Behavioral Software Engineering for Code Review Tools

WASP Software Engineering Assignment

Lo Heander, lo.heander@cs.lth.se

1 Introduction

The DAPPER project's goal is to answer the research question **"how can code reviews be made fit for purpose?"**, which is a question that is more and more relevant today with more and more of software development being done remotely to some degree. The advent of AI-supported development tools like Copilot[1] and ChatGPT[2] also shifts the focus for developers even more from writing new code to reviewing code and integrating it into their code bases.

Code review is a task with a high cognitive load[3], requiring the reviewer to focus, hold the code in their memory, look for problems and connections, keep the goal of the merge request in mind and also take decisions around writing comments and approving or rejecting the merge request. This limits how long a developer can keep doing code reviews and how many they can complete before they are fatigued and the benefits decrease.

Despite these problems, code review tools have changed surprisingly little since the first software tool for code view, ICICLE[4], was introduced over three decades ago. More modern tools have colored diff-views and run in the browser, but ICICLE already had many of the core features such as the central diff view, the ability to add comments, support for annotations by static analyzers and support for working in a distributed network environment. This, I think, indicates a superlativist assumption[5] where one tool is assumed to be the best regardless of context. It favors certain kinds of software development, for example writing code over UI prototyping and promotes a file-centric view with a lexical diff over looking at code changes through the lens of exported APIs, inheritance structures or execution flow.

To explore these goals and the research question I use an interdisciplinary approach combining computer science with methods from sociology and psychology such as ethnography[6], grounded theory[7] and distributed cognition[8]. I believe that to find the answers here, the practices, tools, interactions and patterns *needs* to be observed from an interdisciplinary perspective to raise the perspective to a higher level. To observe the effects and the outcomes and acknowledge that these systems work in interaction with a team and with their process for version control, continuous integration, code review, code merging, deployment and maintenance.

2 Software Engineering principles from lectures

2.1 Behavioral Software Engineering

The core perspective of Behavioral Software Engineering is to focus on the people involved in developing, maintaining, using, specifying, deploying and testing software systems. The argument is that since these people are human and not rational, focusing on the technical aspects or the process is not enough. The individuals, groups and organizations needs to be an important factor in how the systems are engineered and having this as the main focus can be at least as effective as focusing on the technology if not more so.

This brings in many ideas and concepts from psychology into the software engineering field. A key figure in this movement is Professor Daniel Kahneman who have made important contributions within the theories of loss aversion, prospect theory, hedonic psychology and more. While Kahneman applied these theories mainly to economics, they have a reflection in the behavior of the people involved also in software engineering. By better understanding the humans involved, we can better understand the dynamics and motivations that

drive software projects to succeed or fail.

These aspects are central to my research on trying to make the specific software process of code review fit for the humans involved in it. To find and understand the effects of both the code review process and the tools used on software quality, software teams and collaboration within an organization the focus must be on the humans involved. This, I believe, is best studied using methods from the social sciences and psychology in order to understand the tools and processes not just as a list of features but as a cause and effect relationship on the individuals and their organizations.

2.2 Quality Assurance in Software Engineering

In the lecture about Quality Assurance in Software Engineering, Robert Feldt starts by introducing the difference between Verification and Validation. Verification is checking whether we are building the product right and can be acheived for example through automated software testing and code reviews. Validation on the other hand is about checking whether we are building the right product, which brings us into the domain of Requirements Engineering.

To validate a system we must ensure that the system fulfills a real need and is possible to be deployed, maintained and used successfully. For ML-systems, this will often extend to also validate the input data used for training and benchmarks and make sure that it is representative and corrected for biases that could make the system have undesired unfairness or poor performance.

For efficient verification, one technique stood out as particularily interesting to me: Property-based Testing. The idea is to combine invariant parameterised properties of a system with generation of random test data. This makes it possible to express the expected behavior in a compact way that is easy to understand, and still create and test thousands of test cases to be able to find corner cases where the expected properties break. In for example autonomous driving, I have seen this combined with machine-learning algorithms that try to predict the most tricky data possible and generate a higher density of random values around these boundaries to really challenge the system and improve its robustness [9].

3 Software Engineering principles from guest lectures

3.1 Levels of assistance and automation

Per Lenberg held a very interesting guest lecture explaining why Behavioral Software Engineering was especially interesting and necessary to create and maintain the complex systems built by SAAB Air Trafic Management. He highlighted how organizational charts may look simple and heirarchical, but in practice they are full of hidden interpersonal connections that are what truly defines how the organization behaves.

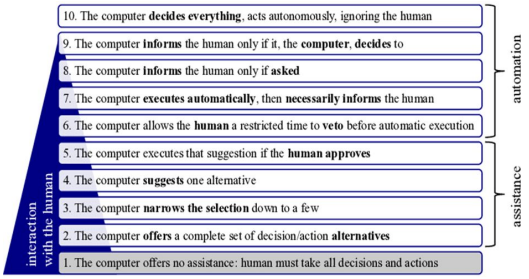

Figure 1: Automation levels by Per Lenberg

Lenberg also talked about automation levels (figure 1), ranging from level 1 where the computer does nothing at all up to level 10 where the computer takes and executes all decisions without involving the humans at all. It is an interesting discussion both what level of automation that is desired and safe for an application such as Air Trafic Management, but also applied to something like code review, testing and software engineering.

3.2 System Verification using AI-based systems

Dhasarathy Parthasarathy from Volvo introduced how they used an AI-based system to generate new test cases for system verification. They used large language models to analyze the documentation and system specifications across the different subsystems involved in the operation of a truck and translate for example different API-names and constants and then automatically write test cases.

I do find the approach a bit odd. In an environment that is already very complex and safety critical, such as for a truck driving on the highway during a long-haul delivery, it seems counter-intuitive to introduce another complex system that is even harder to explain and verify. If there are systematic faults or misunderstandings in the tests generated that don't cause the tests to fail but rather to enforce unwanted behavior, how will that be discovered? How can the test generation system be re-trained to correct for this? In many ways, it ties into the discussions raised under Lenberg's lecture (section 3.1): what level of automation is appropriate, safe and desired for which tasks?

4 Related work

4.1 Design Patterns for AI-based Systems

"Design Patterns" is a concept in Software Engineering that describes and names proven and reusable solutions to frequently occuring problems [10]. In my first chosen research paper, Heiland et al. [11] presents an multivocal literature review over both traditional Software Engineering Design Patterns that are applicable to AI-based system, as well as new emerging patterns that are unique to, or more relevant to, these systems. The authors motivation for the study is that since there are indications that design patterns improves the overall software quality, they are likely to give similar effects on system quality for AI-based systems as well. Having a comprehensive overview over possible patterns will help both in future research of this as well as when implementing and refactoring AI-based systems.

Heiland et al. finds 70 design patterns described both in academic and practitioner literature. Roughly half of these, 34 patterns, are specific to AI-based system while the rest are applicable to both AI-based system and traditional software systems. An interesting finding is that these categories of patterns address different areas of the system quality. Traditional design patterns are applied predominately to improve process and implementation aspects while new AI-system-specific patterns focus on improving security, safety and deployment aspects. The authors see this as an argument in the debate around if there is a need for a new field of ML Engineering distinct from Software Engineering. If around half of the patterns are specific to AI-based systems and also address aspects unique to this domain, that is an indicator on the differences between traditional software systems and AI-based systems are large enough to merit their own research fields and professional disciplines.

Techniques and best practices to improve system quality is closely related to my research field. Because code review in itself is (among other things) a quality assurance tool, identifying proven design patterns in the code under review could help the reviewers understand and critique the code. In the paper, Heiland et al. presents a web-based application they developed to make the patterns accessible and easily browsable[1]. Through this app, the design patterns are stored and described in a machine-readable JSON format. This could assist in building tools that both gives developers access to this database during code reviews or automatically tries to identify and mark out the patterns in the code under review.

Further, my research project also aims to propose improvements in code review tools that address the shortcomings and frictions found by both other researchers and in my initial studies. One important avenue to explore here is how to leverage AI-based system to assist software developers before, during and after code reviews. To make these systems perform well it would make sense to use established design patterns for both implementation, architecture, security, and so on.

4.2 SOLID Design Principles Applied to Machine Learning Code

SOLID design principles are principles for improving the design of object-oriented software [12] that are well-know and widely applied in the industry. In my second chosen CAIN research paper Cabral et al. [13] studies if applying these principles also improves code understanding of Machine Learning code. The acronym stands for **S**ingle-responsibility principle, **O**pen-closed principle, **L**iskov substitution principle, **I**nterface segregation principle and **D**ependency inversion principle. When applying all of them, the goal is to get code that is both

[1] `https://swe4ai.github.io/ai-patterns/`

easy to read and possible to refactor and extend without extensive modification.

The study was structured as a controlled experiment where code for training and evaluating machine learning models was presented both as it was first written and then in a refactored variant applying the SOLID principles. The participants each got a code variant selected randomly and rated the code on a scale of how hard the code was to understand. To get a baseline for the participants familiarity with the SOLID principles, they also rated their agreement with each of the principles.

The result show a statistical significance for the SOLID principles improving the readablility of the ML-code and the participants rating it as easier to understand. While this is not very surprising in itself, I do feel that there is an opinion in parts of the ML community that ML code is special, more experimental and should have an emphasis on being fast to write rather than easy to read or understand. This is in many ways similar to how some programmers thought about software in the early days of professional software development. That it was supposed to be hard, and if you didn't understand the code you simply were not smart enough. Over time, traditional software have matured and realized that all code is read hundreds of times more than it is written and the person not understanding it is often the same person who wrote it, just a few months later and after forgetting the context or the clever tricks they used.

In the end, code that has a clear design and is easy to read, extend and reason about is always an asset. And often not harder or slower to write in the first place. Many of the examples in the study show simple things such as breaking up a huge training code block into functions that handle each separate part of it and breaking out a controller that manages the overall flow. Not only does this make the code easier to read, it is also much easier to re-use controllers, training loops and evaluation functions for the next ML project or to compare different models with each other.

In the end, I think this is also an argument in the debate over ML Engineering versus Software Engineering. While there are definitely challenges and solutions that are unique to AI-based systems, there is also much to learn and apply from traditional software engineering. After all, the code for data analysis, training and evaluation is just software. And so is much of the services and systems gluing together a trained model into a product or service trying to fulfill a need out in the world. Considering best practices both from traditional software engineering and putting the humans front-and-center through behavioral software engineering has the potential to unlock the true performance and usefulness of these systems.

References

[1] C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, and I. Gazit, "Taking flight with copilot: Early insights and opportunities of AI-powered pair-programming tools," vol. 20, no. 6, pp. 35–57. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[2] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of ChatGPT."

[3] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information Needs in Contemporary Code Review," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, pp. 1–27, Nov. 2018.

[4] L. Brothers, V. Sembugamoorthy, and M. Muller, "ICICLE: Groupware for Code Inspection," in *Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work*, CSCW '90, (New York, NY, USA), pp. 169–181, Association for Computing Machinery, 1990. event-place: Los Angeles, California, USA.

[5] T. R. Green, M. Petre, R. K. Bellamy, *et al.*, "Comprehensibility of visual and textual programs: A test of superlativism against the'match-mismatch'conjecture," in *Empirical studies of programmers: Fourth workshop*, vol. 121146, Ablex Publishing, Norwood, NJ, 1991.

[6] H. Sharp, Y. Dittrich, and C. R. B. de Souza, "The Role of Ethnographic Studies in Empirical Software Engineering," *IEEE Transactions on Software Engineering*, vol. 42, pp. 786–804, Aug. 2016.

[7] S. Adolph, W. Hall, and P. Kruchten, "Using grounded theory to study the experience of software development," *Empirical Software Engineering*, vol. 16, pp. 487–513, 2011. Publisher: Springer.

[8] E. Hutchins, *Cognition in the Wild*. MIT press, 1995.

[9] Q. Song, E. Engström, and P. Runeson, "Industry practices for challenging autonomous driving systems with critical scenarios," *ACM Transactions on Software Engineering and Methodology*, vol. 33, Apr. 2024. Publisher Copyright: © 2024 Copyright held by the owner/author(s).

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, pp. 361–388. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[11] L. Heiland, M. Hauser, and J. Bogner, "Design Patterns for AI-based Systems: A Multivocal Literature Review and Pattern Repository," in *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, (Melbourne, Australia), pp. 184–196, IEEE, May 2023.

[12] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[13] R. Cabral, M. Kalinowski, M. T. Baldassarre, H. Villamizar, T. Escovedo, and H. Lopes, "Investigating the impact of solid design principles on machine learning code understanding," in *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, CAIN '24, (New York, NY, USA), p. 7–17, Association for Computing Machinery, 2024.