Best Practices in Code Development

Tamas Gal (ECAP)

July 24, 2020

Disclaimer

This talk sheds light on a few things to think about when developing code. It's OK if you say "Why certainly!" every other slide. These are things which need to be spoken out to have an actual impact.

Everyone of us would came up with the same ideas after thinking about them proactively.

Any similarities to persons living or dead, or actual events are purely intentional.

Introduction

- Tamas Gal (`tamasgal` on GitHub/GitLab/Twitter)
- Physicist at the Erlangen Centre for Astroparticle Physics (ECAP)
- Working on the KM3NeT Neutrino Telescope experiment
- One of the maintainers of the IT services and infrastructure of KM3NeT and ECAP

Roadmap

- Source code
- Testing
- The API
- Versioning
- Documentation
- Automation
- Contributing
- Tooling…

Level 1: Source Code

Serves multiple purposes:

- the actual implementation
- foundation for the documentation

- the home of amazing, hilarious and exotic bugs
- probably the best place to make others cry
- the thing you stare at almost all the time

Basic principle

"Indeed, the ratio of time spent **reading versus writing** is well over **10 to 1**. We are **constantly reading old code** as part of the effort **to write new code**. . . . [Therefore,] making it easy to read makes it easier to write."

– Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

Naming

`d`

`elapsed_time_in_days`

`el<TAB>`

Tab-Completion is a thing. . .

Comment-Code-Redundancy

Don't comment obvious things. Try to express yourself through code.

Annoying

```
def read_humidities(sensors):
    "Auxiliary function to read the humidities from multiple sensors"
    data = []  # list to store the humidities
    n = len(sensors)  # number of sensors
    for i in range(n):
        value = sensor.read()
        data.append(value)
    return data  # return a list of humidities
```

Concise

```
def read_humidities(sensors):
    "Read the humidity from multiple sensors"
    return [sensor.read() for sensor in sensors]
```

A helpful comment

```
# Matches the UTC format YYYY-MM-DDThh:mm:ssZ
match(r"^[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}:[0-9]{2}Z$", datetime)
```

Let the computer do the loops. . .

```
34432        case 342:
34433            calib[341] = "/pmt_342.dat"
34434            break;
34435        case 343:
        calib[342] = "/pmt_343.dat"
        break;
    case 344:
        calib[343] = "/pnt_344.dat"
        break;
    case 345:
        calib[344] = "/pmt_345.dat"
        break;
```

Keep the number of function arguments low

Questionable

```
def calibrate(fname, d, phi, gamma, pos_x, pos_y, pos_z, n, max_n, start_at, wait_until, n_iteratio
```

Better

```
def calibrate(fname, params: CalibParams, opts: CalibOptions)
```

Using type hinting in Python and extra classes `CalibParams` and `CalibOptions` which take care of further documentation, error checking and default values.

Boy Scout Rule

Leave the code better than you found it

But: do not mix unrelated things into a single commit or merge/pull request

Try to increase the code test coverage with each commit (see Level 2)

Code format

- Stick to a (preferable widely accepted) style guide
- Use a tool to do it for you (for Python: Black, yapf, ...)
- Adapt when contributing to projects (even if they do not share your aesthetics)
- Otherwise convince him that code style xzy is superiour and make a pull/merge request

Makefile

- Use a `Makefile` to create one-word commands to run a set of tasks
- `make test` vs `py.test --junitxml=reports/junit.xml -o junit_suite_name=main the_module`
- Adds an additional abstraction layer for the CI configuration (see Level 6)
- Take care of setting up the development workspace (`make install-dev`)
  - Create the virtual environment
  - Install development dependencies

Example `Makefile`

```
install:
pip install .

install-dev:
pip install -r requirements.txt
pip install -r requirements-dev.txt
pip install -e .

test:
py.test --junitxml=./reports/junit.xml -o junit_suite_name=$(PKGNAME) tests

test-cov:
py.test --cov ./km3io --cov-report term-missing --cov-report xml:reports/coverage.xml --cov-report

test-loop:
py.test tests
ptw --ext=.py,.pyx --ignore=doc tests
```

Level 2: Testing

Testing is crucial

- It makes sure that a given behaviour is reproducible
- Adds an additional layer to the documentation: the user can learn from them
- Most importantly: it suppresses the fear to change existing code

Reproducible/expected behaviour

- Tests are routines which use your code with a give input and make sure that the output meets the expectations
- Everything can be tested, but sometimes it's not straight forward (keywords: mocks, stubs, fakes, . . .)
- To test e.g. a DB access, you don't need the DB itself, you can "mimic" its response based on the expectations

Example (Julia codebase)

```
@testset "binomial" begin
    @test binomial(5,-1) == 0
    @test binomial(5,10) == 0
    @test binomial(5,3) == 10
    @test binomial(2,1) == 2
    @test binomial(1,2) == 0
    @test binomial(-2,1) == -2 # let's agree
    @test binomial(2,-1) == 0

    #Issue 6154
    @test binomial(Int32(34), Int32(15)) == binomial(BigInt(34), BigInt(15)) == 1855967520
    @test binomial(Int64(67), Int64(29)) == binomial(BigInt(67), BigInt(29)) == 788659796224916616
    @test binomial(Int128(131), Int128(62)) == binomial(BigInt(131), BigInt(62)) == 15731172098055
    @test_throws OverflowError binomial(Int64(67), Int64(30))
end
```

The Fear of Change

- A probably well known feeling: the house of cards
- The fear to change a single piece because everything could collapse
- Tests are there to make sure that things still do the same things when changes were introduced
- It makes it easy to refactor the code and do e.g. performance improvements

Testing habits

- Try test-driven development (TDD): write the tests first, then the code
  - Improves the overall design (it's testable by definition)
  - Makes the API more user-oriented
  - Drives a good code coverage
- Create tests for each bug or feature request
- Keep track of the code coverage and aim for 90%+

Level 3: The API

"An application programming interface (API) is a computing interface which defines interactions between multiple software intermediaries."
– Fisher, Sharon (1989). "OS/2 EE to Get 3270 Interface Early

Consider two layers of the API

public API and private API

Try to keep the public API stable, think about what to expose and what not

Once you push your code to a public repository, there is a chance that code is created which immediately depends on your public API

Every time you change the public API, you will potentially break existing code.

Strategies to keep the public API stable

- Tests!
- Don't expose private functions (if possible. . .)
- Express "privateness" (private functions or functions with a leading underscore)
- Hide implementation details
- Communicate with the users and discuss before making changes!

Deprecate vs. “It’s gone, live with it.”

- An additional deprecation stage until the next breaking release will notify people in advance
- Keep the full of the “old” API present and let it use the new API while showing a warning with the exact usage

```
julia> findn([1,0,3])
 Warning: ‘findn(x::AbstractVector)‘ is deprecated, use ‘(findall(!iszero, x),)‘ instead.
   caller = top-level scope at none:0
 @ Core none:0
([1, 3],)
```

I/O

Defining input/output formats is a big deal, similar to the public API

- Each change of these formats will eventually spawn an if/else/switch block somewhere in the universe
- At least a basic data provenance model is generally a good idea to start with
- Keep the I/O format definition in-sync with the public API
- Again: communicate with the users and discuss upcoming changes and their impact

Level 4: Versioning

I recommend sticking to Semantic Versioning: SemVer.org

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR: incompatible API changes
- MINOR: new functionality (backwards compatible)
- PATCH: backwards compatible bug fixes.

This clear scheme will easily tell the user when it’s safe to update or how to restrict dependency versions

Level 5: Documentation

Four layers to rule them all

- Getting Started
- Tutorials
- Concepts
- API

The user will typically fall from the sky and crash through the layers from top to bottom.

Getting Started

- Most of the people will only reach this point, so get this right…
- What is this software about? What problems can it solve? (what other will it create?)
- How to install the software and what dependencies are required
- Show a simple, prominent (and working!) use case (make it copy&paste-able)

Tutorials

- Preferably short guides which demonstrate how to do a specific thing
- Stay close to actual use cases and avoid hypothetical scenarios

Concepts

- The core concepts and design decisions
- Explain the big picture
- Isolate the fundamental building blocks and explain how they connect

API

- The technical documentation of your code
- Lowest level description including implementation details
- Preferably automatically generated (Sphinx, Doxygen, Documenter.jl, …)

Level 6: Automating Things (aka Continuous Integration)

Having a test suite is one thing, running it all the time in different environments is another crucial one

Continuous Integration (CI)

- Let the machine do things we have to do repeatedly
- That's the reason we developed machines in first place…
- We have powerful tools to automate many things
- Docker/Singularity containerisation helps to create isolated environments

How to CI?

GitHub Actions, GitLab CI, Travis, Jenkins, AppVeyor, …

Bug your local system administrator if you don't have such a system running at your institute. A typical Linux nerd can roll off a very basic CI platform within a few days.

GitLab CI

Example (`.gitlab-ci.yml`)

```
build:
  script:
    pip install .
```

That's all you need to check if your Python package can be installed

Things to automate:

- Compilation (if applicable)
- Running the test suite (is everything working?)
- Installation
- Benchmarks (are things slower/faster than before?)
- Documentation (should always be up-to-date)
- Living tutorials
- Publishing of the package (e.g. PyPI)
- Creation of Docker/Singularity images

Running all these on each push (or tag, whatever) is a game changer

- Immediate feedback to contributors in merge/pull requests
  - Status of the CI
  - Test coverage changes
  - Code style checks
- Cover all target environments (different Python versions, Linux distributions, Windows. . .)
- Reproducible, clean environment

Beware of vendor lock-in

- Each CI has its own configuration format and procedure
- A `Makefile` can be used to outsorce tasks
- Switching to another CI system can still be tedious

Level 7: Contributing

Don't Blame Others (even if they make you cry)

Improve Things Instead

Spread Your Ideas

It's awesome when someone opens a pull request to one of your repositories

Why not do the same? (to others)

Add contribution guidelines (`CONTRIBUTING.md`)

Provide issue templates with clear steps to follow

Add `CITATION` or alike and consider writing a JOSS paper

Final Boss: Tooling

For some people tooling is the #1 procrastination: toolcrastination

Typical signs of toolcrastination:

- Opening a thread in a forum to ask for the "best editor/IDE" for a given language
- Reading such threads more than once a day
- A strong belief that a better tool will immediately fix a bug or make you a better developer
- Taking part in flame wars, like Vim vs Emacs

Remember the read/write ratio of 10:1

- Syntax highlighting and recursive search are the most useful functions
- Almost every editor can do this out of the box
- Every tool has to be mastered

However, sophisticated setups can help with complex tasks immensely

- Refactoring large codebases
- Automatic test suite status integration
- Remote pair-programming

But still, most of the time we stare at the screens and think...

P.S. use Emacs with EVIL mode ;-)

Thank you for your valuable time.