CODE QUALITY ANALYSIS SYSTEM BASED ON GENERAL UML DIAGRAMS WITH SUPPORT OF MACHINE LEARNING

Ostapenko D.S., master of PE department,
e-mail: denys.ostapenko@nure.ua
Academic adviser: D. Sc., Professor Andriy L. Yerokhin
Kharkiv National University of Radio Electronics

The suggested system provides an ability to evaluate the quality of given codebase drawing on well-known rules and principles like S.O.L.I.D, Clean Architecture, etc. A performed analysis is visualized in generated main UML diagrams which indicate clearly whether any rule was violated, thereby the amount of badly written code should be reduced dramatically.

Application development involves creating a computer program, or set of programs to perform tasks, from keeping track of inventory and billing customers to maintain bank accounts, speeding up business process and, in fact, even improving application effectiveness. Unlike vanilla programming, application development involves higher levels of responsibility (particularly for requirement capturing and testing) [1]. But the main parts of each software development phase process are design and coding.

The design represents the process of discussing the software architecture and documenting of it. During a design phase, a team headed by an architect clarifies software requirements, chooses appropriate technology and builds the basic architecture using different diagrams, flowcharts, and other auxiliary resources.

The coding phase follows the design one and usually continues much longer time. Basically, the coding is the process of writing a program code by developers in order implement some features or prepare the ground for implementing other functionality. Here are the main criteria which affect the code quality:

1. Time. As the time flies, each day new technologies, frameworks, development practices, and approaches are coming so it becomes quite important to maintain existing applications;

2. Team members experience and skills. The more skillful a team is the better code quality is fair to expect. Overall team qualification is the crucial attribute;
3. Established and adjusted the coding process. Usually, this derives from the item # 2. Customized coding practices, regular detailed code reviews, and knowledge sharing sessions help to improve an overall codebase quality.

In fact, the reality is far from ideal and as a result developers don't have much time for adapting existing code base to some modern technologies, development teams are not staffed properly (lack of seniors) and inside them there is no precise development process which would cover the code quality issue. Mostly, software developers can not do pretty much in this situation and that is why the code quality remains the same or even is worsening over the time.

Basing on the stated above, it's possible to emphasize the main motivation of the system - providing an ability to control and evaluate the code quality of given codebase so that aforementioned risks related to code quality could become negligible. Here is the detailed explanation how this system is solving the issues.

1. Time. Let's say there is a version A of some language and assume that version B has drastic improvements related to Open-Closed principle, so it does make sense to create rules which would simplify the migration process and it would be possible to fed them into Machine Learning software so that respective diagrams will be generated and team can start to plan their further refactoring process accordingly;
2. Probably, this software will not turn junior developers into seniors instantly, but it definitely helps with seeking and identifying potential problems. As a bonus it can help junior developers to learn and understand new practices a bit faster;
3. Established and adjusted the coding process. Since the better part of code quality issues could be covered programmatically a senior developers are able to spend less time reviewing a regular code changes and focus on challenging tasks or global improvements.

Also, Machine Learning system could be expanded with some custom rules and standards, this should help teams to stay on the same page and experiment with new coding ways.

Hopefully, Machine learning technologies had a terrific development boost in past few years and it become possible to make sophisticated analysis and predictions.

The algorithm how this system evaluates the code quality and builds diagram is illustrated in figure 1.

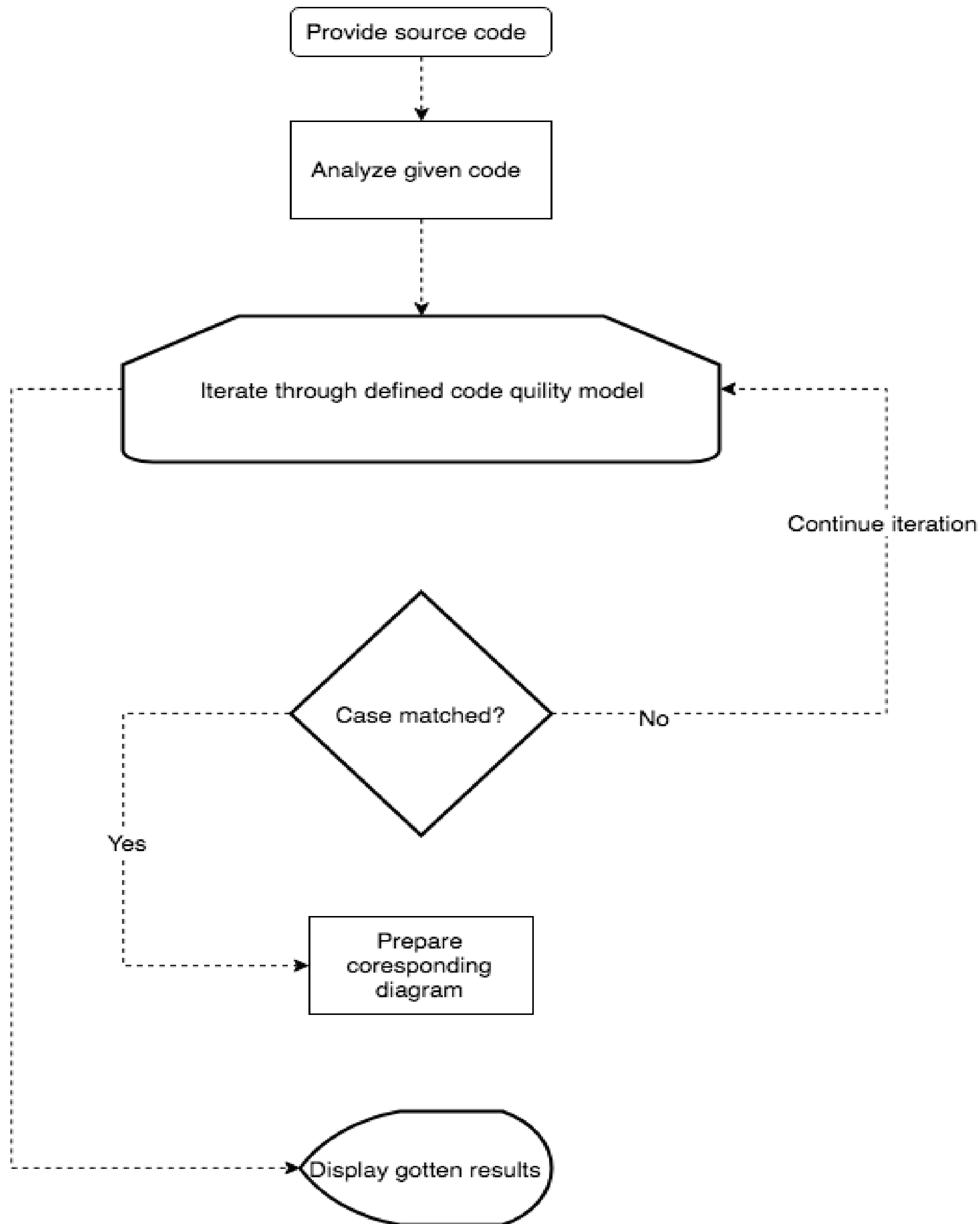

Figure 1 - Code quality evaluation simplified algorithm

Let's look step by step at provided flowchart in more detail.

1. Ideally, analysis shouldn't be applied to the whole project because that can be prone to delays and low efficiency, so taking that fact into account, the preferable way of usage of this system is handling of moderate changes and that is making this system a very good candidate for embedding into web-based hosting services for version control Git like Github or Bitbucket. The command language interface tool will be supported as well.
2. Once a user provided the code the analysis phase starts. Basically, analysis consists of parsing given codebase in order to get abstract syntax trees and then the matching the received trees with already defined models which represent either good practices or bad practices (antipatterns). All good or bad patterns and practices are represented as sets of abstract syntax trees and relations between them. In order to find the violation of some architectural principle or usage of an antipattern the system will be using decisions tree technique[2]. The system contains metadata for each case so that it will choose an appropriate diagram to show found code issue. Supported diagrams are: Class diagram, objects diagram and calls graph[3].
3. After the analysis is completed and data is gathered the system will output generated diagrams, mark there all founded issues so that developers will be able to investigate generated charts and clearly understand the code issues.

Also, there is a pivotal feature which will be available in future releases, this is an auto fixing. Once the system has found some problem in given codebase it can try to refactor the code according to 'good models' so that identified problem will be fixed. That feature should drastically improve the overall code quality and significantly reduce the number of delays in software development.

Since any Machine Learning based system is not able to give precise answers out of the box[4] and must be trained on some real-world data this system can complement already existing collections of

defined models with slightly altered versions of users models, as a result, the system will catch more issues and diagrams will show more detailed and articulated messages and at the end of the day exactly this capability will make possible to provide auto fixing.

Making a conclusion it's worth to note that currently there are no analogs on the market which could provide close functionality so there is a leaves a huge potential for this project.

Sources:

1. Kit, Edward (1992). Software Testing in The Real World. Addison-Wesley Professional. ISBN 0201877562.
2. Kamiński, B.; Jakubczyk, M.; Szufel, P. (2017). "A framework for sensitivity analysis of decision trees". Central European Journal of Operations Research
3. Ryder, B.G., "Constructing the Call Graph of a Program," Software Engineering, IEEE Transactions on, vol. SE-5, no.3pp. 216– 226, May 1979
4. Trevor Hastie, Robert Tibshirani and Jerome H. Friedman (2001). The Elements of Statistical Learning, Springer. ISBN 0-387-95284-5.