

Professional statistical software development: What, why, and how?

Yulia Marchenko

Vice President, Statistics and Data Science
StataCorp LLC

2024 UK Stata Conference

Acknowledgment

I want to express my deep gratitude to Bill Gould, who mentored me on several projects and guided me through my 20-year career at StataCorp. I have learned so much from him and am proud to further share this knowledge with our Stata community.

- 1 Professional statistical software development
 - Who develops statistical software and why?
 - Efficiency and reliability
 - Reproducibility
 - Consistency
 - Extensibility
 - Documentation
 - Ease of use
 - Polish
 - Support
- 2 Final comments

Who develops statistical software and why?

- Researchers who want to integrate their new scientific methods into practice
- Professionals who wish to develop software for use by themselves or others within their organization
- Practitioners who want others to benefit from their work
- Consultants
- Commercial software providers such as Stata

Professional statistical software development (PSSD)

- Programming + Statistics = SSD.
- Programming + Statistics + *more* = PSSD.
- Everything in *more* is what sets PSSD apart from SSD.
- So what is this *more*?
- *more* encompasses all the skills (e.g., careful validation and testing of the software, clear and comprehensive documentation, ability to explain difficult concepts plainly, attention to detail) necessary to produce professional statistical software.
- “Professional” here means “high-quality”, not necessarily “commercial”.

What is professional statistical software?

- **Reliable:** produce correct statistical results; stable results across operating systems
- **Reproducible:** backward compatibility, integrated version control
- **Efficient:** strike a good balance between speed, memory, and disk usage
- **Consistent:** similar concepts accessed the same way to reduce learning curve
- **Extensible:** extensions of official features and addition of new features (community-contributed software)
- **Well documented:** complete, succinct, perhaps entertaining
- **Easy to use:** interface, localization, intuitive and consistent syntax
- **Polished:** consistent syntax, careful error traps, helpful error messages
- **Well supported:** knowledgeable and timely technical support; forums (e.g., Statalist); blog; videos; training; webinars; publications (e.g., *Stata Journal*, Stata Press); ...

Tips for adding a new statistical feature

- Acquire deep knowledge of the statistical area.
- Research the methods, and identify the necessary features to be added.
- Implement the underlying algorithms.
- Design easy-to-use syntax that is consistent with other existing features.
- Verify and certify the developed features: statistical correctness of the results, syntax, consistency between results across different platforms.
- Document the new features, potentially as an entire manual.
- Optionally provide a graphical interface for your features.
- Polish: provide clear and informative error messages, perform more extensive checks of specifications, and so on.

Producing efficient and reliable code

- There are often tradeoffs between speed and space.
- Be aware of your intended users and design your code accordingly.
- A researcher with limited computing resources might be willing to wait longer for the results.
- But another researcher with access to massive hardware would want faster results without regard to resource consumption.
- Sometimes you can offer both approaches—for example, Mata (Stata's matrix programming language) allows users to choose whether to favor speed or favor memory usage when executing code:

```
: mata set matafavor space  
: mata set matafavor speed
```

- Code should produce accurate and stable results.

Some good programming practices

- Avoid duplicate code by writing reusable functions and programs.
- Perform intermediate computations in the highest-possible precision (relative to the operation). For instance,

```
. tempvar x2  
. generate double 'x2' = x^2
```

- Vectorize the code whenever possible. For instance, in Mata, instead of looping over observations of a vector to obtain squared values,

```
for (i=1; i<=100; i++) x2[i] = x[i]^2
```

use the colon operator (`help mata op_colon`) for element-by-element matrix operations:

```
x2 = x:^2
```

Some good programming practices (cont.)

- Use more efficient operations whenever possible. For instance, using Mata's function `cross(X,Z)` may be more efficient than computing $X'Z$ manually. In Stata, if you need to compute only the mean of a variable, use

```
. summarize myvar, meanonly
```

- Comment your code:

```
// compute prediction error
```

```
/* compute prediction error */
```

- Use a consistent coding style and readable function names, argument names, etc.
- Handle edge and error conditions gracefully.
- Keep backward compatibility in mind. (See later.)

Some good programming practices (cont.)

- Keep reproducibility in mind (random-number seed, sort stability, etc.).
- Keep accuracy in mind (normalized scale, computation in log scale, etc.).
- Read more about programming in Gould (2018).

Verifying, testing, and certifying your software

- **External verification**—comparison with known benchmarks from published results or other software; independent derivations; verification by simulation of finite-sample properties and comparison with published simulation results.
- **Testing**—self-testing; external testing by someone else, including bad input, edge conditions, GUI interface (if any), etc.
- **Certification**—establishing accuracy of results against verified benchmarks; consistency from run to run; consistency across platforms (e.g., Mac vs. Windows vs. Linux), architectures (e.g., Intel vs. M1/M2), OS versions, computer configurations (e.g., different amounts of memory), and multiple cores.

Certification

- Certify against verified results—find examples with known answers, or use the results from simulations as benchmarks.
- Write a script that compares the results produced with benchmarks at the desired level of precision.
- Certify that the software gracefully handles edge and error conditions.
- Certification should stop upon error or produce a report of all encountered problems.
- You can easily automate certification in Stata by writing what we call a *certification script*.

Certification (cont.)

- Stata offers tools to aid in certification: `capture`, `assert`, `rcoef`, `dta_equal`, and more; type `help cscript` for more.

```
// begin myreg.do cert script
cscript

use mytestdata

// certify command produces error return code 198 with bad syntax
/* option badoption not allowed */
rcoef "noisily regress y x, badoption" == 198

regress y x

// certify R-squared value within 1e-8 tolerance of benchmark
assert reldif(e(r2), .6067688052) < 1e-8

// certify coefficients within 1e-8 tolerance of benchmark
matrix b_test = (<benchmark values>)
assert mreldif(e(b), b_test) < 1e-8

display "All is well with myreg.do!"
```

- Read more about certification in Gould (2001).

What is reproducibility?

- In science, we often think of reproducibility as the ability to obtain the same results repeatedly **under the same conditions**—*scientific reproducibility*.
- We might also think of it as a way to replicate published results—*scientific replication*: access to data, scripts, code, and scientific reproducibility.
- Or we might think of it as the ability to repeat (automate) certain tasks—automation: creating scripts, reports, tables, and scientific reproducibility.
- The steps to achieve reproducibility and the meaning of “the same conditions” depend on the type of reproducibility.
- We should also distinguish between *exact reproducibility*, where we expect results to be exactly the same, and *finite-precision reproducibility*, where we expect results to agree within an acceptable tolerance.

Aspects of reproducibility

Depending on the general type of reproducibility you are interested in, the following may be of interest:

- Stochastic reproducibility
- Numerical reproducibility
- Computer reproducibility
- Backward compatibility and integrated version control

Stochastic reproducibility

- The ability to produce the same results from a stochastic (random) procedure repeatedly.
- Stochastic procedures depend on random numbers.
- Each run of a random-number generator produces a different sequence of values.
- To obtain the same sequence, you need to specify a random-number seed; type `help set seed`. For instance,

```
. set seed 3876231
```

or use Stata's `rseed()` option with stochastic commands.

- Beware of computations performed in parallel. They should use streams (not all software does)—subsequences of random numbers drawn simultaneously that are guaranteed to be independent. Reproducibility requires the same seed for all streams; type `help set rngstream`.

Numerical reproducibility

- The ability to produce the same results from a deterministic (possibly iterative) procedure repeatedly.
- The same algorithm should be used.
- The same implementation of the algorithm should be used.
- Computations are performed using the same precision.
- Iterative procedures should use the same stopping and convergence criteria; e.g., see the `tolerance()`, `ltolerance()`, and `nrtolerance()` options in `help maximize`.
- Operations that depend on ordering or scale of values may need to be performed in the same order each time for numerical reproducibility; type `help sortseed`.
- Single versus multiple cores.
- Numerical differences across different computers.

Computer reproducibility

- The ability to produce the same results across different computers and operating systems.
- Closely related to numerical reproducibility
- Different computers have different chips, libraries, etc., that may lead to slight numerical differences across systems, especially for unstable computations.
- Single versus multiple cores.
- Exact computer reproducibility is rarely possible across different platforms because it typically requires exactly the same hardware and OS setup.
- As a result, exact numerical reproducibility is rarely attainable across different platforms.

Integrated version control (IVC)

- Backward compatibility and IVC—the ability to produce the same results in the future using the same software.
- IVC allows you to write your code such that it runs reproducibly 10, 20, etc., years later even in newer versions of the software.
- Do not confuse IVC with source version control, which helps you track changes in your code and project files over time.
- And IVC is not just a container that bundles the existing software and operating system to run on other infrastructures.
- IVC requires careful (ongoing) physical merging of all existing software code bases or versions over time.
- As far as I know, Stata is the only statistical software package, commercial or open source, that provides IVC.

Integrated version control (IVC) (cont.)

- As a user, you can access the old syntax, stored results, etc., by simply prefixing your old command with the `version` statement; type `help version`. For instance, in Stata 18, you type

```
. ci proportions y1 y2
```

to run the `ci` command to compute binomial CIs. But you can also type

```
. version 13: ci y1 y2, binomial
```

to run the old syntax of `ci`.

- As a programmer, you specify the version statement at the top of your program:

```
program myci
    version 13
    ci y1 y2, binomial
end
```

Integrated version control (IVC) (cont.)

- If you need to change the behavior of your program going forward, but preserve the old behavior to avoid broken old scripts, use the `_caller()` function with the current Stata version:

```
if (_caller() < 18) {  
    <old code>  
}  
else {  
    <new code>  
}
```

- In addition to traditional version, Stata also offers what we call a *user version*, which only changes when version `#` is typed interactively or used in a do-file. A user version is unaffected by the `version` statements in ado-files and programs.

Integrated version control (IVC) (cont.)

- User version was added later to handle important improvements that all existing programs should use regardless of the `version` set in an ado-file or other programs. But the need for user version control is typically rare.
- For instance, in Stata 14, we replaced the KISS random-number generator with the 64-bit Mersenne Twister. Now all existing ado-files and programs use this improved algorithm to generate random numbers despite the set `version`.

Integrated version control (IVC) (cont.)

- However, as a user, you may need to reproduce your old results before Stata 14 that used the KISS algorithm. If you set the `version` to less than 14 in your do-file or prefix the command with that version statement interactively, Stata will use the KISS algorithm to generate random numbers.
- As a programmer, if you want your program to respect a user version, you would replace `caller()` in the earlier code block with `c(userversion)`. Read more about user version in *User version* in **[P] version**.
- IVC is not a time machine—incorrect results or behavior is not version controlled!
- Read more about IVC at stata.com/version.

“Under the same conditions”

The key condition for scientific reproducibility is the ability to perform a procedure under the same conditions:

- same operating system and computer setup;
- same implementation of the procedure;
- same iteration criteria of the procedure;
- same random-number seed; and
- more.

Exact scientific reproducibility requires numerical and computer reproducibility and may not always be feasible!

Producing reproducible code

- For stochastic methods, make sure to provide a way to specify a random-number seed such as Stata's `rseed()` option, and, if streams are used, use it for all streams. Do not hardcode the seed in your program!
- Beware of sorting order when performing computations. Some computations may need to be made sort-order independent (`sort`, `stable`), but others may not.
- Use the same stopping and convergence criteria if, for instance, trying to match other software; e.g., type `help maximize` for Stata defaults.
- Avoid performing operations on values of very different scales: standardize or think carefully about operations. For instance, adding very small and very large numbers in different order might lead to numerical differences.
- Think of backward compatibility and version control.

Creating consistent software

- Establish rules for how to input and output information, and follow them as you add new features. For instance, if you are writing a command for linear regression and want the outcome and predictors to be specified following the command name,

```
. regress y x1 x2
```

do this for all regression estimators,

```
. logistic y x1 x2
```

instead of, say, specifying them in options with a new estimator:

```
. logistic, outcome(y) predictors(x1 x2)
```

Creating consistent software (cont.)

- Name your options, programs, etc., consistently:

```
. regress y x1 x2, level(90)
```

```
. logistic y x1 x2, level(90)
```

versus

```
. regress y x1 x2, level(90)
```

```
. logistic y x1 x2, cilevel(90)
```

- Label the same concepts consistently in your output, and maintain a consistent output style.

Producing extensible code

- Write clear code and use consistent style. You can often follow the style of the official code—all the `.ado` and many `.mata` files are viewable with the `viewsource` command.
- Include comments in your code.
- Follow the established software syntax when designing a new command. You can use Stata's `syntax` command and other parsing utilities (e.g., type `help _parse`).
- Write generalizable code, e.g., do not hardcode names of user inputs such as assuming an outcome is always named “y”.
- Write flexible code: allow the user to provide overrides for default parameters such as initial values.

Producing extensible code (cont.)

- Make sure any results displayed by the command and possibly some useful intermediate results are accessible programatically; type `help stored results`.
- Implement useful procedures as standalone utilities that can be easily reused by others. (Type `help undocumented` for some utilities we use in-house.)
- Structure your code in a way that lets others benefit from some built-in computations (e.g., Stata's `power` command supports user-defined methods with automatic support of multiple values, graphs of power curves, etc.).

Providing good documentation

- Any software documentation must
 - provide a description of what your procedure does;
 - describe all options and features;
 - describe the underlying methods and formulas; and
 - provide references.
- Additionally, including examples of various command usages will greatly increase the chance of your procedure being successfully used by others.

Providing good documentation (cont.)

- Good software documentation
 - describes all of the above clearly and concisely;
 - provides a brief motivation and introduction for the implemented methodology;
 - provides detailed examples (ideally in a variety of fields and applications) that are easy to follow and to adopt to user-specific applications;
 - describes the output and interprets the results;
 - is comprehensive yet concise;
 - is possibly entertaining.

How to write good documentation

- Consider your audience.
- Avoid jargon unless your software is intended only for experts in the area.
- Don't assume that all users will be familiar with the terminology. Define the main terms relevant to the implemented methodology.
- Try to provide intuitive explanations of concepts before resorting to formulas. Leave the latter to the more technical section about methods and formulas!
- Research the literature for applications in a variety of disciplines, and provide detailed examples of various usages.

How to write good documentation (cont.)

- Try to strike a good balance between comprehensiveness and conciseness.
- Be careful to properly cite the sources and acknowledge previous work.
- Include the original references for the data and not only the methodological papers that use them.
- Have someone unfamiliar with the methodology read your documentation and try your command when you are done.
- See Stata documentation at [stata.com/documentation](https://www.stata.com/documentation).

Making your software user-friendly

- Provide consistent and convenient specifications.
- Use logical and intuitive names for commands, options, etc.
- Provide good documentation.
- Polish your software! (More about this later.)
- If possible, provide a graphical interface that teaches users how to use your command. (Type help dialog programming.)

Polishing your software

- Check for consistency of syntax, option names, and other specifications.
- Check that your output is grammatically correct, clear, and consistent.
- Check that you blocked all the nonsensical uses of the command. For instance,

```
. rcof "noisily mycmd, dots nodots" == 198  
only one of options dots or nodots is allowed
```

Programmers' utility `opts_exclusive` is helpful for checking mutually exclusive options.

Polishing your software (cont.)

- Check for “silent” specifications/options—what I call the junk test. (Think of mistyped options, etc.)

```
. rcof "noisily mycmd, junk" == 198  
option junk not allowed
```

- Provide helpful error messages to explain incorrect or unsupported usages.

```
mi impute logit: perfect predictor(s) detected
```

Variables that perfectly predict an outcome were detected when **logit** executed on the observed data. First, specify **mi impute**'s option **noisily** to identify the problem covariates. Then either remove perfect predictors from the model or specify **mi impute logit**'s option **augment** to perform augmented regression; see [The issue of perfect prediction during imputation of categorical data in \[MI\] mi impute](#) for details.

```
r(498);
```

Polishing your software (cont.)

- Give warnings when the results might not be trustworthy:

Warning: Convergence not achieved.

- Display notes that help explain behavior of commands:

```
. bayes: regress y x  
( output omitted)
```

Note: [Default priors](#) are used for model parameters.

You can even link from your notes (warnings, error messages, etc.) to help files with more explanation. In this note, [Default priors](#) links to help `j_bayes_defaultpriors`.

Providing software support

- Be ready to answer questions about your software promptly.
- Be ready to fix problems with the software if they occur.
- You might want to write blog entries, create videos, and offer trainings and webinars to help users learn about your software.
- And you might want to submit your software for publication in the *Stata Journal* (stata-journal.com) or even write a book about it with Stata Press (stata-press.com).

Final comments

I think producing professional statistical software is

- impactful,
- rewarding, and
- challenging.

It brings together practitioners and researchers from different disciplines. And it provides exposure to a wide variety of statistical areas.

Reference

Gould, W. W. 2001. Statistical software certification. *Stata Journal* 1(1): 29–50.

Gould, W. W. 2018. *The Mata Book: A Book for Serious Programmers and Those Who Want to Be*. College Station, TX: Stata Press.

THANK YOU!