMake*, *Make*, and any command line or config-file based tool.

In a graphical interface tool, as most public-oriented software, available actions are usually suggested to you through visual display, for example in a menu. This is convenient when you do not exactly know what you want to do or how you want to do it: menus serve as a quick and tiny documentation.
With language-based tool, it is the opposite: if you ignore how to do things, you will have to search and read external documentation on the syntax. On the other hand, if you know well the language of the tool, you will be extremely efficient when using it.

Efficiently using a language-based tool requires more practice than for a graphical tool, in order for you to become *fluent* in the language, enabling you to focus on what the tool should do and not how to tell it to. Also, struggling a bit with the syntax when learning a new syntax is perfectly normal. When following a tutorial or course, it is advised to reproduce all demonstrated manipulations to get a fair amount of practice.

SMALL FOCUSED STEPS

From language comes a lot of abstraction, and nearly everything in a software is represented using sole text, the source code. While it enables extremely efficient factorization and constructs, as mathematics do, it can also be difficult to manage for humans as we do not see directly the result of what we are doing.

To minimize this, one can do two things. First, split your project in small step. Each steps starts with a functional version of the software and ends with another functional version. Second, try to have your steps to be as specialized as possible. For example, if you want to improve automation in your projects, take steps with only automation improvement and no development in it.

It applies to both development and learning, since we often learn on the fly, by experimenting on a project.

FINDING KNOWLEDGE

Software development is not science-bound[1], and it is likely that you will acquire relevant knowledge on the Internet instead of through papers, and most information to be found are not sourced, and even less with scientific sources.

A lot of tools have official tutorials and documentation associated. These are to be prioritized when learning or solving issues, since they have the highest chance to be reliable and accurate.

When no documentation is available, or if it is insufficient, rest of Internet or books are also fairly good sources. Beware, there are a lot of inaccurate or context-tied information out there, think with criticism, especially before taking important decisions. In regard of this matter, you can always submit your doubts on a help forum and specify you want sourced answers.

While finding technical knowledge is fairly simple, complete software development courses are rare on the internet, and learning will often pass by following sparse tutorials. Regarding this issue, I wrote a software development course, presented in next section, providing most of the basic knowledge that a scientist could need to develop software.

[1] Well, it is, but not enough to fit my taste

MY COURSE AND SERVICES

COURSE

As mentioned in the previous section, I'm offering a software development course targeted at scientists, *Software Development for Scientists*. The course book and other materials are free to access at my website[2].

The book is designed to be sufficient for autonomous learning, with incremental examples to drive explanations, and covers various software development skills a scientist may want to learn. It also mentions numerous advanced notions to help reader find more specialized knowledge. Here is a quick overview of the course content.

Development environment explores tools used by software developers, providing basic knowledge in some of them, as well as development methods basics and concerns when distributing a software.

Programming presents basic and medium concepts used in programming, including variables, types, functions and classes. It also shows basics of software architecture and performance concerns. It is driven by a large example.

Code quality discusses practices to make code easier to read and write in the form of guidelines and tools to enforce them.

Testing shows principles to efficiently test a software, including Test Driven Development method and a wide example based on the one of Programming chapter.

[2] https://courses.felix-bertoni.fr/0423.html

Currently, the course uses Python as main language and shows small pieces of C++ to compares. In a later version, C++ will be used as main language.

MY SERVICES

Some people may find easier to learn if they have supervision or support when learning. Even if my course materials are designed to be self-sufficient for autonomous learning, I can provide teaching services. Such services are mostly targeted at institutions as laboratories, but are also available for individuals, and can take various forms, including but not limited to those listed below.

Lecturing and supervision deliver the course traditionally, with presentations and practical work. I try to make lectures as entertaining as possible.

Support makes me available for student to answer questions and help them in case they encounter problem in their autonomous learning adventure.

Specific editing is for you if you require not already planned modifications regarding an existing course, as addition of examples closer to your domain. It also applies if you want me to write a new, not already planned, course.

Prices are to be negotiated for each case specifically, as I try to keep them fair. It ranges from 25 to 60 euros per required hour of work, which includes execution of the service as well as eventual preparation. For example, an hour of lecture is to be considered as around five hours of work. Any material resulting from an order, as books, slides and examples, are published under free culture licenses, as Creative Commons or GPL.

Finally, ordering a service supports me in my work to provide free to access learning materials. Feel free to contact me if you are interested in either my course, my work in general, or my services.