What Can Statisticians Learn From Software Engineers?

Paul Teetor
Quant Development LLC

Chicago Chapter
American Statistical Association
Oct 2014

Does this sound like your world?

Frequent copy-paste-edit.
Keep reinventing the wheel.
Testing code is difficult at best.
You don't trust your own code.
Too much code and too many files.
Keeping multiple versions.
Can't identify the latest version.
New-comers are baffled by the code.
Major changes are somewhere between difficult and unthinkable.

I see SAS and R users swimming in code.

Maybe drowning in it.

Software engineers have solved many of these problems.

Decades ago.

There is a better way.

- Better coding practices
- Better program structure
- Testing, testing, testing
- Codebase management

There is a payoff for the investment of time and energy.

- Correctness
- Maintainability
- Less frustration, more happiness

Programming in the Small

vs.

Programming in the Large

Coding:
Cornerstone concepts and techniques

- Clean code
- Defensive programming
- Code walk-throughs
- Refactoring
- Modularity

Clean code: Good indentation reveals structure

Quick! What is the structure of this code?

```
gcd = function(a,b) {
if (b == 0) a
else gcd(b, a %% b) }
f0 = function(x,y) {
m = lm(y ~ x + 0)
confint(m) }
```

Clean code: Good indentation reveals structure

Ah! Now we see the structure.

```
gcd = function(a,b) {
  if (b == 0)
    a
  else
    gcd(b, a %% b)
}

f0 = function(x,y) {
  m = lm(y ~ x + 0)
  confint(m)
}
```

Clean code:
Use names that enhance readability.

- Suggestive or descriptive names
- Stay close to natural language
- Not too short, not too long
- Typical rule of thumb:
  - Use nouns for variable names: *scores*
  - Use verbs for names of functions: *loadScore*
  - Property names for predicates: *isScore*

Clean code:
Use names that enhance readability.

- Too short:

```
lm(p ~ m + s)
```

- Too long:

```
lm(profitByMktAndSeason ~ MarketSector + AnnualSeason)
```

- Just right:

```
lm(profit ~ mkt + season)
```

Clean code:
Minimize or eliminate global variables.

- Wrong way:

```
fitXY = function() glm(y ~ x, family=binomial)

y = loadY();  x = loadX();  fitXY()
```

- The *fitXY* function is too inflexible
- Cannot easily use model in different context (e.g., on *y* and *z*)

Clean code: Functions should not use or change global variables.

- Better:

```
fitXY = function(x, y) glm(y ~ x, family=binomial)
```

- Remove dependence on global *x* and *y*
- Can easily reuse this function later

Defensive programming catches bugs.

- All code makes assumptions
  - Parameter types
  - Available data elements
  - Expected results
  - Etc.
- Check the assumptions
- Halt when assumptions violated (or at least warn)

Defensive programming: Checking types catches many bugs.

- The *inherits* function checks type

  `inherits(x, “numeric”)` - *TRUE if x is a number*

- Typical usage: Halt if parameter not data frame

```
fun = function(dfrm) {
   stopifnot(inherits(dfrm, “data.frame”))
   . . .
```

Defensive programming: Checking for required input and size

- Does a data frame contain a required column?

```
if (!(“date” %in% colnames(dfrm))
    stop(“Input is missing 'date' column”)
```

- Does a data frame actually contain data?

```
if (nrow(dfrm) == 0)
    stop(“Input is empty”)
```

Defensive programming: Checking expected results

- Example: Did an optimization converge?

```
opt = optim(par, fn)
if (opt$convergence != 0)
   stop(“Optimizer did not converge”)
```

Comments in code:
Blessing or blight?

- Old School: Comments required
- New School: Comments are a waste of time and energy
- My standard:

Does the comment enhance the reader's understanding of the code?

Does it improve the readability?

Comments in code: Some situations demand commenting.

- Odd or unexpected steps must be explained
- Mathematical notation in code must be “decoded”
- Example: What the heck does this do?

```
C = pnorm(d1)*S – pnorm(d2)*K*exp(-tau*(T-t))
```

- Explain in the comments:

```
# C = B/S call price; S = stock price
# K = strike price; T = expiration time
```

Comments in code:
Some recommendations

- Focus on *why, not how.*
- Before every function, explain its purpose (the *why*).
- Inside functions, minimal comments on *how*.
- Document assumptions regarding parameters *and* check those assumptions.
- Always comment on the special situations (as shown earlier).

Code walk-throughs are an opportunity for feedback and quality control

- H_0: Your code is correct, clean, and readable.
- H_1: You're kidding yourself. It's not.
- Perform an experiment: Let others review it.
- Line-by-line reading. Detailed feedback.
- Leave your ego at the door.
- Code standards very important.

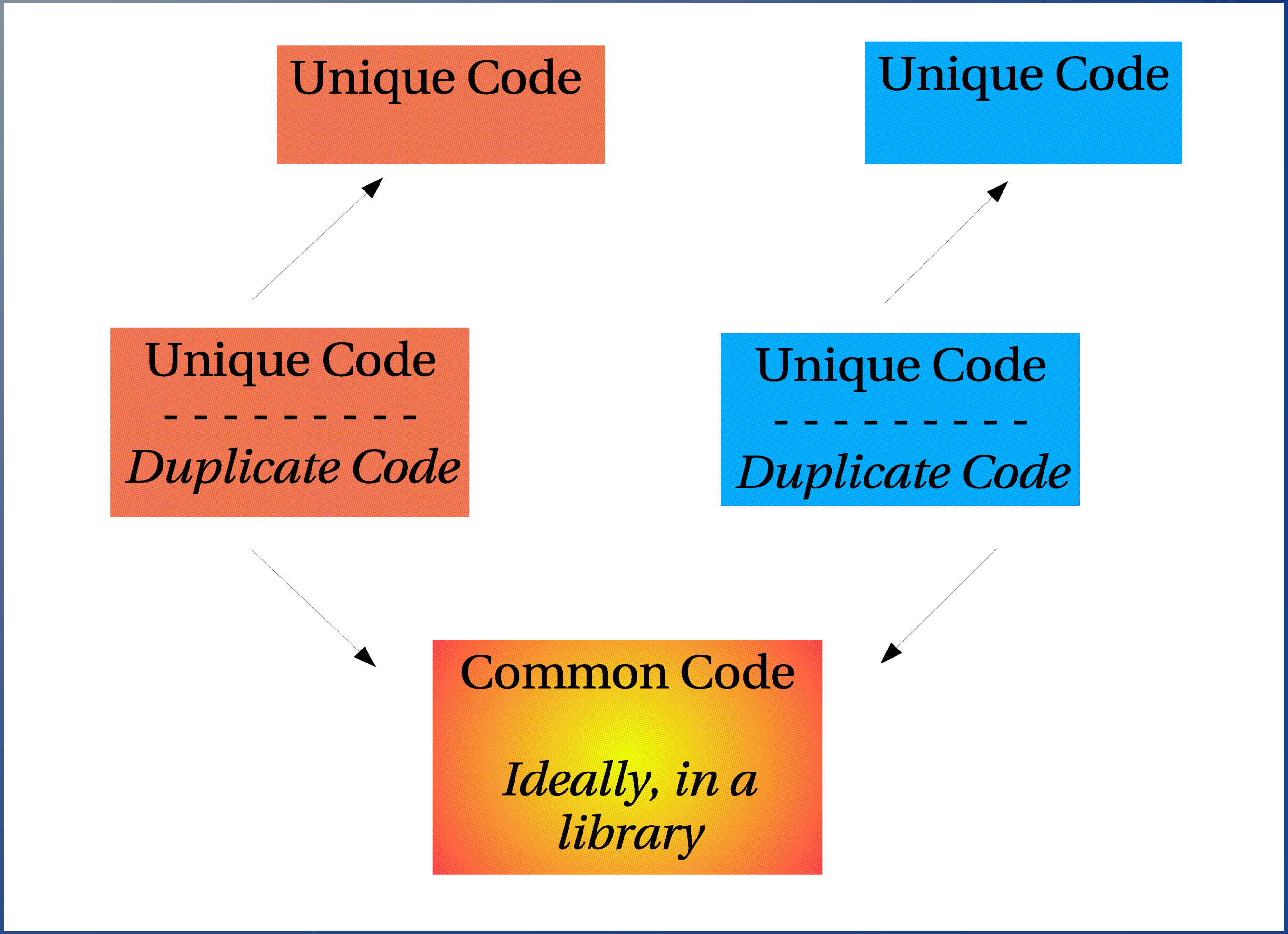
Code walk-throughs:
Let's read this code. What's your feedback?

```
coalesce = function(...) {
for (x in list(...)) {
if (!is.null(x) && !is.na(x)) return(z)
}
NULL
}
```

Code walk-throughs:
Better for the feedback.

```
#
# Return first non-NA and non-NULL parameter;
# otherwise, return NULL.
#
coalesce = function(...) {
  for (x in list(...)) {
    if (!is.null(x) && !is.na(x))
      return(x)
  }
  NULL
}
```

Refactoring means abstracting and reorganizing code to improve the design.



Refactoring means abstracting and reorganizing code to improve the design.

- Warning sign: frequent copy-paste-edit
- Symptom: Much code looks alike
- Symptom: New-comers are baffled by design
- Abstraction to reduce redundancy
- Eliminate contorted structure
- Often after many/significant changes
- *Why write 10,000 lines when 1,000 will do?*

Good *modularity* means the design is clean and the code reflects that design.

- Goal: Design is easy to understand, easy to change
- Modules mirror design
- Every module's purpose is clear.
- Every module's purpose is distinct.
- More art than science

.

Modularity:
Break the problem into clear, clean, manageable pieces.

Beginners tend to write large, monolithic scripts.

Experienced programmers write many, little functions

Modularity:
Will this program be easy to change?

```
# Load data, cleanse it, fit model
m = buildModel(date)

# Evaluate model and report results
evalModel(m)
```

Modularity: Will this design be robust in the face of change? *Probably not.*

```
# Load data, cleanse it, fit model
m = buildModel(date)
# Evaluate model and report results
evalModel(m, filename)
```

- *Q: What happens if model changes?*
- *Q: What happens if models are added?*
- *Q: What happens if we change the data source?*

Modularity: This design has clearer separation of purpose.

```
in = loadInput(date)
clean = cleanseInput(in)
m = fitModel(clean)
ev = evalModel(m)
reportEvaluation(ev)
```

- *Q: Can we change the model? Add add'l models? Change the data source?*

Software engineers use many tools and processes to streamline their work.

- Test tools
- Libraries of reusable components
- Separation of development and production
- Version control software (VCS)
- *The R community uses these tools extensively, for example.*

Unit testing

- Identify low-level bugs quickly
- Reliable foundation
- Test all significant functions
- Known inputs lead to expected results
- Invalid inputs trapped

Unit testing: Example
For R, I recommend the *test_that* package.

From my library, testing the `coalesce` *function:*

```
test_that("coalesce works as expected", {
    expect_that(coalesce(), equals(NULL))
    expect_that(coalesce(NULL), equals(NULL))
    expect_that(coalesce(NULL, NULL), equals(NULL))
    expect_that(coalesce(NULL, "foo"), equals("foo"))
    expect_that(coalesce("foo", NULL), equals("foo"))
    expect_that(coalesce("foo", "fum"), equals("foo"))
})
```

This is one of hundreds of tests
designed to insure the integrity of my library.

Unit testing:
There are some challenges

- Investment of time and energy
- *Coverage* is critical
- Maintainance

Libraries

- Capture the common parts
- Share the wealth
- Don't reinvent the wheel
- Don't debug twice

Capture, share, and reuse
your intellectual property.

Libraries:
It's easy to create a simple, local library in R.

- Identify and isolate common parts
- Use *devtools* package and RStudio
- Use `library(OurLib)`
- Excellent opportunity for documentation
- Excellent opportunity for unit tests

Execution environments: Don't mix stable code and new code.

- Development environment
- Production environment
- Optional: Testing environment
- Wall them off
- Each environment is self-contained: apps, lib, data

Execution environments: As simple or complex as you need

- Structure can be a simple set of parallel folders:

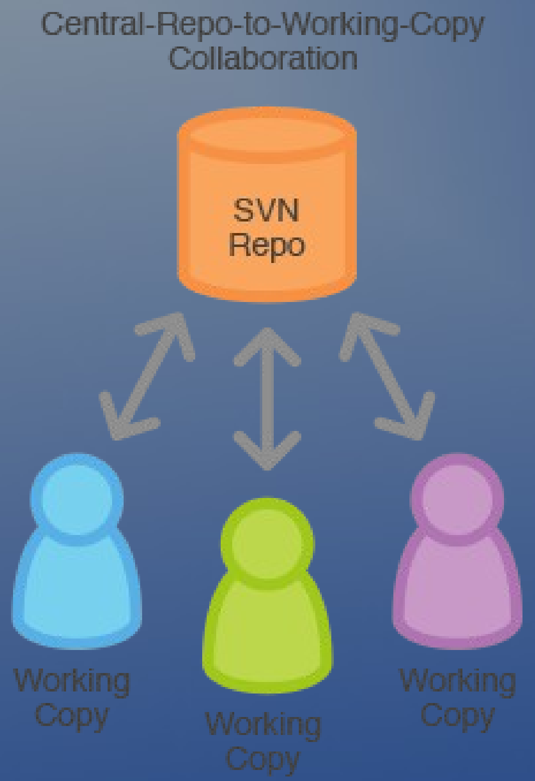
```
dev
    apps
    lib
    data
prod
    apps
    lib
    data
```

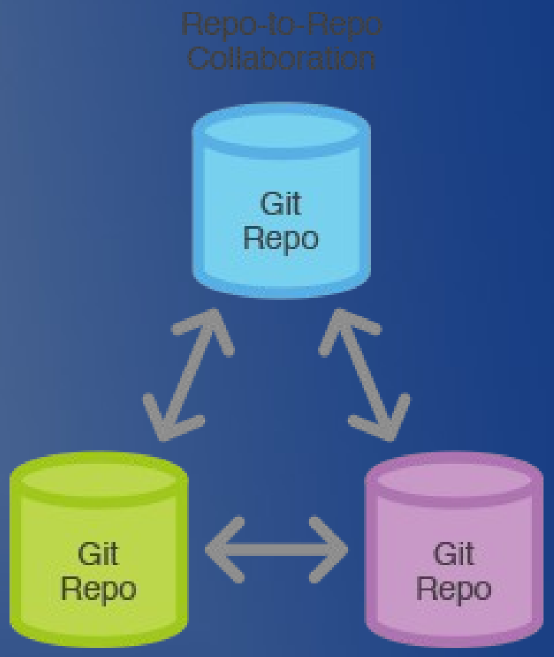
- Or very complicated; e.g., multiple machines

If you recognize these symptoms, you may need *version control*.

- Swimming in code and files
- Numerous versions
- Saving everything “just in case”
- “Where's the latest version?”

Version control provides a safe repository for your files and their history.





Version control:
The repository is your “system of record”

- Exactly defines codebase
- Complete history
- Protection against loss or deletion
- Supports parallel development activity
- Supports versioning

Version control:
Lets you commit and share files.

The *git* package, for example, provides:

- add *filename* – Add file to repository
- commit *filename* – Commit changes to repository
- diff *filename* – Compare against repository version
- log *filename* – Show history of changes
- status – Show status of changes and repository
- push – Share your changes with others
- pull – Receive changes from others

Finally, a dark secret of software engineering: *The 20% Rule*

If you need to change 20% of your application or more, just start over.

- Big changes will likely invalidate design
- Make code reflect new design, not old history
- Recycle as much as possible.
- *Don't tell your boss*.

Remember...

- Treat programming like a social activity
- Keep your code readable and true to the structure
- Use good coding technique
  - Code cleanly, be defensive, do walk-throughs, refactor, modularize
- Use the right tools
  - Unit tests, libraries, environments, version control

Thank you!

Slides available at

`bit.ly/ccasa-2014-oct`