

Software Engineering for Economists^{*}

Building Confidence in a Model

- Computational models of socio-economic phenomena are a manifestation of our perceived knowledge about the underlying processes. The key question is how much confidence should we have in a particular model?
- It turns out to be useful to structure such a discussion around three interrelated questions ?.
- Software Engineering encompasses the tools and methods for defining requirements for designing, programming, testing, and managing software. It is crucial to ensure that the computational implementation is a faithful representation of the original mathematical model ?. Thus, it is part of the verification step.
- As an aside, for those interested in structural microeconometrics, we were lucky enough to have Prof. Keane talk about the process of developing, estimating, and validating in the *Computation Economics Colloquium*.
- Basic software engineering allows free cognitive resources that we can use to expand the set of possible economic questions we can address responsibly.
- Computational implementation is part of the scholarship.

^{*}For further information or questions and suggestions, please contact us at info@policy-lab.org.

Research Example

Running Example

- For the rest of this lecture, we will use a small examples to illustrate ideas of different software engineering tools. However, we will also have a brief look how these tools are applied in the more complex setting of my current research. The online code repository is available online.

Goal of the Lecture

- The ideas and tools are most powerful when they are all acting in concert. However, as a word of caution: *A Fool with a Tools is still a Fool.*

Version Control

- *Flexibility* refers to moving across different machines.

Testing

- To see these basic ideas in action, let us check out the testing harness for my current research project online.
- Using bugs to define test cases ensures that they only need to be fixed once.
- Test generation for ?, efforts to control randomness.

Code Review

- We check for comments regarding our *epy* package, we will fix them later when talking about continuous integration workflow.
- Let us check out other projects' reports and the list of code patterns. They also published a *Knowledge Base* for best practices.

Continuous Integration Workflow

- By running the testing harness early and often, bugs are caught closer to their creation. This makes debugging much easier.

- Scalability of research team is improved as basic quality assurance is automated.
- The badges signal to your fellow researchers that we take your responsibilities as a developer of research software serious.
- Reliable work-flow increases own satisfaction.
- Fix problems of code quality, check notifications, students update their cloned *GitHub* repository.

Profiling

- Now that we have a well designed and tested version of our code in place and established a robust workflow, it is time address any performance issues. We will profile our program by measuring the execution time of the program.
- Profiling tools also measure the time spend in each function allowing us to target our development efforts at particularly time-consuming parts of the code.
- Studying the output directly can be rather tedious for large programs. That is when visualization tools turn out very useful. We build on SNAKEVIZ.
- For even more advanced visualization, check out pyprof2calltree. Tutorial for advanced visualization using KcacheGrind.

Best Practices

- Iterative project development with only incremental addition of features. Testing harness ensures that old features are not broken.

References