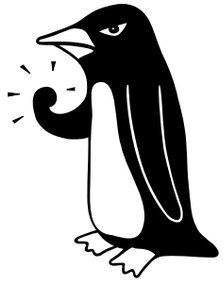
Introduction to Software Development



Warwick RSE

11/12/2017

Introduction

Aims

- Understand concepts in software design
  - NOT coding
- How to lay out a larger code
- Documentation and support
- IO and data sharing
- Code sharing and licensing

Disclaimer

- Should already know basic programming
- Taking practical stance: may not agree with textbooks
- Only say “must” if we mean it, everything else is “should”
- As researchers, your priority is research. You code needs to work, and you need to know it works. Anything that does not enhance that goal is decoration

Basic Design

Warning

- If your code involves any sensitive data, you really must read up on relevant regulations and protocol

- Includes:
  - Personal data
  - Credit card or other monetary data
  - Protected information (e.g. medical data)
  - Safety critical interactions

Design:
Structuring Code

- Essential Rules
- Guidelines
- Guidelines that don’t always work

Structuring Code

- Essential Rules
  - Create a working code - key thing is to end up with code that works, does what was intended, and is reasonably dependable, i.e. wont break unexpectedly or in hard to detect ways.
  - Follow your chosen **language standard** - standards dictate what is valid code and a valid program. E.g. in a valid Fortran program, all variable definitions must go at the top of a function. In a valid C program, all functions must be defined before use.
  - Use some sort of versioning - keep some sort of record of when changes were made (and by whom). Either disciplined dated backups or VCS (see later).
  - Validate your code enough - checking and testing is vital, but what "enough" means varies. For a one-off small script, you might just run an example case; for large projects you will want to consider formal testing and validation.

Structuring Code

- Useful Guidelines
  - Divide into reasonably sized functions - in general, functions should do one thing - can be complicated E.g. writing complex data to file. Not “X lines or less” But easier to write, and far easier to debug a function if you are not having to think about twenty things at once.
  - Consider laying out equations etc to help a human - formatting is gone once your code is compiled or hits the interpreter. Can use line breaks to split calculation into logical groups, align horizontally etc. Can keep equations as in book/papers. Can put human-readable equations in comments.
  - Document as you go - we’ll discuss documenting later: in general do it as you go, but never before a function’s purpose has been worked out. Orphaned comments are confusing.

Structuring Code

- Sometimes Useful Guidelines
  - Global variables - May hear “global variables are evil”. Truth: unexpected side effects are evil. For truly global state: carefully named, documented global variable is a solution.
  - You Aren’t Going to Need It (YAGNI) - In general, want simplest program that solves your problem. BUT do consider slightly what you’ll need from it next week or you’ll find yourself writing non-extensible code and wasting time. Don’t add every bell and whistle!
  - Do not optimise (at this stage) - until your code works, don’t bother with (most) optimisation. BUT do avoid making choices that can never be fast enough.
  - Don’t repeat yourself (DRY) - good general principle, don’t take to extremes. In general don’t copy-and-paste code and then make small edits. But many examples where things are subtly different and combining is worse than not.
  - “There’s only one way to do it” (Zen of Python) - sounds nice, breaks down a lot. Many ways to solve most problems: academic code has unusual concerns. Getting things to work, and to perform adequately is more important than any points of philosophy.

Design:
Planning Code

- Things to consider
- Language Standards
- Algorithm Selection

Planning Code

- Always plan before starting - but crux is ***to*** plan, not ***have*** the plan
- For small scripts, informal, not written down
- For large code, consider UML etc
- Flow chart is good start

Planning Code

1. Are any program sections dependent on others?

2. Where do you need user input?

3. Which sections are likely to be the crucial ones for testing and optimising?

4. Where does your code link to other code such as libraries?

5. Do any parts require research, study or unfamiliar libraries?

Language Standards

- Many languages have detailed language standards managed by ISO (International Organization for Standardization)

- Dictate what valid code is, what guarantees compiler or interpreter must make about how it implements function

- Code which does not conform to the standards is invalid. May not work on a different compiler. Extensions subject to change (e.g. Windows display support)

- Rule: do not write code that violates standards or has undefined behaviour

Selecting Algorithms

- Imagine sorting N things, e.g. essays by name
- First try: look over items; find smallest; put in place; repeat for new smallest.
  - **Time: N^2**
    - 10 items -> 100 units; 1000 items -> 1,000,000 units
- Better: split on first letter into 26 piles in order; split each of these into 26 on second letter; repeat until each pile is one item; combine
  - **Time: N *ln*(N)**
  - 10 items -> 23 units; 1000 items -> 6900 units

Selecting Algorithms

- Another example: finding one essay in sorted pile
- First try: hunt from start until located
  - **Time: N**
- Better: check middle item. Is target before or after? Now take relevant half and repeat.
  - **Time: *ln*(N)**
  - Key is bisection. Remember this idea
- Timings called “algorithmic complexity” - how long is taken at very large N

Selecting Algorithms

Principles:

- As simple as possible, but no simpler. Don't choose complex methods without need: trickier to understand, implement and test.

- Don't re-invent the wheel. Take advantage of prior art. But bicycle wheels don't fit jumbo-jets; and jet wheels are far more complex and costly to maintain than a suitable bicycle wheel.

- Don't pour good money after bad. Sometimes you will make a mis-step and spend time on unproductive routes. Be willing to put that work aside and try again. Don't throw it away though - it may be useful another time.

- Better the devil you know. If you know a technique that will work, but is a bit more computationally demanding, may still be a better choice than something unknown.

Design:
Other Considerations

- Don’t write anything
- Libraries
- Scoping and Processes

When Not to Write Software

Sometimes your plan concludes that you shouldn't write anything

- Need only simple, single-use script: don't overbuild
- Tool already exists that can do your task "well enough". E.g. data plotting where Excel will do
- Software already exists and is suitable, affordable etc
- Effort would not be balanced by the reward. Might be better taking a different research tack.
- You lack the expertise. May be better off adapting your research problem to the tools available, as you may not even realise the errors you are making.

Libraries

- Often a library exists to address (part of) your problem
- Powerful, saves work and avoids errors. But are drawbacks
- Left-Pad - micro dependency hell
- **Dependencies are complexity!** Often gets forgotten
- Be selective!

Project Scoping

- Projects grow to fit the time available for them
- Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law. – Douglas Hofstadter
- Trickiest part (?) of development, years to perfect.
- https://warwick.ac.uk/research/rtp/sc/rse/project_estimator.pdf or https://warwick.ac.uk/research/rtp/sc/rse/project_estimatorwexample. pdf
- Tries to quantify size and complexity based on inputs, outputs and libraries.

Waterfall and Agile

- First formal software design methodology: waterfall development
  - Design in stages, each fully and completely determined before next starts
  - Idea for critical programs handling people's data, money or safety
  - Detailed program specifications written & confirmed **before** code created
  - Code tested against these specifications
  - Nothing is released which does not meet the standards set
- Alternative school of methods: agile development
  - Come from industry: customer changes specs at will, requests new features close to deadlines, refuses to clarify their needs
  - Only pin-down what will be done in short chunks, often a week at a time
  - Regularly release working, incomplete code
  - Can work well in academia; minimises time spent on formal design, be sure to plan enough to make your code useful

From a Blank Editor

Prototyping

- Designing a physical object: start with prototype
  - Smaller, cheaper, easier-to-work-with materials
  - Mock functions
  - E.g. concept cars, rats-nest circuits, colour swatches on wall
  - Web pages: layered images, no code at all
  - Functional prototype: partial function to check ideas, trial methods, libraries etc

Minimum Useful Unit

- Smallest unit of program which is useful (surprise!), removing everything not totally essential
- E.g. program uses input file - ideally name is user-supplied but hard-coded is already useful. Eventually maybe want a file picker
- MUU: defines success of program or addition
  - Simplest code that will produce papers; any increment which allows you do produce a new paper
  - Whatever will satisfy your funder
  - Lets you get work done even while you develop your code further

Trialling

- **Write a simple program with new tool, before integrating into actual code**
- Like learning an instrument: start with handling, single notes or chords, scales
- Trialling libraries:
    - Create a small program with basic features you'll need
        - E.g. high-quality, repeatable random number generator. Create small program to seed generator & print a few random numbers, test
- Trialling algorithm:
    - Implement, test it on simple data, check result
    - If critical, check performance of algorithm and implementation
    - **Do not optimise this early but do throw away things that wont work**

Getting to Work

- Many ways to turn plan into code. Likely elements of:

    1. **Comment first** - write, in plain language, what each piece of code should do. Add detail until you know “exactly” what to code. You may want each line fully described, or you may be happy with comments such as “Write the data to the file”.

    2. **Files and functions** - create the files you’ll need, breaking problem code into chunks with shared purpose. E.g. “user interaction” module, object representing a physical object. Gradually code “main” routine, creating function signatures as they arise. Then start filling in layers of functions

    3. **Just do it** - simple scripts, problems you know how to solve, or problems you have no idea how to solve, can just sit down and start writing. Look up solutions as required.

        - No idea means you can’t plan. Any plan likely to fail, better to learn as you go. Already have to consider time to refactor/rewrite at the end

Patterns and Red Flags

Patterns

- Patterns: “language-independent models of robust, extensible solutions to common problems”
- Anti-patterns: “commonly reinvented but generally bad solutions”
- Don’t assume anti-pattern always bad code, but don’t jump to assuming your problem is special

Common Red Flags

- **Magic numbers** - numeric constants in code without comment.
  - What is 86400?
  - How many digits of pi will you need? Don't want to edit every occurrence
- **Global variables** - too many, or the wrong ones.
  - What if a function goes wrong? Is the global in valid state? Was it changed? Partly written (atomicity) i.e. nonsense?
- **The god object/class/function** - one thing that tries to do everything.
  - Functions should do their one thing well
  - Dividing into functions too harshly complicates code, kills performance
- **Golden hammer** - when all you have is a hammer, everything looks like a nail
  - Don't force favourite language/tool/technique to do everything
  - But don't spend time learning a new tool if yours is adequate

Common Red Flags

- Premature optimisation - optimising your code before it works, or at cost of readability/ modifiability/correctness
    - Don’t optimise before code works - doing the wrong thing faster is no use
    - Don’t make choices that make your code hard to make fast enough in the end
- Multiple return types - in e.g. Python
    - Returning dictionary with varying keys OK unless you expect particular keys
    - Returning array (many results) or single value (one) OK, may be more trouble than worth
    - Don’t return strings in some cases, numbers in others Consider class/struct containing both items. You may still make mistakes, but they’ll be less weird
- Reflection - allowing your code to read, modify and write itself
    - Very powerful in the right places, can also be dangerous
    - E.g. constructing a string you execute as code. Are you certain that the string is safe? In particular beware of ever executing user-input (see later)
    - Constructing object “on-the-fly” can be useful, but don’t know what members it has

Discussion:

Practical Design

Case Studies

- No detailed plan survives sitting down and actually writing
- Worse, in real situations you often don't have the luxury of using all of the best practices
- Examples for discussion:
  - some part of our ideals has to be relaxed
  - which parts to retain?

MoSCoW

- Common problem: want everything and the kitchen sink
- Don’t know how long things will take
- Have to rank features to get the most out without overrunning
- MoSCoW method Decide for each item/feature, if it:
  - **must** be included, i.e. without it the project is deemed a failure;
  - **should** be included, i.e. it adds real benefit and is strongly desired;
  - **could** be included, i.e. it would be nice to have
  - **wont** be included, i.e. it is not required at this time

Full Grant Not-a-Problem

- Large grant, program is the primary deliverable.
- Apply all the principles!
- Formally design the software
- Trial different algorithms to find the optimum
- Document everything

Create-a-Paper Problem

- Write some code in a general area in order to produce some good papers.
- Hard to formally plan
- Use agile development approach, plan in one-week or one-month blocks. Not the same as not planning at all!
- Informal about selecting algorithms
- Create code intending later to throw it away and write it afresh (refactoring and perhaps also rewriting): don't care much about style, patch together old code with snippets from e.g. Numerical Recipes
- Never relax on standards - undefined behaviour us always bad

Rapid Development Problem

- Code must be written now, either for a conference, a paper, or some other urgent endeavour. Shares elements with Create-a-Paper and Legacy-Necromancy
- Plan anyway!
- Style not so important
- Allow time for checking and testing

Legacy Necromancy Problem

- Have old code that used to work, needs updating or fixing and are tasked with just making it work.
- Don’t have time for a proper rewrite or refactor
- Can’t plan the structure and design, forced to work with what exists
- Don’t touch the bits that work!
- Start with testing?

The I-in-Team Problem

- You are part of some larger team. Somehow you must collaborate to produce a piece of software

- Left-hand must know what the right-hand is doing. Need to consider synchronisation, focus on who delivers what

- Consistency of style can be very helpful, so consider a style guide.

Coy Collaborator Problem

- Like the I-in-Team you are part of a larger team, but some members are unable or unwilling to share their source code and designs. Or perhaps project too big to read all their stuff
- Need to integrate your sections with theirs
- Again need to synchronise
- Focus on guarantees made by sections you may not view or edit. Document your interfaces

Documentation

Tools

- Tools exist that can read your source code and produce output you can display to users or developers

- Documentation describes what the function does, what the parameters are, and what is returned

- Most tools have some awareness of language syntax and can identify things like function parameters. Rely on you to provide a description of what these do etc.

- Can produce hierarchy graphs for classes, call graphs (what functions call which others).

- Generate Todo lists, defects lists etc

Tools

- Example: Doxygen (1.8.9)

```
data_array get_Bx ( std::string  file_prefix,
                    size_t       space_in[2],
                    size_t       time_0
                  )
```

Read reference B_x from file at path file_prefix, dump number time_0. If space_in is not [-1, -1], only the slice it dictates is read

Parameters

file_prefix File path

space_in Limits on x-dimension to slice out

time_0 The dump time to read

Returns

data_array containing bx data

Extension:

Add 3-D space handling!

Self Documenting Code

- Well written code needs less documenting and is much easier to work with
- **Does not mean you do not have to document your code!**
- Don’t try to completely describe functions: quickly get too wordy and complicated
- Think Principle of Least Surprise (PLS) and try to name things to save everybody time and effort, especially yourself
- Don’t repeat information already given by variable types, such as a logical flag, or the fact that a parameter is passed by reference
- Use common acronyms; use abbreviations and omit vowels if you like and your coding standard agrees
- Avoid ambiguous letters: 0 and O are easily confused, as are I, l and 1

Self Documenting Code

```
MODULE num
FUNCTION m(INTEGER n_v, ARRAY val)

FUNCTION av
FUNCTION gav

//These names are very short and neither descriptive nor memorable. It is hard to tell what
things are and what they do

MODULE io
FUNCTION prt (STRING str )
FUNCTION prtf (STRING str , STRING f , FLAG nl )

//Very very short names again. We know 'str ' is a string , but what is it for?

//Often function names here differ only by a single character
```

Self Documenting Code

```
MODULE numerical_functions
FUNCTION max( INTEGER n_values , ARRAY values )

FUNCTION average
FUNCTION geometric_average

//These names give some basic information about their functions

MODULE io
FUNCTION print (STRING text )
FUNCTION print_to_file (STRING text , STRING filename , FLAG new_line)

//Names are longer and more descriptive. The second string is clearly the filename to print into .

//The final flag variable is still quite terse , but we know this is some indicator about a new line

//Note we have no problem with naming the module io. Common acronyms save typing
```

Self Documenting Code

```
MODULE functions_for_finding_basic_numerical_results_by_jim

FUNCTION maximum_value_of_an_array(INTEGER
int_number_of_values_in_the_array , ARRAY arr_values )

FUNCTION average_of_two_numbers
FUNCTION geometric_average_of_two_numbers

//These names are long and irritating to type. The variable names repeat the type
information unnecessarily

MODULE input_and_output
FUNCTION print_text (STRING text_to_print )
FUNCTION print_text_to_named_file(STRING text_to_print , STRING
filename_to_print_to , FLAG print_new_line_at_end_of_text_or_not )

//The names are very long again and much information is redundant (both print text to
named file and filename to print to tell us the same thing )
//The flag name has become very long, and tells us what we (should) know from the type ,
that this is a true–or–false flag whether to do X or not
```

Hungarian Warts

- Hungarian notation or Hungarian “warts” on variables
- Suggested in the 70s: prepend type info to variable s_name
- In original form very useful, gave e.g. purpose info
- E.g. a string variable flagged as s_name vs us_name, “string” or “unsafe” string e.g. storing unvetted user input
- Decayed into repeating type information at the start of variables: string s_name or int i_index. C.f. Fortran implicit types
- Can be useful in languages like Python where type inferred for you, although beware if wart doesn’t match actual type

Documenting: What and How?

- Developer versus User
  - Those editing, extending or for library code using, your code
  - Normal users, provide inputs and get outputs
- Interface versus Implementation
  - What parameters your functions take, what they return
  - Internal assumptions, limitations, performance etc
- Detail level
  - Quick-start guide
  - Full use guide

Getting Data In and Out

IO:
Input

- Types and Strategies
- Validation
- Parallel Input

Input

- Consider purpose, running environment, flexibility of control
- From least to most flexible, with plenty of case dependence:

1. Hard coded values - e.g. a log file name
2. User prompts - e.g. for an output directory
3. File-per-control - e.g. read from specific filename, abort if a particular file is present
4. Environment variables and/or compiler flags - e.g. enable debugging output, control working precision
5. Command line options - e.g. specifying problem size or working directory
6. Simple config files - e.g. ini files, json files, Fortran namelists or key-value pairs
7. Input control systems - e.g. ability to specify maths expressions, automatic setup of a problem with given geometry, full GUI (graphical user interface) controls

1. 2. & 3.

1. Hard coded values

- Use for parameters you change only rarely
- E.g. log filename run.log in the specified working directory.

2. User prompts

- Use sparingly, offer alternative for somebody running your code via a script.

3. File-per-control

- Useful for e.g. abort of long running code. Check every-so-often for the presence of a file, e.g. STOP

4.

4. Environment variables and/or compiler flags

- Share some perils with global variables, change with caution
- Use for simple global config information
- Compiler flags invaluable for including or excluding debugging code
- Selecting a code-path at compile time for performance
- Can be used to select a variable type at compilation (such as float versus double)
- More code paths to test, do not overuse

5.

5. Command line options

- Lots of flexibility
- Perfect for programs invoked using a script
- E.g. -o option when you compile your code
- Problem sizes, input-file names and behaviour can all be controlled this way

6.

- All options so far lack **reproducibility**: you should help your users to preserve the information to reproduce their work in future

- Can output this information, or write it into your normal output files. But user still has to extract this and supply it to the new run

6. Simple config files

- Config files streamline this

- Many formats: name-value pairs, block-wise files e.g. old Windows .ini, complex nested structures e.g. XML or JSON files

- JSON libraries in many languages, good general option

- Consider tagging output with config or some sort of signature

7.

7. Input control systems

- Full input systems difficult, custom etc
- Good, robust GUIs hard, relatively uncommon for scientific codes.
- Often poor choice for a code running on a cluster, as you may have to wait for your job to schedule. GUI to create an input file?
- For very complicated codes may even consider scripting interface. Use Python, Ruby or Lua to setup and run your code
- Note: a live interface brings back the issue of parameter preservation unless you carefully output the configuration before running

Input Validation

- Always validate your inputs.
- Do not trust anything supplied by a user, even you
- Not because users cannot be trusted, but mistakes and ambiguities happen

Input Validation

- Example: code converts from one file format to another. Take two file names from the user, in and out. What happens if they give the same name for both? Will your code cope?

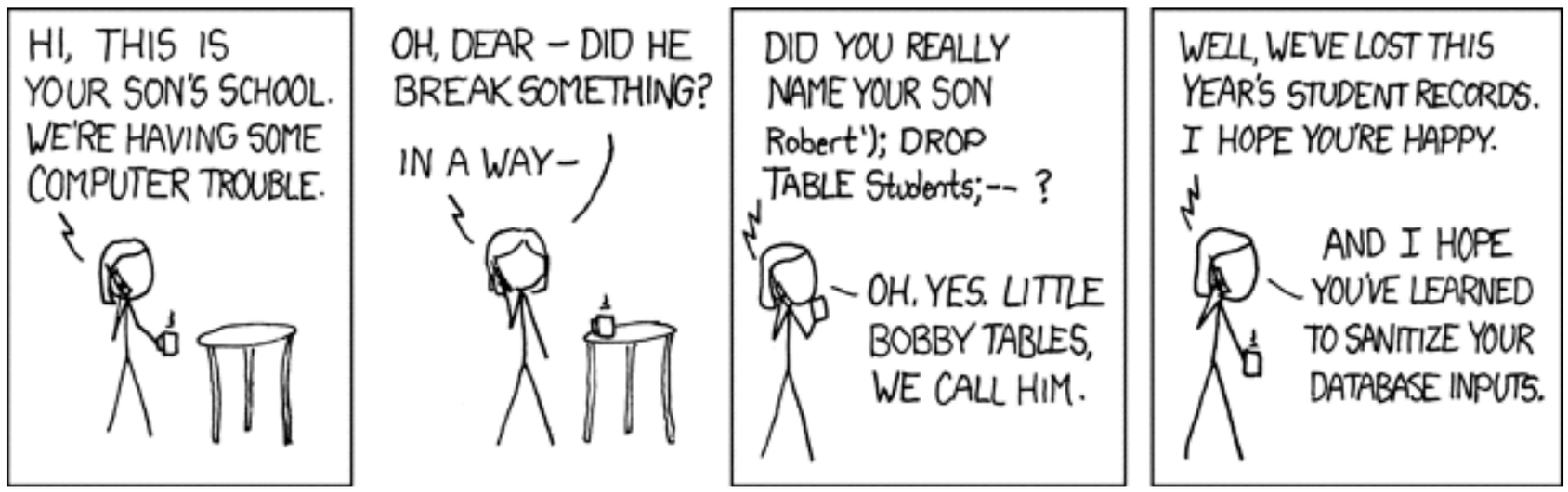
- What about if you ask for a number dictating an iteration count and the user gives you a negative value? Will you get an infinite loop?

- What if you ask for a size and the user specifies a very large number? Will your code eat all their memory and crash?

Input Validation

- Classic example: SQL injection. Cause of many high profile data leaks

- Read about this before you do it!



Input- Parallel Codes

- File-system locking: often only one process at a time can access a file: unexpected bad performance if every process attempts

- If large chunks of data, e.g. a restart file consider using MPI IO for collective reads.

- Alternatives (to MPI-IO):

  - read entire data set in on one processor and broadcast (needs enough memory for the entire set on every processor)

  - read chunks on one processor sending in turn to the relevant process (inefficient, one send per processor)

IO:
Output

- Types and Strategies
- Parallel Output
- Restart Files

Output

Consider size, purpose, lifetime of data files

1. File per variable - Temperature.dat, Density.dat
2. Fixed format files:

   10/11/2017

   10, 20, 13

3. Named block files:

   date

   10/11/2017

   temperatures

   10, 20, 13

4. JSON, XML, etc
5. Full data file formats, e.g. FITS, HDF5

Output

- Other considerations:

1. Do files need to be human readable? Binary files can be half the size of the equivalent text file, but you need to either know their content or use a specific binary format

2. Will the files be kept for long periods? Beware of custom formats that may change or disappear

3. Do the files need to be split or combined?

4. Will you share data files with others?

Output- Parallel Codes

- Several options for output
- Like with input but worse only one process can write to a file at once. May not want/be able to have each process write in turn
- The simplest alternative: file-per-process. Write one file on each processor, recombine during analysis
- For many processors slow, for very many can exceed the number of open files allowed by filesystem. Sysadmin may be upset
- MPI-IO (covered in our Advanced MPI sessions) can be used to perform writes collectively, or you can use one of the parallel aware file IO libraries such as HDF5

Output - Restarting

- Computers crash, power cuts happen etc
- Large programs, or long runs: can lose considerable data.
- Consider restart or checkpoint files: freeze the state of your program so you can stop and restart a job.
- Output state of all essential variables; probably in double precision
- Also any other state you want to re-initialise: e.g if your program generates random numbers must dump and re-init state so that stopped-and-restarted job is identical to continuous run

Sharing Code

Licensing - What and Why?

- Code is effort: want to preserve, use, share
- If created on Research Council time, check their rules (open source required?)
- If used to create papers, may need source to be available to readers
- Sharing code with fellow researchers: enough just to put name, institution, date into source files?
- Once you start distributing your code online, for example using Github, consider choosing a proper license

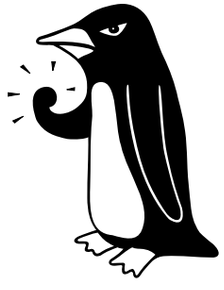
Licensing - How?

- See e.g. https://choosealicense.com/
- In general, you want to consider points such as:
  - Does my software use other software? What are its terms? Do I include their code in mine?
  - Do I want attribution when others use my code? What if they share it?
  - What about if they take it and edit it, producing their own programs they also share?
  - What if they profit from my work, directly or indirectly?
  - What if my code affects somebody's work or computer?
  - Does my funder require that I share my outputs? Or do they own them?

Resources

- Many Universities may provide hosting services for source code, such as a git server or personal FTP filespace
- Some also have contacts to help you choose licenses etc making sure you obey funder regulations
- Finally if your code has commercial value, your University may have resources to help you benefit from this. Warwick offer https://www2.warwick.ac.uk/services/ ventures/softwareincubator/
- If your work is funded by a research council you must read and obey any rules they have as to sharing your code and its results
    - For example, many funders require code be made available on request. Many journals require source-code used to create published data be shared. Be careful that you do not find yourself with incompatible restrictions in these cases

Introduction to Software Development



Warwick RSE

11/12/2017