# Assignment 4 - Experiment

## Abstract

This report presents a new method, called the TDR-W framework, for prioritizing technical debt (TD). It's designed to work better than old methods, which often get prioritization wrong (most do not have much practical impact, focusing more on the theoretical aspect, but there are other issues as well). The core idea is simple: debt risk is a combination of how much effort it takes to fix (the principal) and how much pain it's currently causing (the interest).

This idea is defined with a weighted formula:

TDR\_Score = (Effort to Fix) \* (wcommits \* Recent Commits + wbugs \* Recent Bugs)

This report provides a methodology for gathering these numbers by mining a project's coding history. It explains how to use the Lizard tool to guess the 'Effort', the PyDriller library to count 'Recent Commits', and a practical workaround (called HBFM) to find 'Recent Bugs' from commit messages. There's also a case study on several major open-source projects that shows how this formula helps find the real problem areas (or hotspots) and helps to safely ignore benign debt messy code that isn't actually hurting anyone.

## Chapter 1: The Case for a new TD Prioritization Model (Related Work)

### 1.1 The issue with Static Metrics: The Cost vs. Pain Problem

The main reason a new framework is needed is that current tools and metrics are not very good at predicting real problems. Many studies have tried to find a link between the messiness of code (measured by tools like SonarQube) and real-world developer problems (like how long it takes to deliver a feature). The results are consistently weak or inconsistent.

When they checked if messy code led to slower delivery times, they got mixed results. In some cases, it did, in others it had no meaningful impact and in two cases it actually had a negative impact. Their conclusion was clear: the relationship between code messiness (TDD) and delivery time (lead time) is either very weak or nonexistent.

This shows a big flaw in how debt is typically perceived. It proves that the cost of fixing debt (the principal) is totally separate from the pain it causes (the interest). If they were linked, the most expensive items could just be fixed first, but they're not.

A piece of code can be very expensive to fix but cause zero pain because no one ever touches it. This is why the TDR-W framework is built on the idea that "cost" and "pain" are two different things.

### 1.2 The Problem with Prioritizing

Beyond bad metrics, the whole field of "how to prioritize" is a challenge. Researchers who reviewed all the studies on this topic (a "systematic literature review") found that no one can agree on the best way to do it.

First, it's way too complex. One review found 53 unique, individual factors that different researchers use to prioritize debt.This makes it impossible for a real team to just pick a model and use it.

Second, and more importantly, these methods fail to answer the real question. Most methods only help to decide "which piece of debt should be fixed first?" But that's not the problem teams face. The real question is, "should this debt be fixed or should this new feature be built?"

Third, most methods ignore real-world limits. Another review found that very few methods (only 16.67%) even consider a team's limited resources, like time and developer availability.

This research shows a new framework is needed that is:

1. Simple: Not based on 53 factors
2. Contextual: Helps to weigh debt against new features
3. Measures both the static "cost" and the real-world "pain"

The TDR-W framework is designed to be exactly that.

### 1.3 What is Development Pain?

"Pain" isn't just a feeling; it's something that can be measured by looking at a project's history. The TDR-W formula breaks "pain" into two measurable parts, both of which are backed by research.

**Pain as unreliability (Justification for Recent Bugs)**

The best reason for using bug history comes from a 2016 study by MacCormack & Sturtevant. They suggested that messy, tangled code (high coupling) in the system's core was directly responsible for a huge number of bug reports. This gives a strong, citable reason to use Recent Bugs as a key part of the "Pain Score". It's a direct measure of where debt is causing the product to fail.

**Pain as friction (Justification for Recent Commits)**

Technical debt doesn't just cause big bugs, it also creates a constant tax on development. A 2018 study by Besker et al. measured this and found that developers waste, on average, **"23% of their total development time"** just working around technical debt. This time is wasted when developers are actively working in that part of the code, trying to understand it, writing extra tests and avoiding issues.

The Recent Commits metric (known as "churn" or "volatility") is a direct way to measure this friction. Debt in a file that hasn't been touched in years isn't wasting 23% of anyone's time. But debt in a core file that's changed every day is a hotspot that is hurting the team. Another research proposal by Gupta (2018) also supported this idea of linking TD to change-sets in the code's history.

## Chapter 2: The Technical Debt Risk (TDR-W) Model

### 2.1 Conceptual Framework: "Principal and Interest"

The TDR-W model is based on a simple analogy that makes it easy to understand and explain.

* **The Principal (Effort to Fix):** This is the one-time cost, in hours or days, to completely fix the debt and remove it from the code. This is a standard concept in TD management.
* **The Interest Rate (Pain Score):** This is the model's big idea. It measures the ongoing cost of doing nothing. It's the "interest" being paid every single day in the form of wasted time, developer frustration and customer facing bugs. A high Pain Score is like a burst pipe; it is flooding the room and needs immediate repair. A low Pain Score is like a dripping faucet; it is annoying, but it can wait until the weekend.

The TDR score is calculated by multiplying them: TDR\_Score = (Effort to Fix) \* (Pain Score).

This multiplication is the key. It automatically finds the real risk. It also correctly handles the two cases that static-only tools get wrong:

1. **High-Effort, Low-Pain Debt:** A complex, old module that no one ever touches.
   * Effort = 5000 (high), Pain = 1 (very low)
   * TDR\_Score = 5000. This is a low-priority item.
2. **Low-Effort, High-Pain Debt:** A simple, badly designed function that is changed daily and is full of bugs.
   * Effort = 50 (low), Pain = 100 (very high)
   * TDR\_Score = 5000. This is flagged as an equally important item.

The model correctly identifies and deprioritizes benign debt in stable, old code. If the Pain Score is zero, the TDR Score is also zero, no matter how big the Effort is. This acts as a pain filter, letting teams focus their limited time only on the debt that's actively hurting them.

### 2.2 The TDR-W (weighted) Formula

The specific formula is a weighted version (TDR-W), which adds weights to the Pain Score.

The formal TDR-W formula is:

TDR\_Score = Effort\_to\_Fix \* (wc \* Recent\_Commits + wb \* Recent\_Bugs)

Where:

* TDR\_Score is the final risk score
* Effort\_to\_Fix is the principal/cost
* Recent\_Commits is the proxy for development friction
* Recent\_Bugs is the proxy for product unreliability
* wc (Commit Weight) is the weight for friction
* wb (Bug Weight) is the weight for unreliability

These weights (wc and wb) are how a company's strategy gets put into the model. They address the research gap for models that consider value and resources constraint.

* A Commit (friction) mainly hurts internal developers (slowing down new features).
* A Bug (failure) mainly hurts external customers (damaging the product's reputation).

A company can set these weights to match its goals:

* **Aggressive startup (focus on speed).**
  + wc = 1.0, wb = 1.0
  + This policy says: "Friction is just as bad as instability; both are slowing us down."
* **Mature enterprise (focus on reliability).** This company (ex: in finance) must put stability first.
  + wc = 0.5, wb = 2.0
  + This policy says: "A customer-facing bug is 4x more harmful than developer friction. We will tolerate internal friction, but we will not tolerate external failure."

### 2.3 Academic justification of model components

Each part of the formula is backed by academic research:

* **Effort to Fix:** This is the principal. Including it is standard practice, as seen in the work of Seaman and Guo (2011).
* **Recent Commits:** This measures friction. It's based on the idea that debt in "frequently changed code is more painful" and aligns with research that links TD to change-sets in the code's history.
* **Recent Bugs:** This measures unreliability. It's strongly supported by MacCormack & Sturtevant (2016), who proved you can link architectural problems to defect-related activity.

## Chapter 3: MSR-Based Methodology for TDR Instrumentation (Modeling the experimental part)

This chapter provides the step-by-step guide for getting the three numbers in the formula. The entire process can be automated with a few Python scripts using the Lizard and PyDriller tools. (The python files are in the repository for this assignment).

### 3.1 Instrumentation 1: A Proxy Model for Effort to Fix using Lizard

Challenge: The formula needs an Effort to Fix score, which is a guess of how many hours it will take to fix something. Tools like SonarQube provide this, but it's often just a "rough estimate" made up by the tool's developers, not a scientific number. The specified tool, Lizard, does not provide this number at all. Lizard only gives complexity metrics like NLOC (lines of code) and CCN (Cyclomatic Complexity).

Solution: The Lizard Effort Proxy (LEP) Model:

A proxy (a good guess) for effort must be created, using only the numbers Lizard provides. This is a valid approach, as research has shown a link between refactoring effort and code complexity. One study by Higo et al. (2005) specifically proposed a way to estimate refactoring effects based on changes in complexity metrics. Fixing code involves the volume of code to be read and the complexity of that code. Lizard gives numbers for both.

The Lizard Effort Proxy (LEP) formula is proposed:

Effort\_Proxy = (Sum of NLOC from all functions) + (k \* Sum of CCN from all functions)

Where:

* Sum NLOC is the total lines of code in a module (proxy for volume)
* Sum CCN is the total cyclomatic complexity in a module (proxy for complexity)
* k is a complexity penalty (ex: k=5) that a team can set. It reflects that complex code is non-linearly harder to fix

**Proposed Methodology (for a Python script):**

1. A script runs the Lizard tool on the codebase
2. Lizard outputs a list of all functions and their NLOC and CCN values
3. The script will read this output and, for each function, figure out which module (or directory) it belongs to
4. It will add up the NLOC and CCN values for every module
5. Finally, it applies the LEP formula (Effort\_Proxy = Module\_NLOC\_Sum + (k \* Module\_CCN\_Sum)) to get the final Effort to Fix score for each module

### 3.2 Instrumentation 2: Measuring Recent Commits using PyDriller

**The Goal:** The Recent Commits (C) score is needed, which is just a simple count of how many times each module has been changed in a "recent" period (ex: the last 6 months).

All the code is in the python files.

### 3.3 Instrumentation 3: practical, code-based SZZ-Alternative for Recent Bugs

**Challenge:** The Recent Bugs score is needed. This means figuring out which commits are bug fixes. The standard academic way to do this is called the SZZ algorithm. However, SZZ is very complicated, slow, and not always accurate (and I had many technical issues with it because it's old and unmaintained). A full SZZ implementation has two steps: find the bug-fixing commit, then trace the fix back in time to find the original commit that introduced that bug.

Solution: Heuristic-Based Fix-Mapping (HBFM)

The TDR-W formula just needs to find hotspots of unreliability. A module that needed 10 bug fixes in the last 6 months is clearly a high-pain module. It is not necessary to know when the bugs were introduced.

So the complicated part of SZZ (Step 2) can be completely skipped. This alternative, Heuristic-Based Fix-Mapping (HBFM), only identifies a commit that looks like a bug fix and counts the files in it. This gives a great practical measure of reliability strain at a fraction of the cost.

**Chapter 4: Case study: real-world data analysis**

This chapter presents the results of applying the TDR-W framework to real-world data from several major open-source projects. The goal is to show how the framework's outputs create a risk-based priority list.

**4.1 Subject Selection: Open-source repositories**

To test the TDR-W framework, data was gathered from a diverse set of seven well-known open-source repositories: apache/commons-cli, apache/commons-lang, psf/requests, zxing/zxing, pallets/flask, facebook/react, and chartjs/Chart.js. This selection provides a wide range of project sizes, ages and complexities.

**4.2 Gathering the data**

Using the how-to scripts from Chapter 3, TDR-W data was generated for each repository. The analysis uses the formula from Section 2.2 with weights set to wc = 0.5 (for Commits) and wb = 1.0 (for Bugs). This formula states that a recorded bug is twice as impactful as general development friction.

**4.3 Results: TDR hotspot analysis**

The combined data from all projects provides the framework's main deliverable: a risk-ranked list of technical debt hotspots. Table 4.1 shows the top 10 highest-risk modules from the combined dataset, showing where development pain is most concentrated.

**Table 4.1: Top 10 TDR-W hotspots across all analyzed projects**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Project** | **Module** | **TDR\_Score** | **Effort** | **Pain** | **Commits** | **Bugs** |
| react | packages\react-reconciler\src | 6,024,431.0 | 59,354 | 101.5 | 171 | 16 |
| react | packages\react-server\src | 3,169,460.0 | 22,639 | 140.0 | 164 | 58 |
| commons-lang | src\main\java\org\apache\commons\lang3 | 2,767,800.0 | 31,632 | 87.5 | 145 | 15 |
| react | compiler\packages\babel-plugin-react-compiler\src\HIR | 844,984.0 | 10,696 | 79.0 | 136 | 11 |
| react | packages\react-client\src | 762,454.0 | 9,902 | 77.0 | 84 | 35 |
| react | packages\react-devtools-shared\src\backend\fiber | 645,028.0 | 13,724 | 47.0 | 80 | 7 |
| react | packages\react-devtools-shared\src\devtools\views\Components | 461,155.0 | 6,190 | 74.5 | 133 | 8 |
| react | packages\react-dom-bindings\src\client | 341,082.0 | 18,949 | 18.0 | 32 | 2 |
| commons-cli | src\main\java\org\apache\commons\cli | 315,849.5 | 6,133 | 51.5 | 89 | 7 |
| react | compiler\packages\babel-plugin-react-compiler\src\Validation | 293,002.5 | 5,581 | 52.5 | 101 | 2 |

**4.4 What the results tell us**

This TDR ranking gives critical insights that an Effort-only tool would completely miss.

Finding 1: Why looking at Effort alone is wrong

A traditional tool would rank debt by the Effort column. Looking at the commons-lang project, this would incorrectly rank builder (Effort: 7764) as a high priority, while ignoring function (Effort: 1501).

But the TDR analysis shows function (TDR\_Score: 45,780.5) is a big source of pain (Pain\_Score: 30.5 from 53 commits), while builder (TDR\_Score: 62,112.0) is very stable (Pain\_Score: 8.0). The TDR model correctly shows these are much closer in priority than their Effort scores would suggest.

Finding 2: Finding the Real Problem Areas

The Top 10 list clearly identifies the modules causing the most friction. The react repository is high on the list.

* **Rank 1 (react-reconciler):** This is the clear top priority, with a TDR\_Score of over 6 million. It has both the highest Effort (59,354) and a very high Pain\_Score (101.5).
* **Rank 2 (react-server):** This is a classic example of TDR-W's value. Its Effort score (22,639) is less than half that of Rank 1. However, its Pain\_Score is a staggering 140.0, driven by 164 commits and **58 recent bugs**. This is something that a static-only tool would completely miss. The TDR-W model correctly presents this high-pain module as a critical priority.

Finding 3: The Value of Ignoring Code That Isn't Hurting Anyone

The most important finding for managers is identifying debt that is expensive but not "painful". This benign debt can be safely ignored, saving valuable developer time. Table 4.2 shows examples of this.

**Table 4.2: Examples of benign debt (High-Effort, Low-Pain)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project** | **Module** | **Effort** | **Pain** | **TDR\_Score** |
| zxing | core\src\main\java\com\google\zxing\oned | 7,187 | 0.5 | 3,593.5 |
| Chart.js | src\plugins\plugin.filler | 1,282 | 0.5 | 641.0 |
| commons-lang | src\main\java\org\apache\commons\lang3\exception | 1,188 | 0.5 | 594.0 |

The zxing/oned module is a perfect example. It has a huge Effort score (7,187), so old tools would flag it as a top priority.

However, its Pain\_Score is 0.5 (from a single commit and zero bugs). The TDR-W framework calculates its risk score as a tiny 3,593.5. This is the framework succeeding. It has correctly identified this module as benign debt and gives the team a data-driven reason to ignore this debt, freeing up time to fix the actual hotspots.

## Chapter 5: Discussion and Future Work

### 5.1 Implications for Practitioners

The TDR-W framework isn't just a number; it's a tool for making better decisions. Its main purpose is to be the quantitative engine for the "Prioritization Matrix" mentioned in the original research (the previous assignment).

Teams can plot the TDR\_Score (Risk/Pain) on the Y-axis and the Business\_Value (provided by a product owner) on the X-axis. This 2x2 grid allows developers and product owners to have an objective, data-informed conversation and finally solves the "debt vs features" problem identified by researchers. Any debt in the "High TDR, High Business Value" box becomes the team's top priority.

### 5.2 Model Limitations

This model is a framework based on good-enough guesses ("heuristics"), and it's important to be honest about its limits.

1. **The Effort Guess:** The "Lizard Effort Proxy" (LEP) model (Section 3.1) is an unvalidated guess. While it's based on research, the formula itself, especially the k-factor, is just a proposal. Lizard outputs raw metrics like NLOC (lines of code) and CCN (Cyclomatic Complexity) and does **not** provide a "remediation effort" or "time to fix" estimate. The LEP formula has not been tested.
2. **The Bug Guess:** The "Heuristic-Based Fix-Mapping" (HBFM) model (Section 3.3) is completely dependent on developers writing good commit messages. It's easy to fool. If a developer fixes a bug and the commit message is "updated stuff", the script will miss it (a false negative). If they "fix a typo", the script might count it (a false positive).
3. **The Subjective Weights:** The weights (wc, wb) are powerful, but they are subjective. The framework doesn't specify what the "right" weights are. That's a strategic decision for the team.

### 5.3 Future Work

These limitations point to what should be done next.

1. **Validate the Effort formula:** A future study should measure the real effort (in hours or story points) from a development team for 10-20 refactoring tasks. The estimates could then be compared to the scores from our LEP formula. This would prove if the formula is accurate and help to tune the k-factor.
2. **Improve the Bug-Finding script:** The HBFM script (Section 3.3) is good, but it could be great. A better script would not just look for "Jira-like" keys. It would use the issue-tracker's API (ex: the Jira API) to confirm that the issue (ex: "AIRFLOW-123") is actually of type=Bug. This would eliminate almost all false positives. Same would be needed for GitHub issues. The current limitations for this method are:
   * Time complexity – need to check each issue's type which is way slower than using a regex.
   * API limits – Both GitHub and Jira have low limits for requests (GitHub 60/h without token, 5000/h with; Jira doesn't have it publicly, but too many requests result in temporary IP ban).
   * Just checking if Type=Bug could lead to false positives: an issue can be filed as a Bug but closed as "Won't Fix", "Works as Intended", or "Duplicate". So we'd need more complexity to solve this reliably.
3. **Run the real-world A/B test:** The best way to prove this framework is valuable is to run the experiment described in the original research proposal. A company would assign two teams to the same project. Team A (Control) would keep prioritizing debt the way they always have. Team B (Experimental) would use the TDR-W framework. After 6-12 months, the teams would be compared. If Team B has a shorter lead time and fewer bugs, it would prove the TDR-W framework is a powerful, predictive, and practical tool. This test requires extensive resources and specific checks to avoid bias.

## Citations

* Alfayez, R., Alwehaibi, W., Winn, R., Venson, E., & Boehm, B. (2020). A Systematic Literature Review of Technical Debt Prioritization. In International Conference on Technical Debt (TechDebt '20) (pp. 1-10). ACM.
* Besker, T., Martini, A., & Bosch, J. (2018). Technical debt cripples software developer productivity: A longitudinal study on developers’ daily software development work. In Proceedings of the 2018 International Conference on Technical Debt (pp. 105-114)
* Besker, T., Ghanbari, H., Martini, A., & Bosch, J. (2020). The influence of Technical Debt on software developer morale. Journal of Systems and Software, 167, 110586.
* Higo, Y., Matsumoto, Y., Kusumoto, S., & Inoue, K. (2005). Refactoring Effect Estimation based on Complexity Metrics. In Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE'05).
* Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., & Fontana, F. A. (2021). A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools. Journal of Systems and Software, 171, 110827.
* MacCormack, A., & Sturtevant, D. J. (2016). Technical debt and system architecture: The impact of coupling on defect-related activity. Journal of Systems and Software, 120, 170-182.
* Morgenthaler, J. D., Gridnev, M., Sauciuc, R., & Bhansali, S. (2012, June). Searching for build debt: Experiences managing technical debt at Google. In Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD) (pp. 1-6). IEEE.
* Paudel, B., Gonzalez-Huerta, J., Zabardast, E., & Klotins, E. (2024). Towards Measuring the Impact of Technical Debt on Lead Time: An Industrial Case Study. arXiv preprint arXiv:2406.01578.
* Seaman, C., & Guo, Y. (2011). Measuring and monitoring technical debt. In 3rd international workshop on managing technical debt (MTD) (pp. 1-7).