You and your code

Charles Sutton

School of Informatics
University of Edinburgh

Why write programs?

- Want to know if your idea actually works!
- Typical case rather than worst case
- Especially true for “hard fields”
  - Artificial intelligence: All interesting problems
  - Compilers / Program analysis: Most problems Turing-complete
  - Machine learning: No free lunch theorem
- You don’t really understand an algorithm unless you can code it
- Feedback from error analysis drives new ideas

Theorists should program, too

But don’t listen to me (I’m not a theorist):

> Implementing your own algorithm is a good way of checking your work. If you aren't implementing your algorithm, arguably you're skipping a key step in checking your results.
>
> --Michael Mitzenmacher
> http://mybiasedcoin.blogspot.com/2008/11/bugs.html

Also, avoid attempting to prove false statements!

Software Engineering

- Software engineering is a rich area of research
- Commercial software development is
  - Expensive
  - Error-prone
  - Unpredictable
  - Incredibly important
- Many high-profile software failures, often caused by small looking bugs

Methodology

- Much (not all) SE research focuses on methodology
  - Requirements
  - Architecture
  - Design
  - Code
- The first three all have a documents attached. The code is expected to be heavily documented.

Why all this work?

- Customers do not know requirements initially
- Requirements change rapidly
- It is essential but often very hard to understand the domain
- Teams contain dozens or hundreds of developers, worldwide, over many years
- High turnover of software developers
- Documentation needed to avoid argument.
- Most of the cost is in maintenance

[adapted from Perdita Stevens]

Extreme Programming

- A reaction to the high overhead, bureaucracy of traditional approach (all those documents)
- Idea is to be *agile*
  - Do the simplest thing that will work
  - Test-driven development
  - Unit tests
  - Pair programming
  - Focus on refactoring
- Emphasis on quickly getting running code

Extreme Programming

- However, researchers lack the resources to do this all the way
- Some ideas useful to pull out (unit tests definitely, refactoring sometimes)

More info:

Books by Kent Beck, Martin Fowler

http://xprogramming.com/

These measures try to ensure product quality.

When you are a researcher, it is different:

The software is not your product

Your product is **knowledge**

- New algorithms
- New theorems
- New empirical characterization of existing algorithms

4 Principles of Research Code

1. Your code is a means to an end.
   Once the paper is done, you will throw it away.

Don’t really throw it away.
Disk is cheap; save *everything.*

4 Principles of Research Code

2. Unless you succeed.

Then everyone will want to try out your stuff.

If you are lucky,
you will start getting emails from
(PhD students of) famous professors.

Success is a pain in the a@#!

4 Principles of Research Code

3. You need to be able to trust your results.

You do not want to find a bug in your baselines after you publish!

4 Principles of Research Code

4. You need a bespoke set of tools.

Invent as little as possible (but no less!)

This is true both in your code and in your intellectual work.

N.B. All of these principles contradict each other.
Welcome to the real world!

Corollaries

- These principles should guide you in writing code for your research
- Except that they contradict each other
- So let’s see how the principles are reflected in real-world practice

Code is a means to an end

- Optimize for **your time**
  - Except when your research requires your code to be efficient (this happens)
    - Efficient code can be competitive advantage
- Think of documentation as notes to yourself
  - Push as much as documentation as possible into identifier names
- Always ask why you are writing the code... How will this fit into the eventual paper

Code is a means to an end

- Do not think that you know now where you will end up
- Optimize also for **flexibility**
- The only design pattern that I like is the strategy pattern
  - Wherever possible, think about where you might want to be clever later
  - Design for your later cleverness
- You care about bugs that affect your experimental results, not really about other bugs

Unless you succeed

- Commit to reproduceable research
  - Bench scientists have a lab notebook
  - You need the equivalent
  - **You must use version control!**
  - Organization of files, directories
  - “Connect” your code to the figures in the paper

Version Control

- Many examples: RCS, SCCS, CVS, SVN, hg, git, arch
- For new projects you should use at least Subversion, consider the distributed system (e.g., hg, git)
- I version control everything that I care about
  - My code, my research publications, my personal bib file, my Web site
- If you ran an experiment on May 13 that you need to reproduce two years later, reproduce that exact state

Version Control

- Every file that is
  - Text
  - Related to your research
    - Cannot be reproduced automatically
- ...should be under version control
- **To do otherwise is courting disaster**

Keep track of parameters

- Your experiments will have many parameters to tune
  - Which schedule algorithm you used
  - Max queue size
  - Which heuristics for X, Y, Z
  - Learning rate
- Make these parameters of the “experiment running scripts”
- Keep track of them in an organized way

Connecting your paper to your results directories

- Bench scientists have a lab notebook. You need something similar
- You need a system where
  - if you open up your paper file a year later
  - you will know how to rerun all the code needed for Figure 3
- Key idea: Generate figures automatically via script
  - Then you just need a map Figure => script file
  - Could use text file
  - Comments in LateX code
- Also need the version of your code
  - Keep tarball
  - Use version control

Example System

To start an experiment, run a shell script

```
sh scripts/test-tiny-heuristic.sh
```

This script contains the parameters of the experiment:

```
scripts/test-tiny-heuristic.sh

   DIR=results/tiny-heuristic/airlockF/al2.7/
   java uk.ac.ed.inf.csutton.HalMain \
     --open-airlock FALSE \
     --alpha 2.7 \
     --output-dir $DIR
```

Automating your driver script

- Lots of times you will need to run a “parameter sweep”
- Helpful to have a “driver script” that runs through the parameter grid
- Interacts well with grid/cloud computing: Run every parameter setting in parallel

Principles of experiment running scripts

- 1. Automate as much as possible
- 2. Record as much as possible
  - Text files are your friend!
  - Some of my fancy systems friends use databases
  - Makefile's are good for recording preprocessing steps
    - (use the filesystem to keep records for you)
    - Another possibility (haven't tried it): http://neuralensemble.org/trac/sumatra

```
hg/
  projects/
        qneta/
            papers/
                nips10/
                icml11/
            experiments/
                synthetic-expt/
                    code/
                    scripts/
                real-data-expt/
        results/
            synthetic-expt/
                graphs/
            real-data-expt/
                graphs/
```

Notes about dir structure

- Typically for me a “project” is a paper or two worth of work. (May take more than one submission.)
- Never quite sure where to put the results/ directory. Parallel to experiment dir? Underneath? Outside of hg altogether
- **Important**: Everything in results/ automatically generated
- code/ ==> implementation of the algorithms
- scripts/ ==> stuff that only matters for the experiments I happened to run

Notes

- There are other ways to do this
- I can never decide myself what's best
- The principle is:
  - If a year from now you look at the results file alone you can tell:
    - What code you used to generate it
    - With what parameters

Trust your results

- *Unit tests* are great
  - Test a small part of your code
  - The test code itself checks success, not human
  - They should be fast to run, so you can run them after every major change
  - Nip regression in the bud

Testing is hard

- Testing and debugging research code can require ingenuity
  - Often your algorithm is heuristic or approximate
  - If you expect it to fail 10% of the time, how can you be sure it’s correct?
  - Especially true in machine learning, etc

Creative testing

- Look for special cases
  - Maybe if you set alpha = 0, your method reduces to a simpler one
  - Special case on which your approximation is exact? e.g., Beam search on small state space
- There's more than one way to do it
  - e.g., Check different approximation algorithms against each other

Bespoke tools

OS, Standard library, etc

Domain frameworks: BLAS, Matlab, R, etc.

Your personal shared code

Research area A

Research area B

Experiment 1

Experiment 2

Experiment 3

Experiment 4

Making a toolkit

Many times you see researchers making software that many people use. A few examples:

- TinyOS http://www.tinyos.net/
- Jikes (research Java compiler) http://jikes.sourceforge.net/
- SVMlight http://svmlight.joachims.org/ (machine learning)
- Festival (speech synthesis) http://www.cstr.ed.ac.uk/projects/festival/
- Moses (machine translation) http://www.statmt.org/moses/

Making a toolkit

Why would you do this? Seems to violate every principle so far.

- People more likely to use your ideas if they don’t have to implement them
- Shared infrastructure: you can try ideas quickly
- Other people can add components which you then use
- Fame

In any case, this is calculated risk.

Summary

- Remember the four principles
- Your code is an investment of your time
  - Too little: Can’t trust results, Can’t reproduce
  - Too much: You try too few new ideas
- Writing efficient code can give you a competitive advantage as a researcher
- Good luck finding the balance!