Embrace Your Fallibility: Thoughts on Code Integrity

Nick Eubank*

June 6, 2016

Two years ago, I wrote a piece about my experiences over two years testing the code for papers being published in the Quarterly Journal of Political Science, which found problems in the code of many papers. The piece was first published in The Political Methodologist, and later in PS: Politics and Political Science. This piece is an extension of that article based on conversations that article sparked and my own experiences over the past two years.

It's natural to think that the reason we find problems in the code behind published papers is carelessness or inattention on behalf of authors, and that the key to minimizing problems in our code is to be more careful. The truth, I have come to believe, is more subtle: humans are effectively incapable of writing error-free code, and that if we wish to improve the quality of the code we write, we must start learning *and teaching* coding skills that help maximize the probability our mistakes will be found and corrected.

I myself once firmly believed the fallacy that the key to preventing errors was "to be more careful." Indeed, I fear this belief may have colored the tone of of my past work on this subject in unproductive ways. Over the last few years, however, my research has brought me into close contact with computer scientists, and I discovered that computer scientists' mentality about programming is fundamentally different from the mental model I had been carrying around. Computer scientists *assume* programmers will make mistakes, and instead of chiding people to "just be careful," they have developed a battery of practices to address the problem. These practices – often referred to as "defensive programming" – are designed to (a) minimize the probability mistakes occur and (b) maximize the probability that mistakes that do occur are caught.

If we as social scientists wish to continue adopting more and more computational techniques in our research, I feel this is a mentality we must also adopt. This will not always be easy. Defensive programming is a skill, and if it is to become a part of the discipline it will require effort on behalf of researchers to learn, implement, and most important teach these skills to the

*nicholaseubank@stanford.edu. Post-Doctoral Fellow in Political Science at Stanford University. The author is grateful to Adriane Fresh, Simon Ejdemyr, Darin Christensen, Dorothy Kronick, Julia Payson, David Hausman, and Justin Esarey for their comments and contributions to this piece.

next generation. But I think this is necessary to ensure the integrity of our work.

With that in mind, I would like to advocate for two changes to our approach to the computational component of social science.

First, I think we must adopt a number of practices from defensive programming in our own code. This piece lays out a few simple practices that I think are most applicable and practical for social scientists, both for individuals and co-authors working collaboratively. They aren't meant as complete tutorials, but rather as illustrations of the type of practices I think should be promoted.

Second, I think we need to begin teaching these practices to students. Too often, students are either expected to figure out how to program on their own during their econometrics classes, or they are offered short, graduate-student-led workshops to introduce basic skills. Coding is now too central to our discipline to be given this second-tier status in our curriculum. If we are going to expect our students to engage in computational research, it is our obligation to equip them with the tools they need to stay out of danger.

Together, I think these two changes will improve the integrity of our research as coding becomes ever more central to our discipline. Will they preclude errors completely? Unlikely – even when perfectly employed, "defensive programming" is not fool-proof, and there will always be problems that these tools will not catch. But at least with these tools we can start to minimize the likelihood of errors, especially large ones.

This piece is organized into five sections. Section 1 presents an overview of specific defensive programming practices we can all implement in our own code. Section 2 then lays out some examples of how "defensive programming" principles can guide workflow in collaborative projects. Finally, after introducing these concrete skills, I offer a few reflections on the implications of the "defensive programming" paradigm for third-party review of code by academic journals in Section 3, and for how the discipline responds to errors in Section 4. Section 5 concludes with a short list of other resources, to which additional suggestions are welcome!

1 Defensive Programming Practices

Defensive Programming Practice 1: Adding Tests

If we could only adopt one practice to improve the quality of our code, my vote would be for the addition of tests.

Tests are simple true-false statements users place in their code. A test checks for a certain condition (like whether the sample size in a regression is what you expect), and if the condition is not met, stops your code and alerts you to the problem.

Right now, many users may say "Yeah, I always check that kind of stuff by hand when I'm writing my code. Why do I need to add tests?"

The answer is four-fold:

- **Tests are executed *every time your code is run.*** Most of us check things the first time we write a piece of code. But days, weeks, or months later, we may come back, modify code the occurs earlier in our code stream, and then just re-run the code. If those changes lead to problems in later files, we don't know about them. If you have tests in place, then those early changes will result in an error in the later files, and you can track down the problem.
- **It gets you in the habit of always checking.** Most of us only stop to check aspects of our data when we suspect problems. But if you become accustomed to writing a handful of tests at the bottom of every file – or after ever execution of a certain operation (I'm trying to always including them after a merge), we get into the habit of *always* stopping to think about what our data should look like.
- **Catch your problems faster.** This is less about code integrity than sanity, but a great upside to tests is that they ensure that if a mistake slips into your code, you become aware of it quickly, making it easier to identify and fix the changes that caused the problem.
- **Tests catch more than anticipated problems.** When problems emerge in code, they often manifest in lots of different ways. Duplicate observations, for example, will not only lead to inaccurate observation counts, but may also give rise to bizarre summary statistics, bad subsequent merges, etc. Thus adding tests not only guards against errors we've thought of, but may also guard against errors we don't anticipate during the test writing process.

Writing Tests

Tests are easy to write in any language. In Stata, for example, tests can be performed using the `assert` statement. For example, to test whether your data set has 100 observations or that a variable meant to hold percentages has reasonable values, you could write:

```
* Test if data has 100 observations
count
assert `r(N)'==100

* Test variable percent_employed has reasonable values
assert percent_employed > 0 & percent_employed < 100
```

Similarly in R, one could do the same tests on a data.frame `df` using:

```
# Test if data has 100 observations
stopifnot(nrow(df)==100)

# Test variable has reasonable values
stopifnot(df$percent_employed > 0 & df$percent_employed < 100)
```

Defensive Programming Practice 2: Never Transcribe

We've already covered tricks to maximize the probability we catch our mistakes, but how do we minimize the probability they will occur?

If there is anything we learned at the *QJPS*, it is that authors should never transcribe numbers from their statistical software into their papers by hand. This was *easily* the largest source of replication issues we encountered, as doing so introduced two types of errors:

- **Mis-Transcriptions:** Humans just aren't built to transcribe dozens of numbers by hand reliably. If the error is in the last decimal place, it doesn't mean much, but when a decimal point drifts or a negative sign is dropped, the results are often quite substantively important.
- **Failures to Update:** We are constantly updating our code, and authors who hand transcribe their results often update their code and forget to update all of their results, leaving old results in their paper.

How do you avoid this problem? For LaTeXusers, I strongly suggest tools that export `.tex` files that can be pulled directly into LaTeXdocuments. I also suggest users not only do this for tables – which is increasingly common – but also statistics that appear in text. In your code, generate the number you want to cite, convert it to a string, and save it as a `.tex` file (e.g. `exported_statistic.tex`). Then in your paper, simply add a `\input{exported_statistic.tex}` call, and LaTeX will insert the contents of that `.tex` file verbatim into your paper.

Directly integrating output is somewhat harder to do if you work in Word, but is still feasible. For example, most packages that generate `.tex` files also have options to export to `.txt` or `.rtf` files that you can easily use in Word. `write.table()` in R or `esttab` in Stata, for example, will both create output of this type you can put in a Word document. These tools can be used to generate tables can either be (a) copied whole-cloth into Word by hand (minimizing the risk of mis-transcriptions that may occur when typing individual values), or (b) using Word's Link to Existing File feature to connect your Word document to the output of your code in a way that ensures the Word doc loads the most recent version of the table every time Word is opened. Some great tips for combining R with Word can be found here.

Defensive Programming Practice 3: Style Matters

Formatting isn't just about aesthetics, it also makes it easier to read your code and thus recognize potential problems. Here are a few tips:

- **Use informative variable names.** Don't call something `var212` if you can call it `unemployment_perce`
  Informative names require more typing, but they make your code so much easier to read. Moreover, including units in your variables names (percentage, km, etc.) can also help avoid confusion.
- **Comment!** Comments help in two ways.
  - First, and most obviously, they make it easy to figure out what's going on when you come back to code days, weeks, or months after it was originally written.

  - Second, it forces you to think about what you're doing in substantive terms ("This section calculates the share of people within each occupation who have college degrees") rather than just in programming logic, which can help you catch substantive problems with code that may run without problems but will not actually generate the quantity of interest.
- **Use indentation.** Indentation is a way of visually representing the logical structure of code – use it to your advantage!
- **Let your code breathe.** In general, you should put a space between every operator in your code, and feel free to use empty lines. Space makes your code more readable, as illustrated in the following examples:

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

A full style guide for R can be found here, and a Stata style guide can be found here.

Defensive Programming Practice 4: Don't Duplicate Information

Tricks to minimize the probability of errors often require a little more sophisticated programming, so they won't be for everyone (tests, I feel, are more accessible to everyone). Nevertheless, here's another valuable practice: **Never replicate information**.

Information should only be expressed once in a file. For example, say you want to drop observations if the value of a set of variables is greater than a common cutoff (just assume this is something you want to do – the specific operation is not important). In Stata, for example, you could do this by:

```
drop if var1 < 110
drop if var2 < 110
drop if var3 < 110
```

And indeed, this would work. But suppose you decided to change that cutoff from 110 to 100. The way this is written, you've opened yourself up to the possibility that in trying to change these cutoffs, you may change two of these but forget the third (something especially likely if the uses of the cutoff aren't all in exactly the same place in your code).

A better way of expressing this that avoids this possibility is:

```
local cutoff = 110
drop if var1 < `cutoff'
drop if var2 < `cutoff'
drop if var3 < `cutoff'
```

Written like this, if you ever decide to go back and change the common cutoff, you only have to make one change, and there's no way to make the change in some cases but forget others.

2 Collaboration

Until now, the focus of this piece has been on *individual* coding practices that minimize the risk of errors. But as social science becomes increasingly collaborative, we also need to think about how to avoid errors in collaborative projects.

In my experience, the way most social scientists collaborate on code (myself included, historically) is to place their code in a shared folder (like Dropbox or Box) and have co-authors work on the same files.

There are a number of problems with this strategy, however:

1. Participants can ever be certain about the changes the other author has made. Changes may be obvious when an author adds a new file or large block of code, but if one participant makes a small change in an existing file, the other authors are unlikely to notice. If the other authors then write their code assuming the prior coding was still in place, problems can easily emerge.
2. There is no clear mechanism for review built into the workflow. Edits occur silently, and immediately become part of the files used in a project.

I am aware of three strategies for avoiding these types of problems.

The first and most conservative solution to this is **full replication,** where each author conducts the full analysis independently and authors then compare results. If results match, authors can feel confident there are no problems in their code. But this strategy requires a massive duplication of effort – offsetting many of the benefits of co-authorship – and requires both authors be *able* to conduct the entire analysis, which is not always the case.

The second strategy is **compartmentalization**, in which each author is assigned responsibility for coding specific parts of the analysis. Author A, for example, may be responsible for importing, cleaning, and formatting data from an outside source while Author B is responsible for subsequent analysis. In this system, if Author B finds she need an additional variable for the analysis, she ask Author A to modify Author A's code rather than making modifications herself. This ensures responsibility for each block of code is clearly delimited, and changes are unlikely to sneak into an Author's code without their knowledge. In addition, authors can also then review one another's code prior to project finalization.[1][2]

[1] Note that the separation of responsibility does not need to be as crude as "cleaning" and "analysis" – this strategy simply requires that a single person has clear and sole responsibility for every line of code in the project.

[2] Another intermediate strategy – which can be combined with compartmentalization – is to **maintain a change log** where authors record the date, files, and line-numbers of any changes they make. This eliminates the problem of edits going unnoticed. However, it is worth noting that this strategy only works if both authors are sufficiently diligent. If either (a) the author making changes fails to log all changes or does not describe them well, or (b) the reviewing author fails to go back into the code to check all the changes reported in the change log, the system may

The final strategy is to **use version control**, which is by far the most robust solution and the one most used by computer scientists, but also the one that requires the most upfront investment in learning a new skill.

"Version control" is the name for a piece of software specifically designed to manage collaboration on code (several exist, but `git` is by far the most well known and the only one I would recommend). Version control does several things. First, as the name implies, it keeps track of *every* version of your code that has ever existed and makes it easy to go back to old versions. This service is often provided by services like Dropbox, it is much easier to review old versions and identifying differences between old and new versions in `git` than through a service like Dropbox, whose interface is sufficiently cumbersome and most of us never use it unless we accidentally delete an important file.

What really makes version control exceptional is that it makes it easy to (a) keep track of what differs between any two versions, and to (b) "propose" changes to code in a way that other authors can easily review before those changes are fully integrated. If Author A wants to modify code in version control, she first creates a "branch" – a kind of working version of the project. She then makes her changes on that branch and propose the branch be re-integrated into the main code. Version control is then able to present this proposed change in a very clear way, highlighting every change that the new branch would make to the code base to ensure no changes – no matter how small – go unnoticed. The author that made the proposed changes can then ask their co-author to review the changes before they are integrated into the code base. To illustrate, Figure 1 shows an example of what a simple proposed change to code looks like on *GitHub*, a popular site for managing `git` projects online.

Version control is an incredible tool, but it must be noted that it is not very user friendly. For those interested in making the jump, the tool to learn is `git`, and you can find a terrific set of tutorials from Atlassian here, a nice (free, online) book on git here, and a very nice, longer discussion of `git` for political scientists on The Political Methodologist here.

In addition, there are also two projects that attempt to smooth out the rough edges of the `git` user-interface. Github Desktop, for example, offers a Graphical User Interface and streamlines how `git` works. Similarly, git-legit mimics the changes Github Desktop has made to how `git` works, but in the form of a command-line interface. These services are fully compatible with normal `git`, but learning one of these versions has the downside of not learning the industry-standard `git` interface. For researchers who don't plan to engage in contributing to open-source software or get a job in industry, however, that's probably not a huge loss.

still fail.

Figure 1: `git` Pull Request on GitHub

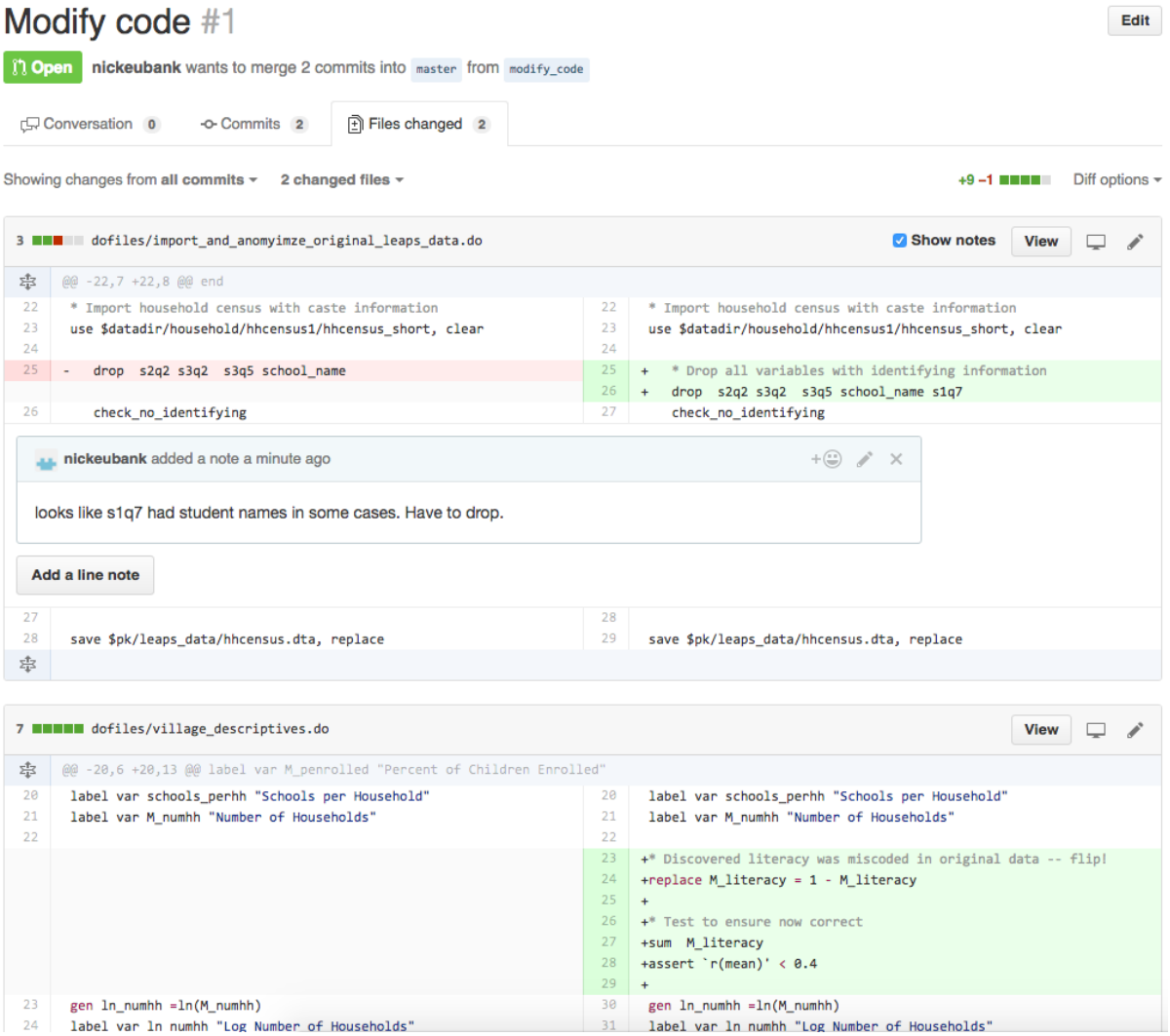

Figure shows an example of a small proposed change to the code for a project on *GitHub*. Several aspects of the interface are worth noting. First, the interface displays *all* changes and the lines just above and below the changes across all documents in the project. This ensures no changes are overlooked. (Authors can click to "unfold" the code around a change if they need more context.) Second, the interface shows the prior contents of the project (on the left) and new content (on the right). In the upper pane, content has been changed, so old content is shown in red and new content in green. In the lower pane, new content has just been added, so simple grey space is shown on the left. Third, authors can easily comment (and discuss) individual lines of code, as shown here.

3 Third-Party Code Review by Journals

As the discipline increasingly embraces in-house review of code prior to publication, one might wonder whether this is necessary. I am a strong advocate of Third-Party Review, but I think it is important to understand its limitations.

First, journals that conduct systematic review of replication code like *QJPS* and more recently the *AJPS* can only conduct the most basic of reviews. At the *QJPS*, in-house review only consists of ensuring the code is well-documented, that it runs without errors, and that the output it generates matches the results in the paper being published. Journals simply do not have the resources to check code line-by-line for correctness.

Second, even if Third-Party Review protects the integrity of the discipline, it does nothing to protect individual researchers. Appropriately or not, we expect researcher's code to be error free, and when errors are found, the career implications can be tremendous. Indeed, it is for this reason that I think we have an obligation to teach defensive programming skills to our students.

Finally, even detailed Third-Party Review is not fool-proof. Indeed, the reason writing tests has become popular in computer science is a recognition of the fact that people aren't built to stare at code and think about all possible issues that might arise. Even in computer science, Third-Party Review of code focuses on whether code passes comprehensive suites of tests.

4 Responding to Errors

For all the reasons detailed here, I think it makes sense for the discipline to think more carefully about how we respond to errors discovered in the code underlying published papers.

The status quo in the discipline is, I think most people would agree, to assume that most code both is and *should be* error-free. When errors are discovered, therefore, the result is often severe professional sanction.

But is this appropriate? The people who work most with code (computer scientists) have long ago moved away from the expectation that code can be error-free. At the same time, however, we cannot simply say "errors are ok." The middle road, I believe, lies in recognizing that not all errors are the same, and that we must tailor our responses to the nature of the coding error. Errors caused by gross negligence are obviously unacceptable, but I feel we should be more understanding of authors who write careful code but nevertheless make mistakes.

To be more specific, I think that as a discipline we should try to coordinate on a set of coding practices we deem appropriate. Then if an error is uncovered in the work of someone who has followed these practices – adding tests, commenting their code, using good naming conventions, no duplicating information, etc. – then we should recognize that they took their programming seriously and that to err is human, even in programming. Errors should not be ignored, but in these settings I feel it is more appropriate to respond to them in the same way we respond to an error in logical argumentation, rather than as an indicator of sloppiness or carelessness.

To be clear, this is not to say we should persecute authors who do not follow these practices. Someday – when these practices are being consistently taught to students – I think it will be reasonable to respond to errors differentially depending on whether the author was employing error-minimizing precautions. But the onus is on us – the instructors and advisors of rising to scholars – to ensure students are appropriately armed with these tools if we wish to later hold them responsible for programming mistakes. To do otherwise is simply unfair.

But what we can do today is agree to be especially understanding of scholars who work hard to ensure the integrity of their code who nevertheless make mistakes, now and in the future. This, I feel, is not only normatively appropriate, but also creates a positive incentive for the adoption of good programming practices.

5 Other Resources

This document includes just a handful of practices that I think can be of use to social scientists. I'm sure there are many more I am unaware of, and I encourage readers who are aware of useful practices to send them my way.[3] Here are some starting references:

- Code and Data for Social Scientists handbook written by Matthew Gentzkow and Jesse M. Shapiro
- The Political Methodologist issue on Work-Flow Management

[3]Users who google "defensive programming" will find many resources, but be aware many may not seem immediately applicable. Most defensive programming resources are written for computer scientists who are interested in writing applications to be distributed to users. Thus much of what is written is about how coders should "never trust the user to do what you expect." There's a clear analogy to "never assume your data looks like what you expect," but nevertheless mapping the lessons in those documents to data analysis applications can be tricky.