Architectural Guardrails: An Opinionated Framework for Preventing Structural Decay in Greenfield Research Software

Art O'Liathain
22363092@studentmail.ul.ie
University of Limerick

December 2025

Keywords Evolutionary Algorithms · Genetic Programming · Grammar Evolution · Symbolic Regression · Concurrent Computing

List of Figures

List of Tables

1 Introduction

This is just notes for into and paper structure

- Talk about code quality as a whole
- Talk about issues in research funding cycle
- Talk about how pervasive technical debt is
- Talk about reproducibility paper -> Mention the affect this has on the validity of papers

The core idea of the debt analysis is a 3 step process Identify high risk/churn code areas, use a tool maybe machine learning to analyse and understand the code, derive rules from both texts such as architectural smells that are correlated to high *debt interest* [1]

1.0.1 Research question

Can opinionated objective rules be derived from historical and static analysis on code repositories

Do these rules notably increase code quality

1.0.2 Thoughts on structure

Instead of one literature review, I'm almost thinking of stages so I do one review now for identification methods and then another for creating an AST analyser as well as the methodology since both feel they could've been a mini thesis in their own right and if I don't do it this way i feel it'll be very very front loaded

2 Background on Identifying technical debt

- Maintainability
- Technical Debt
- State of Research software
- Importance of architectural debt
- Current methods to identify debt (Smells and [1])
-

2.1 Maintainability

Maintainability is term used frequently in software engineering, there is no definite definition on what maintainability is but ISO25010 defines it as "The degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements." It defines that the sub sections of maintainability are *Modularity, Re-usability, Analysability, Modifiability, Testability.* This is a very broad definition which simply means how easy can the system be modified for change. Due to the broadness of the definition it can easily be used to bring other non functional requirements as sub requirements ie. Flexibility, Adaptability, Scalability, Fault tolerance, Learn ability. The lack of clear testable outcomes for each quality in the ISO standard leads to a conceptual overlap where maintainability becomes a 'catch-all' category for non-functional requirements.

2.1.1 What does it mean to be maintainable?

To label code as maintainable, modularity is a key factor in that assessment. Modularity done correctly facilitates changeability through logical decomposition of functionality. This may seem simple but this area if done incorrectly can have long lasting consequences on the maintainability of a codebase. D.L Parnas presented a study revealing how easily developers could fall into the pitfall of designing the system in a human like manner [2]. Parnas argued that designing systems following a flowchart of logical operations is a method that creates code that is not resilient and while modular is highly coupled which hurts the maintainability of the codebase. The core purpose of modularisation shouldn't be encapsulating subroutines into modules. Design decisions that are likely to change should serve as the foundational points to creating modular code, allowing subroutines to compositions of modules. This will serve as a foundational idea on how maintainability will be measured in this paper.

2.1.2 Measuring Maintainability

Because maintainability is an abstract concept, various frameworks have attempted to reduce it to a concrete numerical metric. An early approach in this area is the Maintainability Index (MI), which calculates a score based on a weighted combination of Halstead Volume, cyclomatic complexity, lines of code (LOC), and comment density [3]. The formula typically utilized for this assessment is:MI = 171 - 5.2 ln(V) - 0.23(G) - 16.2 ln(LOC) + 50 sin(sqrt{2.46 times C}) Where V represents Halstead Volume, G is Cyclomatic Complexity, and C is the percentage of comment lines.

- Need to mention how the formula is likely outdated but even a modernisation is not the solution

The Oman study validated this metric through feedback from 16 developers and empirical testing on industrial systems, including those at Hewlett-Packard and various defense contractors. This methodology provided a pragmatic tool for engineering managers to prioritize maintenance efforts by assigning a tangible value to code quality. While the MI provides a high-level overview of code density, it suffers from what can be described as “semantic blindness.” Metrics such as Cyclomatic Complexity and Halstead Volume analyze the control flow and token count of a file but fail to interpret the structural intent or the relationships between components.

In regards to Architectural Technical Debt (ATD) a script may maintain a high MI score due to low complexity and short length, yet contain architectural flaws such as “God Objects” or tight coupling to external datasets that inhibit reuse. Because the MI only evaluates the “bulk” of the code rather than its structure, it cannot identify these high-interest debt artifacts.

A more modern approach was proposed by Xiao et al, of identifying architectural debt through evolutionary analysis of a codebase [1]. Where architectural debt can be used as a measure of maintainability albeit without a clear score. The paper proposed that there are four key architectural patterns that are the main proponents of ATD, Hub, Anchor Submissive, Anchor Dominant, and Modularity Violation. These patterns are all based on evolutionary dependencies between files, particularly those that are correlated only through commits and lack structural commonality. This was measured by preforming a pseudo longitudinal study on large open source codebases, using tools such as Understand and Titan to calculate commit coefficient between every file. A metric which relates to the chance of a commit on one file will require a commit on another. An example would be given files A, B and C, with a commit history of {A,B} and {A,C} the commit coefficients in relation to A are there is a 100% chance if B or C is modified A will be also, but if A is modified there is only a 50% chance that B or C will be modified. This would then be called a fileset, this would be extrapolated over the entire commit history and codebase creating many filesets. This would allow the tool to extract semantic relations between files that are not apparent structurally. The study was able to compound this effect by measuring the number of commits labeled as bug fixes against the number of feature commits, this allows simple data analytics to measure the amount of maintenance debt that each fileset would have. Using this method

Another more modern method of

2.1.3 Problems with Blind metrics

As this is a very wishy washy definition it makes more sense for this paper to define maintainability as "The inverse amount of technical debt to lines of code". This is a much simpler definition that is quantifiable, although it requires that technical debt be explained in detail for it to be truly understood.

2.2 Technical Debt

Technical debt (TD) was first defined as "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite… The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise." By Ward Cunningham [4]. In the following years research has shown TD is not a singular type of problem and there are many forms to it, the five most prevalent types of TD are **Design debt, Test debt, Code debt, Architecture debt and Documentation debt** this was extracted from 700+ surveys across 6 countries [5]. The artifacts used to identify design debt, code debt and architectural debt have significant overlap and these artifacts exhibit behaviours similar to Architectural Technical Debt (ATD)[1]. Similarly to Maintainability, this paper defines Technical debt in regards to design, architecture and code as ATD and will be the primary focus of this paper.

Technical debt is a pervasive problem in research software development [6] where does it compiles is the only quality tracked in the codebase. This means that technical debt accrues significant interest as time increases

Bibliography

[1] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*, Austin Texas: ACM, May 2016, pp. 488–498. doi: 10.1145/2884781.2884822.

[2] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972, doi: 10.1145/361598.361623.

[3] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Proceedings Conference on Software Maintenance 1992*, Nov. 1992, pp. 337–344. doi: 10.1109/ICSM.1992.242525.

[4] "c2.com/doc/oopsla92.html." Accessed: Feb. 02, 2026. [Online]. Available: https://c2.com/doc/oopsla92.html

[5] R. Ramač *et al.*, "Prevalence, common causes and effects of technical debt: Results from a family of surveys with the IT industry," *Journal of Systems and Software*, vol. 184, p. 111114, Feb. 2022, doi: 10.1016/j.jss.2021.111114.

[6] Z. Hassan, C. Treude, M. Norrish, G. Williams, and A. Potanin, “Characterising reproducibility debt in scientific software: A systematic literature review,” *Journal of Systems and Software*, vol. 222, p. 112327, Apr. 2025, doi: 10.1016/j.jss.2024.112327.