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. One could easily make an argument for most sub categories and even categories defined within the ISO 25010 standard [2] to be sub sections within maintainability with maintainability being the true goal of non functional requirements.

This naturally comes with the problem of how does one measure maintainability? There have been many different approaches taken to measuring maintainability.

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 defintion that is quantifiable, although it requires that technical debt be explained in detail for it to be truly understood.

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 [3]. 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 [4]. 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 [5] 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] "ISO 25010." Accessed: Feb. 02, 2026. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

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

[4] 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.

[5] 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.