Introduction to Software Development

Chris Brady
Heather Ratcliffe

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

Warwick RSE

23/10/2019

Aims and some Vital Rules

Aims

- Introduce tools and methods that are useful for software development
  - Debugging principles
  - Profiling
  - 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 (next time)
- 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!

Debugging Code

Bugs

- Definition: error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.” [Wikipedia (Software bug)]
- Not every example of a program “misbehaving” is a bug
- Sometimes there are several valid ways of doing something, or the programmer may have chosen a compromise.
- Compromises should be covered in code documentation. Make sure to consider this in your own code, as you may not remember your reasons in a few months time.

Bugs

- Bugs which you can reproduce are troublesome - bugs which you can't are awful. Not only are you working in the dark to diagnose it, you can never be sure it has gone!
- If you've ever had one fire alarm with a dying battery you can imagine this
  - Beep
  - ..... beep
  - .....
  - beep

Every time you think you hear it, it's gone, and you can't work out where it came from. Bugs like this are a nightmare!

Bug Catalogue

- We put together a rough “catalogue” of sorts of bug
- Copy at https://warwick.ac.uk/research/rtp/sc/rse/training/bugcatalogue.pdf
- Not exhaustive, but worth a read. It also covers a bit about numerics
- Bug spotting comes with experience. As you get to know the symptoms you can often “see” what the problem must be and then you just have to find it

Debuggers

- "**Symbolic" debuggers** understand your code. They know what symbols (variables, functions etc) it contains
- They let you step through the code one line at a time
- Can examine variables as you do
- Can set "**breakpoints**" - run until this point and then stop
- Learn to use them, they are invaluable

GDB is classic example for C/Fortran/C++ code. Python has pdb.
Important note: bad enough errors can crash the debugger, especially pdb. So sometimes you have to turn to other tools, or the method on the next slide.

If you are working with C/C++ or Fortran code called from within python (usually from a library) then you might need to use python extensions for gdb (see https://wiki.python.org/moin/DebuggingWithGdb) so that you can track where a crash in the compiled code was called from your python code.

Print or Caveman Debugging

- Sometimes, especially when working with parallel code, or when remotely debugging something it's not practical to use "proper" debugger
- Old method but it works
- Insert prints. Slowly restrict down to troublesome area and find bug
- Advantage - forces you to think about what is happening and what you expect to
  - Can work better when error is a logical one

Two really important things to remember.
First is practical - code output can be buffered, so if problem is a crash, you might have to take care to flush output to see your prints. See https://blogs.warwick.ac.uk/researchsoftware/entry/pretty_please_print/
Second is a real timesaver - **bisection**. Possibly the most important method you'll ever call on. When trying to find name in phone book, you don't start from the beginning. Flip to somewhere around the middle. Is the target before or after this? Now flip to middle of the correct chunk and repeat. Each time the number of pages drops by half. This is the core idea behind binary searches, several sorts of sorting algorithm and lots of more complicated stuff in data structures. For N items to search through, this way takes log_e(N) flips, rather than N/2 (on average you find your name after flipping through half of the phone book)

Symbolic Debugging

- Using symbolic debugger not much changes except how you work
- Set breakpoint on the failing line
  - Examine state of variables
    - Trace back anything which is wrong
      - Either bisection
      - Or careful stepping
  - Adjust (if possible) bad value
    - Step on and make sure things work

Handy Things

- Features of gdb and family to keep in mind
- Delayed breakpoints - break only on the *n*th time we run this line
- Conditional breakpoints - break only if *x* is true (e.g. i > 0)
- gdb can be attached to a running process - useful with parallel jobs (gdb -p ***PID***)
- Can edit a variable and continue running

Profiling

Profiling

- We mentioned “premature optimisation” as a red flag
- When it is time to optimise, how do you make sure you’re
  1. targeting the crucial (rate determining) parts
  2. making things faster/more efficient
  3. not breaking other things
- First two are job of a profiler, third is solved by testing and verification

The whole point of profiling is to try and work out which parts of your code are doing all of the work (quite often these are called “hot spots”) and then make those faster.

Profiling

- Imagine program of 2 functions called once each - FnA takes 9 s, FnB takes 1 s
  - Optimise FnB 100x faster to almost 0.01 s - overall time 90%
  - Optimise FnA to 8 sec (1.125 faster) - overall time 90%
- After optimising, rate-limiting function can change
  - Start with FnC of 50%
  - 2x faster -> overall 75%
  - 10x faster -> overall 55%
  - No point going further - diminishing returns

Profiling

- Profiler sits inside and around your code and monitors all or some of:
  - How many times functions are called
  - Where they’re called from
  - How long they take
- Shows where to focus optimisation effort
  - Target functions which make up much of runtime
  - Make them “fast enough”
    - Optimisation means trade-offs in clarity, simplicity etc

Note: just because a function is called a lot doesn’t mean you’ll gain from optimising it. Might be worth looking at **inlining** - just the function call can be substantial time sink

Parallel Profiling

- We often work with multi-core codes on clusters
  - Want to profile “as close to reality” as possible
- For MPI codes there is a free tool called MpiP
  - Replaces MPI calls with its own monitored ones, so can tell you call stats
- For trickier problems there are commercial tools
  - Arm/Allinea Map/DDT/Forge
  - Intel Advisor

Here are very much trading programmer effort for “real money” or at least allocation
The Arm (formerly Allinea) tools are available on some supercomputers and are very handy.
Intel Advisor tells you things about optimising your code like how well it vectorises

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