Software Quality Metrics (SQM)

Abstract:
In the course, students will learn different approaches to software quality, starting from the most traditional ones, like McCabe's complexity and Halstead effort, and finishing with most recently developed, based on Neural Networks and Large Language Models (LLM).

What is the goal?

Writing code that compiles is easy. Writing code that is easy to read is a much bigger challenge. Without a proper understanding of what maintainability and software quality mean and why they matter, it may be hard or even impossible to become a professional programmer, capable of designing large software systems. At this course, we will attempt to investigate this very subject in order to increase the level of students' professionalism.

Who is the teacher?

Yegor is developing software for more than 30 years, being a hands-on programmer (see his GitHub account with 4.9K followers: @yegor256) and a manager of other programmers. At the moment, he is a director of an R&D laboratory in Huawei. His recent conference talks are in his YouTube channel. He also published a few books and wrote a blog about software engineering and object-oriented programming. He previously taught a few courses in Innopolis University (Kazan, Russia) and HSE University (Moscow, Russia), for example, SSD16 (2021), EQSP (2022), PPA (2023), COOP (2023), and PMBA (2023) (all videos are available).

Why this course?

Unfortunately, the quality of software is not getting better in the industry of software development while the amount of code we programmers write every year steadily grows. There is an urgent need to help programmers understand their craft better through the perspective of quantitative and qualitative analysis of software code.

What's the methodology?

There are 24 lectures, each dedicated to one (or a few) metrics that were designed to measure the quality or quantity of code over the last few decades of software engineering and computer science. Students will hear the theories behind the metrics accompanied with the stories experienced by the lecturer in his practical participation in software projects.

Course Structure

Prerequisites to the course (it is expected that a student knows this):

- How to write code
- How to design software
- How to use Git

After the course, students *hopefully* will understand:

- How to measure code size?
- How to keep complexity under control?
- How to simplify code for the sake of readability?
- How to motivate programmers write code of higher quality?
- How to reduce coupling between modules?
- How to aim for higher cohesion?
- How to convince a manager that quality matters?
- How to decrease code duplication?
- How to reduce the amount of dead/unused code?
- How to fight with tech debt?
- How to automate software analysis?
- How to employ mutation testing to increase quality?
- How to comment code correctly?
- How to automate builds effectively?
- How to check code style?
- How to find bugs without running the code?
- How to keep Git history in order?
- How to use Machine Learning for code analysis?
- How to employ LLMs for code quality analysis?

Lectures & Labs

The following 80-minute lectures constitute the course:

1. Lines of Code
2. McCabe's Cyclomatic Complexity
3. Cognitive Complexity
4. Halstead Complexity
5. Maintainability Index
6. Coupling
7. LCOM 1, 2, 3, 4, 5
8. TCC and LCC
9. CAMC and NHD
10. Object Dimensions
11. Clone Coverage
12. Dead Code
13. Code Churn
14. Tech Debt
15. Code Coverage
16. Mutation Coverage
17. Function Points
18. Defects Density
19. Comments Density
20. Commits Density
21. Builds
22. Code Style
23. Static Analysis
24. Neural Metrics

At laboratory classes, organized by a Teaching Assistant (TA), students either make pull requests to GitHub repositories suggested by the teacher, or write sections for their research papers.

Most probably, one of the following repositories will be suggested by the teacher for contribution during the course: yegor256/cam (Bash, Python), yegor256/rultor (Java, XML), yegor256/qulice (Java), cqfn/jpeek (Java, XML), and objectionary/eo (Java, XSLT).

Grading

At the first lecture, students form groups of **2–3** people in each one (no exceptions!). Each group picks a research topic from the list suggested by the teacher.

Each group writes a research paper in LaTeX, according to the guideline. The length of the pager may not exceed four pages in acmart/sigplan 10pt format (including references and appendices). The paper must be presented to the teacher incrementally, section by section: 1) Method, 2) Related Work, 3) Results, 4) Discussion, 5) Conclusion, 6) Introduction, and 7) Abstract. Once a section is approved by the teacher, the next section may be presented for review.

After a presentation of a section, the teacher may ask the group to stop working with the paper. In this case, no sections may be presented for review any more: they all will be rejected. This decision is subjectively made by the teacher and will not be explained to the students, however the following may contribute to such a negative decision: a) ChatGPT, b) plagiarism, c) negligence, and d) laziness.

When the Abstract is accepted by the teacher, a group may ask a student from another group to review their paper, according to this guideline. The review must be accepted by the TA.

There is no exam at the end of the course. Instead, each student earns points for the following results:

Result	Points	Limit
Attended a lecture	+2	8
Attended a lab	+2	8
Merged a pull request to a suggested repo	+4	48
Reviewed a paper of another group	+3	9
“Method” section (guideline)	+8	
“Related Work” section (guideline)	+12	
“Results” and “Discussion” sections (guideline)	+12	
“Conclusion”	+6	
“Introduction” section	+6	
“Abstract” and Title	+4	

Then, 55+ points mean “A,” 47+ mean “B,” and 23+ mean “C.”

An online lecture is counted as “attended” only if a student was personally presented in Zoom for more than 75% of the lecture’s time. Watching the lecture from the computer of a friend doesn’t count.