High-Quality Code

Petr Zemek

Lead Software Engineer at Avast
Threat Intelligence at Threat Labs
petr.zemek@avast.com
https://petrzemek.net, @s3rvac

Practical Aspects of Software Design, BUT FIT, 2022-05-04

Outline

https://bit.ly/374qs8m

Introduction

A tale of two libraries

1. The first library:
   - Very poor documentation
   - Trouble with compilation because of missing dependencies
   - Segfaults when given a file without an extension
   - Calls `exit()` when encountering an error
   - Sometimes prints output to `stdout/stderr`
   - Hard to change because of unreadable code and missing tests
2. The second library:
   - Amazing documentation
   - Seamless integration, automatic resolving of missing dependencies
   - Crystal-clear interface
   - Proper error handling and propagation of errors
   - Readable code that makes modifications a breeze
   - Code is completely covered by tests

Which one would you use? Which one do you write? ;-)

What is high-quality code?

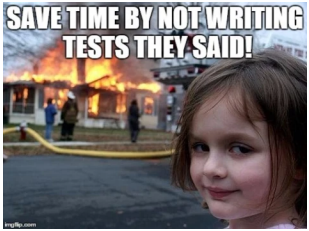
- Correct
- Robust, anticipates and handles edge cases and errors
- Safe and secure
- Well designed and organized without being over-engineered
- Readable, easy to change, allows sustainable development
- Testable and covered by tests
- Thoroughly documented
- Efficient without being prematurely optimized

Notes:

- Everything mentioned above is connected
- Code quality is not binary or absolute
- Perfection is not attainable

Why do we strive to write high-quality code?

- To satisfy our users and employers
- To save time and money
- To prevent catastrophes or security breaches
- Code is written once but read/modified many times
- To show that we are true professionals
- Thinking of your fellow programmers (or your future self)

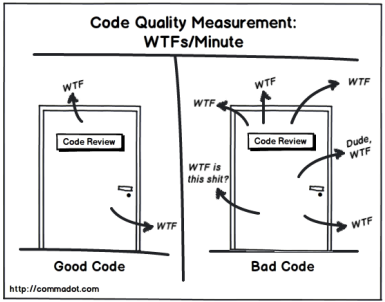


https://bit.ly/2RgMMU0

When you're trying to look at the code you wrote a month ago

https://bit.ly/3s4pdgK

How to measure code quality?



Another measure: How easy is to correctly change the code.

What to focus on

Code correctness

- What is correctness?
- Correctness with respect to what?
- Absolutely correct code
- Understand functional requirements
- Understand non-functional requirements

Code robustness and error handling

- One of the hardest parts of software development
- Robust programming
  - Paranoia
  - Stupidity
  - Cannot happen
- Be conservative in what you send, be liberal in what you accept
- Anything that might happen will happen, handle all edge cases
- Understand what might fail, handle all errors

```
int fclose(FILE *stream);
```

- Understand error-handling mechanisms
- Propagating errors upwards

Code safety and security

- Safety vs security
- Buffer overflows, crashes

```
char buf[BUFSIZE];
std::cin >> buf; // gets(buf);
```

- Thread (un)safety, common concurrency issues
- Resource leaks
- Improper handling of inputs

```
$id = $_GET['id'];
$sql = "SELECT * FROM users WHERE id = $id";
$result = $mysqli->query($sql);
```

- Understand common safety and security flaws

Code readability, extensibility, and maintainability

- Great, descriptive naming
- Consistency is key
- Split code into smaller functions/classes
- Keep code at a single level of abstraction

```
if ((currentDate() - user.getBirthDate()) >= Years(18))
// vs
if (user.isOldEnoughToDrink())
```

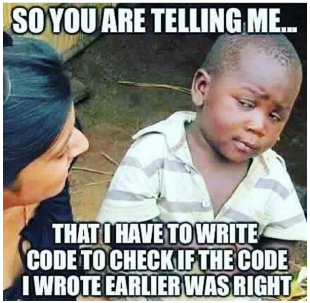
- Logical organization into functions, classes, etc.
- High cohesion, low coupling
- Comments explaining *why*
- Understandable is better than clever
- Learn design principles and patterns (e.g. SOLID, GoF)

Junior devs writing comments:

https://bit.ly/3krJzML

Code covered by tests

- Why do we write tests?
- Untested code does not work
- Selected types of tests:
  - Unit tests
  - Integration tests
  - End to end tests
  - Performance tests
  - Compatibility tests
- Code coverage
- Continuous integration (CI)
- Testing the UI
- Testing examples in the documentation



https://bit.ly/3nCt01e

Testability

- Code has to be testable
- Learning how to write testable code takes time
- An example technique: Dependency injection

```
public class Service {
    private DBConn dbConn;

    public Service(Config config) {
        dbConn = new PostgreSQLConn(config);
    }
    // vs
    public Service(DBConn dbConn) {
        this.dbConn = dbConn;
    }
}
```

- Tests improve your code
- Consider writing tests first

Documentation

- The bane of programmers
- Everybody wants to have it, nobody wants to write it
- User vs development documentation
- Important to keep up-to-date
- Although outdated documentation is better than no documentation

https://bit.ly/3vDMPbn

Knowledge of the used programming language(s)

- Syntax and semantics
- Abstractions
- Memory management
- Language idioms

```
i = 0
while i < len(items):
    print(items[i])
    i += 1
# vs
for item in items:
    print(item)
```

- Different implementations, OS specifics
- Common pitfalls
- Strengths and weaknesses, when to use a particular language

Knowledge of the used libraries

- Learn what is provided by standard libraries
- Thoroughly read and understand the documentation
- Know what libraries are available (or search)

https://bit.ly/3399X52

Knowledge of miscellaneous topics

Just a few examples:

- Regular expressions
- Floating point arithmetic
- Encodings
- Time zones
- Cryptography
- Commonly used protocols, such as HTTP, DNS, IP, TCP vs UDP
- Concurrency and parallelism, synchronization primitives
- Data structures and algorithms
- Databases
- Operating systems, HW

Interface design

- *Make interfaces easy to use correctly and hard to use incorrectly.*
  – Scott Meyers
- Your public interface should be crystal clear
- Aim for having a consistent interface

```
// Inconsistent position of parameters
int fputs(const char *s, FILE *stream);
int fprintf(FILE *stream, const char *format, ...);

// Inconsistent naming
#include <sstream>
std::stringstream s;

// Duplicities
size_type size() const;
size_type length() const;
```

Coding conventions

- Follow style guides and code conventions
  - Spaces vs tabs
  - Naming of variables (`snake_case` vs `camelCase`)
  - Code formatting in general (e.g. placement of curly braces, line wrapping)
- Uniformity is king
- Pay attention to detail
- Check typos and grammar in strings/comments

Performance

- What is an optimization?
- Typical optimization areas
  - Execution time
  - Memory usage
  - Response times
  - Throughput
  - Network communication
- Effectivity vs efficiency
- Golden rule: Do not optimize
- Understand trade-offs
- Always do profiling and perform benchmarks (avoid *pessimization*)
- Do not write needlessly inefficient code
- Know your language, compiler, operating system, architecture, etc.

Petr Zemek: Optimalizace kódu (BUT FIT, 2013)

Selected techniques

Pull requests and code reviews

The "lone wolf" workflow:

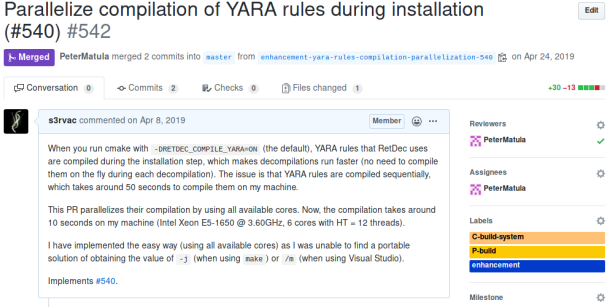
1. Put all your changes directly into `master`
   (There is no step 2)

A more cautious workflow:

1. Create a new branch from the current `master`
2. Implement the needed change there
3. Push the branch and create a *pull request* (PR) from it
4. Make the PR pass through a *code review* (CR)
5. The PR is approved and the branch is merged into `master`

What is a pull request (PR)?

- A request to review your changes and merge them
- Most commonly associated with PRs on GitHub:



https://github.com/avast/retdec/pull/542

- Note: Called a *merge request* (MR) in some systems

What is a code review (CR)?

- A process of looking at another person's code and checking if it is correct
- Consists of:
  1. Writing comments towards the code
  2. Giving evaluation (approve or request changes)
  3. Discussing comments with the author

Reasons for creating PRs and doing CRs

- Finding bugs and other defects
- Learning something new
- Increasing the sense of mutual responsibility within your team
- Finding a better solution
- Running automated checks before the code is merged
- Writing better code
- and more...

Petr Zemek: Pull requesty a revize kódu (IVS 2020)

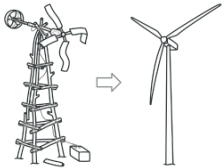
Pair programming

- Two programmers work together at one workstation
- Roles: driver and navigator
- Increased person/hours vs fewer defects
- Knowledge sharing
- Remote pairing
- Mob programming

https://bit.ly/3nyhDYe

Refactoring

- Restructuring existing code without changing its external behavior
- Code smells
- Improves maintainability and extensibility
- When to refactor
- Requires having tests
- Not all changes are refactorings
- https://refactoring.guru/refactoring



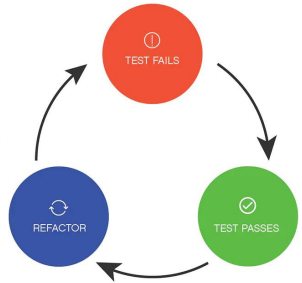
https://bit.ly/3e7fu2x

https://bit.ly/3vym48c

Test-driven development (TDD)

- A software development practice
- Clean code that works
- Leads to testable code
- Writing the interface you wish you had
- Seeing the test fail is important
- Do not refactor when your tests are failing
- Tests are already written when the code is finished

TDD Cycle

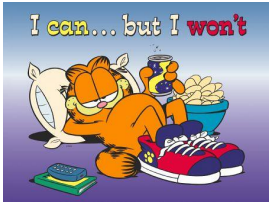


https://bit.ly/3gQntTm

Anti-patterns

What prevents programmers from writing high-quality code

- Inexperience
- Laziness
- Disinterest, unwillingness to learn
- Lack of sense for detail, sloppiness
- Bosses or coworkers
- Circumstances (e.g. deadlines)



https://bit.ly/3u6c1XB

https://bit.ly/3eL0qXh

Anti-pattern: Cargo cult programming

- A ritual inclusion of code that serves no real purpose

```
with open('file.txt') as f:
    data = f.read()
    f.close()
```

- Copy-and-paste programming
- Blind following of practices without understanding why
- Some cargo culting might be unavoidable

```
public static void main(String[] args)
```

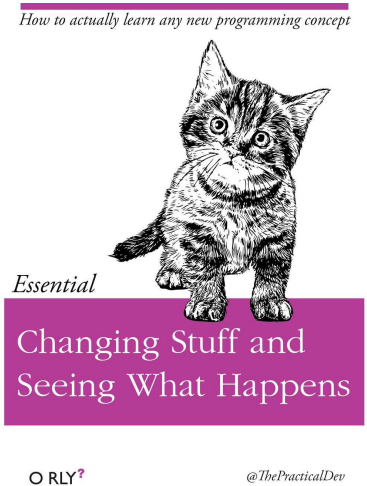
https://bit.ly/3gMTCLH

Anti-pattern: Voodoo programming

- Example: `if x > 1` (fail)
  - `if x >= 1` (fail)
  - `if x >= 0` (fail)
  - `if x < 1` (pass)
- Another example:

When your code compiles after 253 failed attempts

https://bit.ly/3ktYtC3



https://bit.ly/335UCIK

Anti-pattern: Not invented here (NIH) syndrome

https://bit.ly/3nApp3K

- Let's write our own HTTP library; how hard could it be?
- But by reinventing the wheel, I will learn! Or not?
- Possible issues with software licenses or patents
- Beware of blind inclusion of third-party projects (security)

Recommended reading and summary

Recommended reading

A. Hunt, D. Thomas: The Pragmatic Programmer (2nd edition), Addison-Wesley, 2019

In Czech: A. Hunt, D. Thomas: Programátor pragmatik, Computer Press, 2007

S. McConnell: Code Complete (2nd edition), Microsoft Press, 2004

In Czech: S. McConnell: Dokonalý kód, Computer Press, 2006

R. C. Martin: Clean Code, Prentice Hall, 2008

In Czech: R. C. Martin: Čistý kód, Computer Press, 2009

M. Fowler: Refactoring: Improving the Design of Existing Code (2nd edition), Addison-Wesley, 2018

In Czech: M. Fowler: Refaktoring: Zlepšení existujícího kódu, Grada, 2003

K. Beck: Test Driven Development: By Example, Addison-Wesley, 2002

In Czech: K. Beck: Programování řízené testy, Grada, 2004

S. H. Huseby: Innocent Code, John Wiley & Sons, 2004

In Czech: S. H. Huseby: Zranitelný kód, Computer Press, 2006

A bit of harmless self-promotion (my blog posts)

- Petr Zemek: Čistý kód, který funguje (2009-10-24)
- Petr Zemek: Vysoce kvalitní kód (2014-04-18)
- Petr Zemek: Důvody, proč psát jednotkové testy (2014-06-20)
- Petr Zemek: Zakomentovaný kód (2014-11-02)
- Petr Zemek: Proč psát kód na jedné úrovni abstrakce (2015-02-21)
- Petr Zemek: Udržitelný vývoj (2015-03-15)
- Petr Zemek: Proč rozlišovat jednotkové a integrační testy (2015-04-18)
- Petr Zemek: Proč vytvářet funkce (2019-07-27)
- Petr Zemek: Série "Chyby v návrhu"
- Petr Zemek: Série "Ještě jednou a lépe"

Summary

- High-quality code provides many benefits
- We (as professionals) should strive to write high-quality code
- There are many aspects of high-quality code
- There are techniques that can help us achieving high-quality code
- There are also anti-patterns that hinder our efforts
- Many books have been written on this topic
- Code quality is not binary or absolute
- Perfection is not attainable