Introduction to Software Development

Chris Brady
Heather Ratcliffe

"The Angry Penguin", used under creative commons licence from Swantje Hess and Jannis Pohlmann.

Warwick RSE

28/11/2018

Aims and some Vital Rules

Aims

- Introduce tools and methods that are useful for software development
  - Version Control
  - Testing
  - Documentation
  - Licensing

Software “design” and “development” are not really any different to “writing code”. They’re just usually at a more abstract level. It’s impossible to write even a slightly complicated code without some degree of design, even if this happens only as you write. The only real distinction is that the “design” encompasses much more, including how you write the code (which language, libraries, techniques), what the code should actually do (essential features, limitations etc). The “development” process also includes things like how to collaborate on code, how to debug it etc.

Setup/Disclaimer

- Software development is massive, changing field
  - Web technologies change yearly
  - Most languages add features every few year if not more
- Even “best practice” neither universal nor fixed
- You’ll never know everything:
  - Should spend lots of time saying “aha! I’m sure I read/ heard something about this. Now what was the key word to look for?”
  - Then you turn to a book index/search engine/colleague and find out what you need to know

We mostly stay away from specific languages here, only discussing interpreted vs compiled since they are written, debugged, tested, quite differently

Ultimate Rule

As researchers, your priority is research. Your code needs to work, and you need to know it works. You need to be able to do it again in ten* years. Anything that does not enhance that goal is decoration

*replace ten with whatever your funder/supervisor/ conscience dictates

This is really a rule.
Any sufficiently large or complex piece of code will have bugs - i.e. sometimes or somehow it doesn’t quite work. The trick is distinguishing the known knowns (bugs you know, understand, and can quarantine away from your work), the unknown knowns (bits you know might not work or shouldn’t be trusted) and the really risky one, the unknown unknowns (bits you don’t even know are suspect or in error). Testing and validation is all about trying to eliminate the last one. Rewriting, refactoring etc are about making it right afterwards.

Ultimate software rules

- Follow your language standards
- Have some kind of history of versions of your code
  - Version control
- Test that your code works. Don’t assume it
  - The more complicated the system the more important this is
- Document your code!

If you follow these rules then you’ll manage to miss about 90% of the pitfalls in software development. The other 10% still aren’t much fun though and you can’t avoid everything!

Version Control

Version Control

- Record changes that you make to a file or system of files
- Allows you to keep a log of why/by whom those changes were made
- Allows you to go back through those changes to get back to old versions
- Help deal with merging incompatible changes from different sources
- Similar term “Source Code Management”

One of our sort-of rules was to use some sort of Version Control. Some insist that this means one of the dedicated systems. We are slightly more flexible. It is possible, if you’re conscientious enough, to keep a systematic, time stamped version of code and scripts that does everything you need. But its hard. On the other hand, if you take no care at all version-control wont help you. It can’t.

Why Use Version Control?

- "I didn't mean to do that!"
  - Can go back to before you made edits that haven't worked
- "What did this code look like when I wrote that?"
  - Can go back as far as you want to look at old versions that you used for papers or talks
- "How can I work on these different things without them interfering?"
  - Branches allow you to work on different bits and then merge them at the end

Important note: all of these things are true IF you use your version control system carefully and "properly". These are things the systems allow you to do: they're not in general done for you.
This is true, but you can only go back to things which are "known" to the version control, and most rely on you to tell it when to take a snapshot.

Why Use Version Control?

- "I want a secure copy of my code"
  - Most version control systems have a client-server functionality. Can easily store an offsite backup.
  - Many suitable free services, and can easily set up your own
- "How do I work with other people collaboratively?"
  - Most modern version control systems include specific tools for working with other people.
  - There are more powerful tools to make it even easier too

Version control is not a backup! BUT most systems allow you to go "back in time" so it can protect you from accidentally deleting code from a file.
Version control systems can HELP you back things up, since most systems make it easier to sync multiple copies.

For collaborative *editing* version control is awful. Few, if any, systems are designed to let two people work on one file at the same time. Actual editing systems like Googledocs or Overleaf try and do this, but even they struggle. What version control can do is help keep changes separate, and help combine them together.

Why Use Version Control?

- "My funder* wants me to"
  - More and more funding bodies want code to be managed and made available online
  - Version control is a good way of doing it

* (Or supervisor, PI, Institution etc)

What Version Control Isn't

- Not a backup
  - If you use a remote server are safe against disk failure etc
  - But other people can still wipe out your work
- Not a collaborative editing tool
  - You can merge changes from many people
  - But it is hard work, not intended to handle editing the same files
- Not magic
  - Some language awareness, has to be conservative
  - Wont fix all your problems

Git

- Git is a very popular VCS system
  - Not the only one
  - Github is NOT git and git is NOT github
- Not going to walk through every command
  - See our full Git materials at https://warwick.ac.uk/research/rtp/sc/rse/training/introgit

We're not going to go through all the details of using Git here. We're just going to introduce some of the essential words and direct you to either our materials with examples at https://warwick.ac.uk/research/rtp/sc/rse/training/introgit, the December holiday workshop we'll be running, or the many and varied online resources. There's dozens of places you can get examples and tips, but if you don't have the core ideas in place first it can be very confusing.
Github, by the way, is one remote Git server. It has its own ways of working which aren't inherent to git, and it's not the only option by far. It's just a very popular one.

Create a repository

```
chris@Maximillian:~/demo$ git init
Initialized empty Git repository in /home/chris/
demo/.git/
chris@Maximillian:~/demo$
```

- Simply type “git init”
- Directory is now a working git repository
- Be careful about creating a git repository in a directory that isn’t the bottom of the directory tree!

A git repository is list of files that are under git control. All of the information that is needed is stored in a hidden directory called “.git” and that turns a simple directory into a git controlled directory. You can either create a git repository by “init”-ing a directory or by cloning a git repository from a remote server. NOT EVERY FILE IN A DIRECTORY IS AUTOMATICALLY IN THE REPOSITORY

Designate files for repository

```
chris@Maximillian:~/demo$ mkdir src
chris@Maximillian:~/demo$ touch src/wave.f90
chris@Maximillian:~/demo$ git add src/
chris@Maximillian:~/demo$
```

- Create a directory and put a file in it
- "git add src/" tells git to put the directory src and all files within it under version control
  - Not yet actually in the repository!
- I'm using Fortran because I'm a physicist
  - Works pretty well with almost any text based file
    - Best with things like C/C++/Fortran/Python that it understands
    - Can now work with Jupyter notebooks without showing you all of the guts

Technically "adding" a file adds it to the "staging area". Files in the staging area will be added to (or modified in) the actual repository when you next "commit" them

Add files to the repository

```
chris@Maximillian:~/demo$ git commit
```

- "git commit" will actually add the file to the repository
- Will open an editor to specify a "commit message"

This commits all files in the staging area

Add files to the repository

```
# Please enter the commit message for your changes.
Lines starting
# with '#' will be ignored, and an empty message aborts
the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#       new file:   src/wave.f90
```

- I'm using Vim. Default will depend on your system
- Generally git commit messages should follow standard format

Git commit message

```
First check in of wave.f90

wave.f90 will be a demo of using a "wave" type MPI cyclic transfer
0->1->2->3->4->0 etc. in order. This is inefficient and is shown
merely for teaching purposes
```

- First line is the subject. Keep it to <= 50 characters
- Second line should be blank
- Subsequent lines are the “body” of the message
- Should limit body lines to <=72 characters
- As many as you want, but be concise

After writing message

```
[master (root-commit) f55a639] First check in of wave.f90
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 src/wave.f90
```

- When you save the file and exit your editor git will give you a summary of what's just happened
  - In this case, it's created the file "wave.f90" as I wanted it to
- If you quit your editor without saving this cancels the commit
- "wave.f90" is now under version control, and I can always get back to this version

Editing wave.f90

```
PROGRAM wave

  USE mpi
  IMPLICIT NONE

  INTEGER, PARAMETER :: tag = 100

  INTEGER :: rank, recv_rank
  INTEGER :: nproc
  INTEGER :: left, right
  INTEGER :: ierr

  CALL MPI_Init(ierr)

  CALL MPI_Comm_size(MPI_COMM_WORLD, nproc, ierr)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

  !Set up periodic domain
  left = rank - 1
  IF (left < 0) left = nproc - 1
  right = rank + 1
  IF (right > nproc - 1) right = 0

  IF (rank == 0)
    CALL MPI_Send(rank, 1, MPI_INTEGER, right, tag, MPI_COMM_WORLD, ierr)
    CALL MPI_Recv(recv_rank, 1, MPI_INTEGER, left, tag, MPI_COMM_WORLD, &
        MPI_STATUS_IGNORE, ierr)
  ELSE
    CALL MPI_Recv(recv_rank, 1, MPI_INTEGER, left, tag, MPI_COMM_WORLD, &
        MPI_STATUS_IGNORE, ierr)
    CALL MPI_Send(rank, 1, MPI_INTEGER, right, tag, MPI_COMM_WORLD, ierr)
  END IF


  CALL MPI_Finalize(ierr)

END PROGRAM wave
```

Adding the changes

```
chris@Maximillian:~/demo$ git commit
On branch master
Changes not staged for commit:
  modified:   src/wave.f90

no changes added to commit
chris@Maximillian:~/demo$
```

- Not just “git commit” again!
- That tells me that I have a modified file, but it isn’t “staged for commit”
- Have to “git add” it again, then “git commit”
- Can have as many adds as you want before a commit. That is “staging” the files

This is why the staging area concept is important. wave.f90 is under git control so it *is* in the repository, but it isn’t in the staging area until you “add” it so it isn’t affected by “commit”

Adding the changes

```
[master 4a04f03] Add changes
 1 file changed, 37 insertions(+)
```

- Once again editor comes up
- Same commit message format
- Should describe the changes that you have made
- On saving the file in the editor see the same commit summary
  - Now telling me that it's added 37 lines

Remote git

- Other use of git is to push and pull files to and from remote servers
  - Like github
- Allows working with other people
  - Can lead to troublesome **merges** where you have to combine multiple people's work
- Plenty of tutorials on using github

If you want a starting point, the basic remote git commands are

git clone - Copy a git repository from a remote repository to this machine
git push - Send my changes to a remote repository
git pull - Get changes made by other people to a previously cloned repository

Testing

Testing

- As a general rule you want to get the right answer
- You want to **test** your code on known problems
  - Unit testing - Make sure that individual units (usually functions) work as expected
  - Regression testing - Test that entire code still works the way that it's supposed to

There are unit testing frameworks like pytest, pfunit, Aceunit etc. They allow you to "instrument" code so that individual units can be run from an external testing program. This has some advantages but you can unit test perfectly well without them. See https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks for a good list of possibilities

Testing

- Need to come up with sensible test problems
  - Not too easy - slightly broken implementations will pass
  - Need to test what you care about
  - Need to test everything that you care about
    - That includes things you thought couldn't go wrong
  - Tests generally grow with discoveries of new ways to go wrong

Writing good tests is generally very difficult and almost entirely dependent on what you are doing

Test Driven Development

- Write such a complete suite of tests that you can say “when the code passes all tests it works”
- Write the tests
- Write the code
- Keep modifying the code until it passes all tests
- Doesn’t always map well to academic and technical code

If you try to follow TDD methodology always keep in mind that sometimes you can be finding problems with your test suite rather than problems with your code.

Continuous Integration

- Overlap of testing with version control
- On some condition the version control server tests your code
  - Often when you ask your code to be merged into the main release
  - Often every time you push to the server
- Can be very useful at avoiding mistakes getting into a code base
- NOT MAGIC

CI is not really a magic bullet. It prevents people from accidentally putting bad code into your repository but it doesn’t do much that a simple test script doesn’t so long as you can trust your developers to always run the test scripts.

Documentation

Self-Documenting Code

- What is 86400?
- It’s the number of seconds in a day
  - You might remember this if you work with it a lot
- What about (51.1279, 1.3134)?
- That is the latitude and longitude of Dover
  - You will not remember this
- Number that appear without description are called magic numbers. AVOID THEM
- Put them in named variables that make sense

Not kidding here. Magic numbers make reading code very difficult and makes maintaining a code much harder. There are cases where they are forgivable, but codes that I write even have a named constant for the number of spatial dimensions we live in (normally considered to be 3)

Self-Documenting Code

- store 86400 in variable seconds_per_day
  - Variable documents itself!
- Function “solve_quadratic(a, b, c) “ pretty obvious
- Function parameter “input_file” pretty obvious
- “seconds”, “quadratic” and “file” insufficient
- logical variable called “flag” unhelpful and redundant
- This is NOT a substitue for “actual” docs

Note first that the quadratic is obvious to me (a physicist) and to many people, because the ax^2 + bx + c quadratic is “almost standard”. “input_file” is pretty clear IF the action of the function is clear (read_file, get_configuration etc) However these may (usually do) have other restrictions. Can ‘a’ be zero in the quadratic? How many roots are returned? What format should “input_file” be? All of these are a task for the “real” docs. What self-documenting code is about is making it easier for you/others to read your code, without having to try and remember what a function does. A parameter called “seconds” may as well be called “bob” in most contexts. If you think this “self-documenting” is a very fancy name for the “giving things useful names” taught in many beginner programming courses, you’re not wrong, this is just the equivalent for larger programs, where you might have classes, namespaces and many other ways to be helpful.

Documenting Code

- For libraries and helper functions, interface docs vital
  - What function/subroutine does
  - What each parameter means
    - In Python and similar, this is the perfect place to mention if parameters should have a particular type/class
  - What is returned
    - Again, in Python type languages, what type it is
- An example call, ideally with context

This is the stuff you find in library docs. E.g. https://docs.scipy.org/doc/numpy/reference/generated/numpy.core.defchararray.capitalize.html#numpy.core.defchararray.capitalize

Documenting Code

- Often nice to have interface docs both in the code, and separated out
  - Avoids drift - if you change the code, docs right there are easy to change at the same time
  - Easier to skim and check parameter names etc are correct
- Many tools will extract your docs and make HTML/ Tex etc from them
- Some also check signatures in docs against code

Drift between code and comments is always a bad thing, and is one reason why good code comments are about the “general what” and the “why” and not the details of “how”. The idea of self-documenting code is all about letting the code itself show what is happening, and making that easier to understand. That’s also the reason why we suggest leaving things like equations in a form matching a paper, even when this has minor inefficiencies, rather than optimising but reducing clarity.

Generally, “what” a block of code is doing should never change - but how it does it regularly can. “what” a function does should never change, but which of several equivalent methods it uses also can. This is also why “interface” docs are quite brief, and should usually omit any details other than limitations on parameters etc. More detailed docs, sometimes called “implementation” docs, are also really useful. Opinion varies whether these must be separate. I favour putting them in, but with a tag/ highlight/something to distinguish them.

Documentation Packages

- Sphinx widely used for python, JS
- JSDoc, Javadoc etc
- Doxygen great for C++, good for Fortran
  - Outputs HTML or LaTeX among other formats
- Many many more exist - see https://en.wikipedia.org/wiki/Comparison_of_documentation_generators
- Also several online services to build/host. E.g. readthedocs.org

(Most) options can tell you what things aren't documented (including function params), which is really useful. Can also generate class-hierarchy diagrams, and tell you some other things about interconnections. This relies on them parsing the actual source code though, so the ones that work this way can only handle a restricted set of languages. Others read only specially formatted comments - these work with any language but don't have those nice features.

Documentation Packages

```c
#include <stdlib.h>

void badfn()
{
  int i;
  int *ptr;
  for (i = 0; i< 10; i++){
    ptr = (int*) malloc(100 * sizeof(int));
  }
}
```


This code is not clear. It appears to be a bug, but it might(?) just be doing something sensible

Documentation Packages

```c
#include <stdlib.h>

/** This function will cause a memory leak
 Use only to test the memory leak checker*/
void badfn()
{
  int i;
  int *ptr;
  for (i = 0; i< 10; i++){
    ptr = (int*) malloc(100 * sizeof(int));
  }
}
```


Ah! It isn’t. it’s a deliberate bug to test a bug checker.

Code Licensing and Sharing

Licenses

- Sharing code with fellow researchers:
  - Put your name/date at the start of your files
- If you're using other people's libraries:
  - Check if they impose restrictions
  - "Download *this* from *here* to use"
- If distributing your code online, for example using Github, consider choosing a proper license.

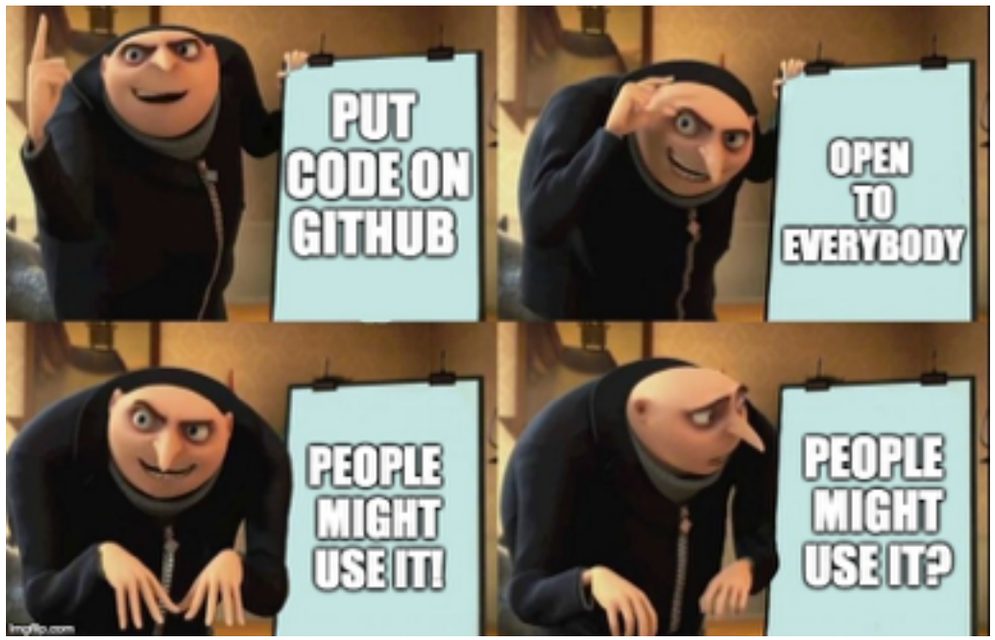
The simplest way to use libraries from other people is just to specify "Download this from here to use" rather than including their code. This doesn't get around their license, but would mostly be considered a fair use, as long as you attribute to them. You're not taking credit for their code, and crucially, you're not accidentally distributing some copy which may not be up-to-date or canonical.

Licenses

- https://choosealicense.com/
  - Describes most of the options
  - Helps you choose based on restrictions
- Primary concern is whatever your funder/institute demands you do
  - They may wish to retain rights
  - May require fully-open source
- Otherwise simplest option - protect yourself from unintended errors
  - https://choosealicense.com/licenses/mit/

Mostly you shouldn’t need to worry about licensing much. Either your funder has restrictions which you should know about, or they don’t. In the latter case, all you need to consider is whether you want to use something like the MIT license, or text to the effect of “By using this code you agree that anything it does, unintended or otherwise, is your own responsibility”. This protects you in the unlikely event that your code messes up somebody else’s work or computer.

Sharing Considerations



Meme created by Imgflip. Image sequence from the film https://en.wikipedia.org/wiki/Despicable_Me_(franchise)

Sharing Considerations

- Tempting to “release” all your code
  - Are you actually happy with somebody using it?
  - Does it work?
    - Is it free of silent failure cases?
    - Does it handle errors?
  - Is it documented?
    - Especially any assumptions!
  - Are you going to fix any issues you find?
    - Or at very least add them to documentation.

Be very careful if the answers to any of these are “no”. You don’t want unfinished or broken code getting away from you.

NOTE: this doesn’t apply to just popping code on somewhere like Github, only when you start presenting it as something for others to use. E.g. giving it a name, putting it on a Python repo, giving usage instructions

Closing Notes

Things to Take Away

- This is a very, very quick overview
- Lots of links for further reading from our longer intro to software development course https://warwick.ac.uk/research/rtp/sc/rse/training/introdevshort
- Pick the things you're interested by or need
- Let us know if there's anything missing
  - rse@warwick.ac.uk