Refactoring in Python
Design Patterns & Approaches

Tin Marković (Kiwi.com)

Introduction

- What is refactoring?
- What’s the point?
- How to do it well?
- Why not throw away everything?

Speaker

- Tin
  - Team Leader at Kiwi.com
  - Software Architecture as passion
  - Experiences working with edX and other big projects
- What can I share? What have I seen?

Company: Kiwi.com

- ***Making travel better***
- Our vision is to make travelling simple and accessible to everyone
- Focused on tech
- Stack: Python with friends :)
- Virtual Interlining

Abstract

- Read from old code, see the secrets it hides
- **Chesterton’s Fence**
- Incremental changes
- Testing code — dependencies, systems and relationships: concerns

Overview

- General topic, specific examples
- Easy Wins
- Patterns and Antipatterns
- Philosophy

Easy Wins: Intro

- Easy wins are easy
  - Plugins
  - Libraries
  - Utilities
- Dances around the root cause

Automated code quality

- Tools are cool
- One decision, vast time saved
- Examples:
  - Pylint
  - MyPy
  - Black
  - Coala

Tools: PyLint and MyPy

- Pylint "lints" code according to rules
- Established industry practice
- Bare minimum, often not automated
- MyPy checks if annotations follow typing
- Opt-in on a per-function basis
- Easy to implement slowly

Tools: Black

- Black keeps code style consistent
- Super simple to run and keep running
- No arguments about unimportant things
- Keeps the same interpreter output

Example before/after tooling

```
def complicated_foo(arg1, arg2, arg3, arg4
                                        arg5, arg6, arg7, **kwargs):
        z = []
        # z.append(arg1)
        # z.append(arg2)
        z += [arg2, arg3, arg4, arg5, arg6, arg7]
        print(z)
        if z:
                return z

# AFTERWARDS:

def complicated_foo(
        arg1, arg2, arg3, arg4, arg5, arg6, arg7
):
        # type: (int, int, int, int, int, int, int) -> List[int]
        """Extremely trivial example of cleaning code formatting."""
        z = []
        z += [arg2, arg3, arg4, arg5, arg6, arg7]
        log.info(z)
        return z
```

Easy Wins: Conclusions

- Tools make a lot of discussion not necessary
- This is a great win:
  - More thinking about problems
  - Less thinking about linebreaks
- Easy bump in code quality
- Just a bump, doesn't solve core issues

Patterns and Antipatterns:
Introduction

- Code hard to use
- Suprising facts
- Principle of Least Astonishment
- Legacy is often astonishing
- "*Historical Reasons*"

Code Smells

- Smells of:
  - Neglect
  - Inconsistency
  - Redundancy
- Because of:
  - Deadlines
  - Cost-cutting
  - Prototyping
  - Top Prio Requests

Levels of Code Smell

- Easy smells:
  - Couple of lines of code, scope nonexistent
- Medium smells:
  - Architecture mistakes
  - Larger scope and respawning
- Hard smells:
  - Easy to notice, impossible to remove
  - "Lets rewrite everything!"

Examples of Code Smell

- Easy

```
if 'AlphaAirline' in affily:
    currency = 'RUB'
elif 'MyBeta' in affily:
    currency = 'RUB'
elif 'GammaWings' in affily:
    currency = 'RUB'
```

- Medium

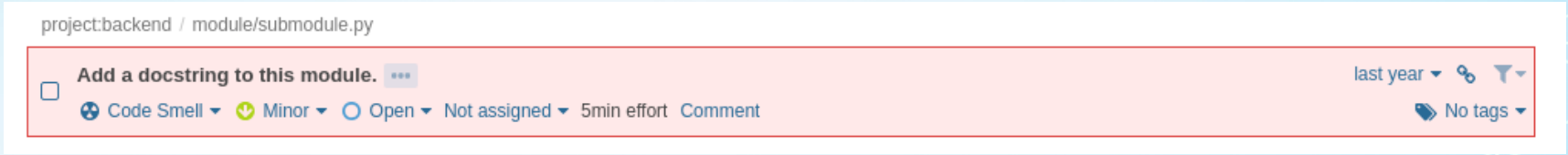
```
class AdditionalOrder:
    def order(self, data):
        # type: (dict) -> dict
        """Verify additional order data &
        payment, then store it."""
```

- Hard: Implement ORM

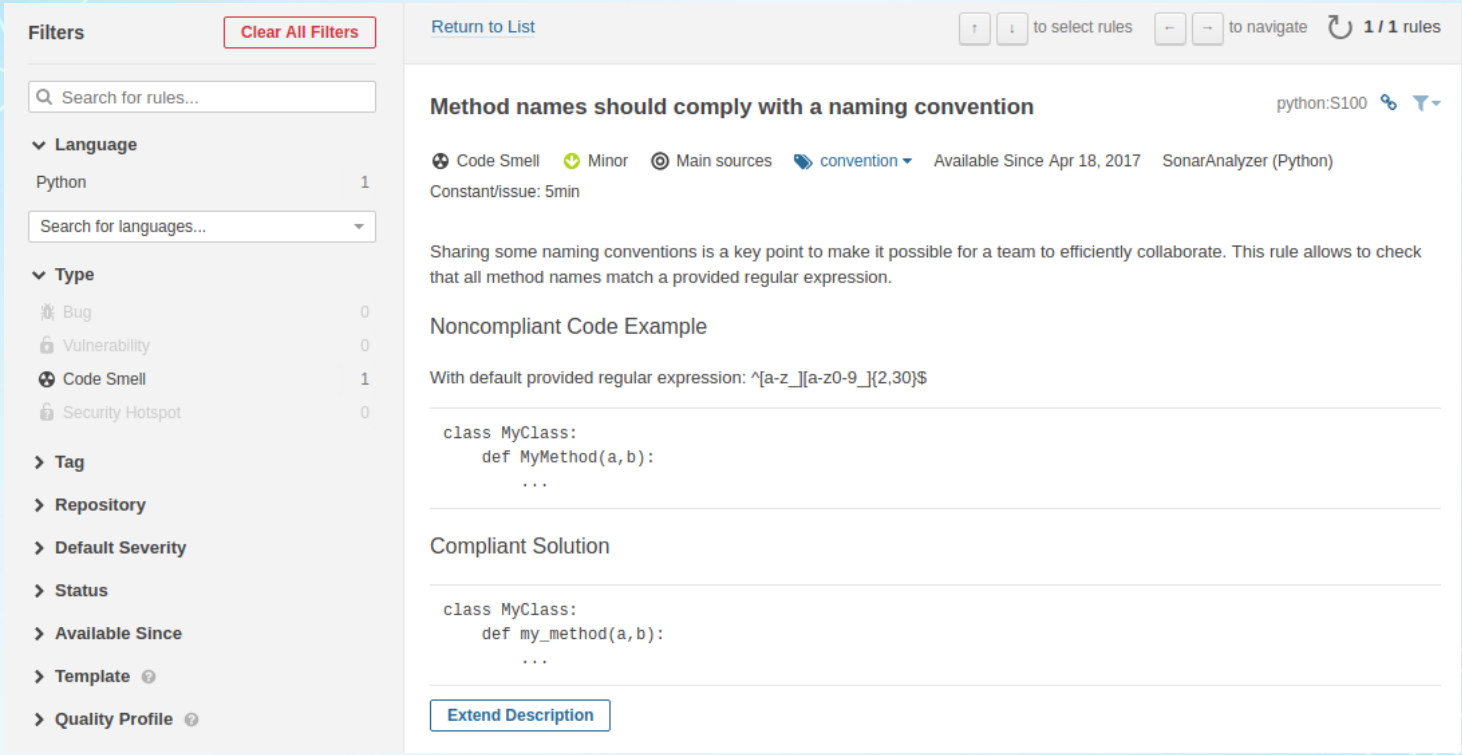
Tools: SonarQube

- Static analysis of code
- Analyses:
  - bugs
  - code smells
  - known security oversights
  - test coverage and complexity
  - comments and docs

Example: SonarQube output



Example: SonarQube output



Antipatterns to recognize

- Antipatterns mostly unique to codebase
- Lack of strong architectural direction
- Organic code growth
- Copy paste coding

Magical methods

- Lacking explicit input and output
- Usually an implemented side effect
- Replaced by better object oriented approach

```
def clean_foo(bar)
        # clean up internals from the response
        bar["gaz"] = bar["ordered_gaz"]
        bar["gaz"]["category"] = bar["gaz"]["category"].value
        del bar["db_record"]
        del bar["ordered_gaz"], bar["selected_gaz"]
        return bar
```

Overly important decorators

- Should not modify function signature
- Should be explicit
- Should not replace method calls

```
@user_auth()
@log_all_for_bottle()
@format_response()
@handle_params(pass_environs=True)
@errorlib.ErrorTracker(level="fatal")
def user_detail(data, environs, user_instance):
    """Get info about user account."""
```

Bubbles in Code

- Code can be a big interconnected mess
- Messy code: hard to mantain, harder to test
- Start implementing separation — isolate
- Isolated segments can be tested
- These "bubbles" can be tested and reliable
- Consistently expand and form new bubbles

Patterns to implement

- Old code needs separation
- New code needs to flourish (bubbles)
- Separation patterns:
  - Interface
  - Facade (and inverted)

Interface

- Find common usages of code pattern
- Try to find base use-case
- Create interface
- Add edge-cases through implementations

Facade (and Inverted!)

- Wrap your code in a facade fitting old code
- Keep required side-effects there, but obvious
- Manage required functionality in one place
- Implementation may be hacky, but you start:
    - implementing a contract
    - standardizing access
    - showing the ideal state

Testing an old codebase

- Very useful, worth all effort — confidence!
- Unit tests, Integration tests, Smoke tests
- Test without making it a charade
- Functional approaches — *side effects*
- Refactor code to accommodate tests
- Tests: a guideline to quality

Patterns and Antipatterns: Conclusions

- Code is almost never pretty after growth
- We can't throw everything away
- We can improve gradually
- Bubble of clean code

Philosophy: Introduction

- Theory is good, implementation better
- Rules need to be established
- If it isn't enforced, it doesn't exist
- Will contain ***In Kiwi.com*** notes, describing real world examples of these policies

Approaching problem slowly

- Rapid changes do not help stability
- It worked so far, keep it working
- Incremental steps, with time to adapt

Code Review Rules

- Enforce code review
- Require tools to pass, add CI if possible
- Split responsibility 1:3
- Reduce bus factor
- *In Kiwi.com: Enforced via CI*

Code Review Best Practices

- Blameless
- Impersonal
- Triple tier system
  - Overall scope
  - System scope
  - Code scope

Education is most influential

- Make sure devs understand the why
- Enforce better documentation
  before and after change
- Explain architecture and direction
- *In Kiwi.com: Inhouse talks, education budget,*
  tech-writing culture, public discourse

There is no Easy Victory

- Easy wins are a step
- Quality increases slowly
- Tools don't replace engineering
- *In Kiwi.com: Attention to Product, not Project*

Code is written to be replaced

- Best code can be rewritten easily
- Less interdependent, better
- Allow easy reuse, allow easy replacement
- *In Kiwi.com: MVP prior to product*

How does Code Debt hurt

- Code debt is real debt
- Eventually, things will crash down
- Mistakes happen more often
- Implementation is slower
- *In Kiwi.com: Minimum Micromanagement*

Philosophy: Conclusions

- Low quality code is often a symptom
- Go for the cause, step by step
- Consistency is more important than bursts
- No easy victory

Conclusions

- Old code tells a story
- The story needs to modernize, not disappear
- Grab the easy boosts
- Rewrite current failures in bubbles
- Mantain quality going forward

Thank you for your attention.

Find me at: tinmarkoviccs@gmail.com
Alternatively: linkedin.com/in/tin-markovic/

Questions?