Friction in Software Engineering; What is it and how can it be reduced?

Authors: Eoin O'Brien, Kieran Flynn, James Miles & Robin O'Shea

Abstract

While a relatively young term, friction or technical debt, has existed since the conception of software development. Development friction is akin to surface resistance in nature - which causes the object (project) in motion to slow down. Juxtaposed; heat is a product of naturalistic friction, while short-term products from software friction, such as quicker project delivery, can provide momentary success - the accumulation of technical debt may prove to cause severe issues further into the development cycle. In this article we will discuss the origins of the debt idea and the numerous kinds of debt that can be encountered. This study will also discuss how technical debt manifests within projects and the methodologies used to reduce and measure friction.

1. Introduction

The idea of Friction, or Technical Debt, in software development first came to prominence in the early nineties by a man named Ward Cunningham. He used the idea of repaying debt as an analogy for the code needed in the development of the WyCash portfolio management system (1). Much like the financial idea of debt, a small amount of technical debt is manageable with timely modifications to the code. However, if developers release their code prematurely and avoid refactoring any potential issues, they are building up the burden of problems that the software has. Thus being the equivalent of taking out a loan from a bank and not making quick repayments and allowing your interest rates to rise gradually.

Other contributors to the idea such as Shane McConnell(19) and Jim Highsmith(20) touch on how the build-up of technical debt throughout development can have an impact on businesses in the long term. Tackling sightings of short term friction may not have an imminent benefit to the bottom line of the firm and there will be costs following on from system modifications at any time in development. That said, by optimising the cost of changing the way the system is designed, you can then minimise the amount of debt built up and maximise the amount of profit the company can garner from the service. How these firms can measure these costs and then go about tackling them is what we will be discussing through this article.

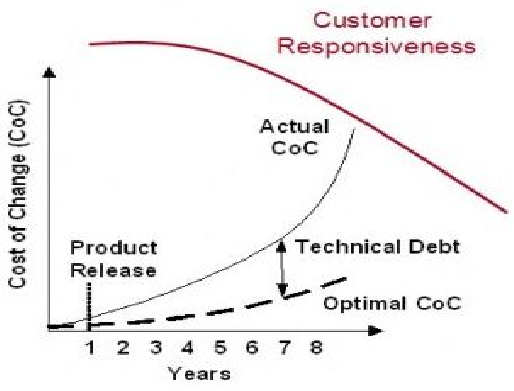


Fig 1. Cost of Change Line Graph - Jim Highsmith

2. Research Methodology

This paper was written in conjunction with the Multivocal Literature Review process proposed by Vahid Garousi (30).

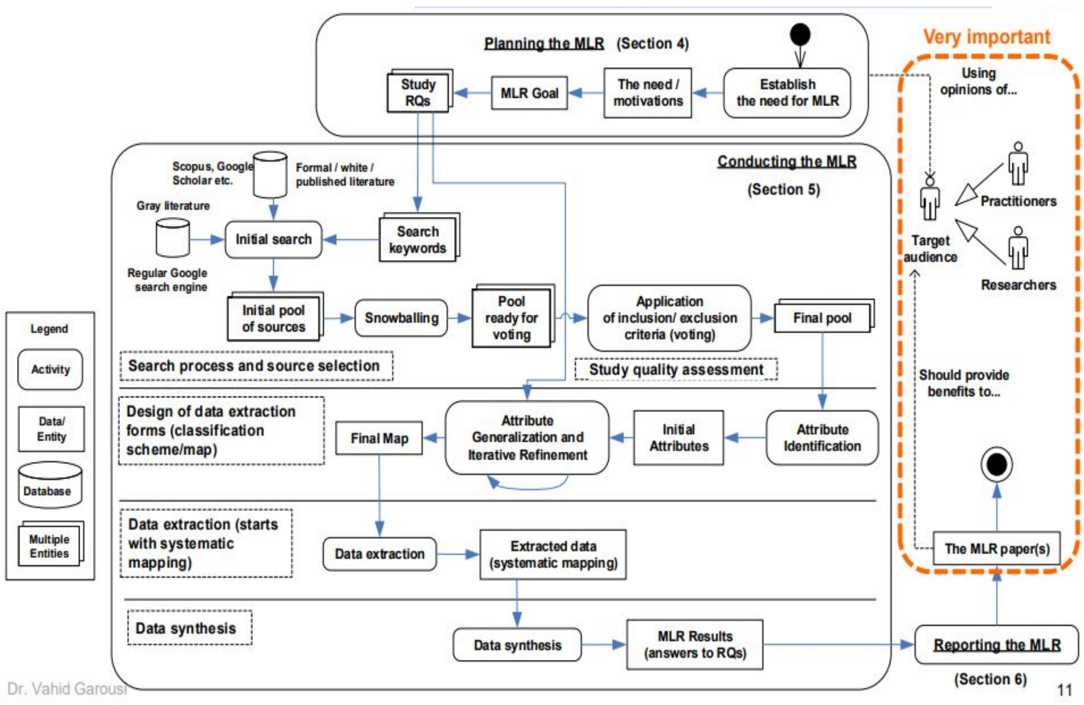

Planning the MLR (Section 4)
Study RQs
MLR Goal
The need / motivations
Establish the need for MLR
Very important
Using opinions of...
Practitioners
Target audience
Researchers
Should provide benefits to...
The MLR paper(s)
Conducting the MLR (Section 5)
Scopus, Google Scholar etc.
Formal / white / published literature
Gray literature
Initial search
Search keywords
Regular Google search engine
Initial pool of sources
Snowballing
Pool ready for voting
Application of inclusion/ exclusion criteria (voting)
Final pool
Search process and source selection
Study quality assessment
Legend
Activity
Data/ Entity
Database
Multiple Entities
Design of data extraction forms (classification scheme/map)
Final Map
Attribute Generalization and Iterative Refinement
Initial Attributes
Attribute Identification
Data extraction (starts with systematic mapping)
Data extraction
Extracted data (systematic mapping)
Data synthesis
Data synthesis
MLR Results (answers to RQs)
Reporting the MLR
(Section 6)
Dr. Vahid Garousi
11

Fig 2. MLR Process - Vahid Garousi

2.1 Research Questions

Initially a number of questions formed the basis of our search strings. These questions were used as a method of gaining a basic understanding of the problem space as well as discovering what the true aims of the paper would be. Examples of these questions can be seen below.

What is Friction?

What is the impact of Friction?

How can Friction be Reduced?

As more information was gathered on the topic of Friction, these questions were refined to be more appropriate in order to give the paper structure and a more comprehensive view of the subject.

The settled upon Research questions are as follows.
RQ1 - What is Friction in Software Development?
RQ2 - How do you Measure Friction?
RQ3 - What tools or methods are used to reduce Friction?

2.2 Search Strings

Below are some of the Search Strings used during the research process. These strings range from strings used in initial searches, intended to further knowledge in the topic area, to final strings used to attempt to answer the research questions developed previously.

Friction “Software Engineering” “Software Development”
Technical Debt “Software Engineering” “Software Development”
Impact Friction Technical Debt “Software Engineering”
Software development "cost" "friction" "technical debt"
Software development "cost"
Refactoring “Software Engineering”
Measure Friction "Software engineering"
Acceptable level friction “Software engineering”
Reduce Friction “Software Engineering” “Software Development”
Reduce Technical Debt “Software Engineering” “Software Development”
Mitigate “Friction” “Technical debt”

2.3 Databases Searched

Google Scholar, Google, IEEE Xplore Digital Library, Journal of Systems and Software, ScienceDirect, StackOverflow.

3. Discussion

3.1 RQ1 - What is Friction?

While Mr. Cunningham was the man responsible for the introduction of the Technical Debt idea, there are numerous individuals who have expanded upon his work and fleshed out the scope of the Friction concept. Martin Fowler uses the idea of crufts to visualise the same kind of shortcomings in software development that Cunningham deemed as Debt (15). The timeframe needed to expand upon an existing system can potentially increase in length if there is a cruft that intrudes on how the developer understands the software. This prompts a dilemma of whether to complete the expansion while ignoring the cruft in the structure, thus building up the technical debt, or resolve any problems with the cruft before proceeding any further. This will postpone the implementation of the new feature, however it will make any forthcoming expansions less complex in the future. Thus showing the need to tackle any type of these crufts in a timely manner.

3.1.1 Types of Friction

There are a number of different kinds of friction that can be encountered throughout the development cycle. These instances normally have little to no correspondence to one another however they can impact the debt build-up of other development factors and not just for the system overall. One instance of friction that can be rather costly is Architecture Debt. This occurs when there are difficulties that violate the requirements of how the system is meant to perform. Mistakes in the development of system architecture can be significant as it can inhibit the agility of how the system can be built upon in the future (13). This crisis will have a significant impact on the cost of change of the software since the amount of Architectural Debt built up over time will need to be matched by a similar amount of refactoring.

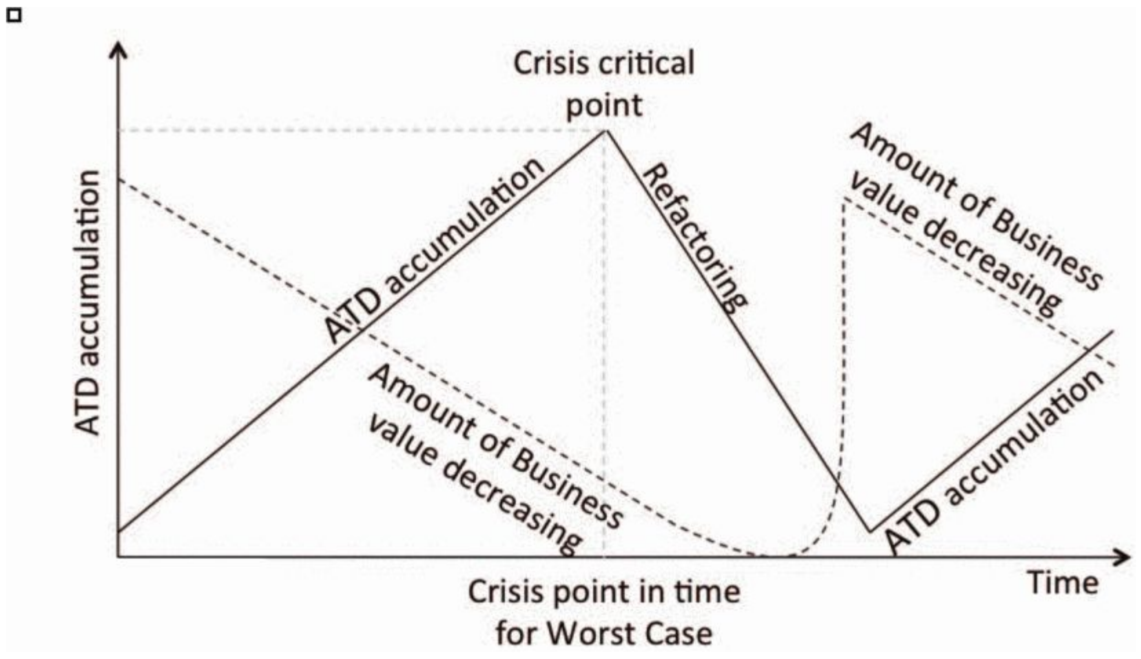


Fig 3. ATD Accumulation Graph - Gothenburg University 2014

Similarly to Architecture Debt, shortcomings in the development cycle can build up debt in phases such as Design and Testing (18). Design Debt is encountered when a decision made during development violates the principles of the design approach for the system, such as Object-Oriented design. Meanwhile Testing Debt refers to lapses in how the system is meant to be tested. This can refer to a lack of coverage for code fragments in the software or lack of planned testing altogether. Both can potentially lead to defects in your software and can have a major role in the cost of change.

Though it is not just decisions made when constructing the system that can impact the development teams' cost of change. There are kinds of debt that can gather interest from events outside of the control of the company. Documentation Debt can build when documentation in regards to the project has insufficient information to determine what is required in the system (18). There can also be complications found when analysing the trade-offs between requirements for the system, thus generating Requirement Debt. Issues affecting the staff and infrastructure of the company can also build up People Debt and Infrastructure Debt respectively (17). While it is important for a company to minimise the friction they encounter, they need to know what kind of friction it is beforehand and what was the cause of the friction in the first place.

3.1.2 Causes of Friction

"Understanding the reasons that development teams incur TD is important in order to expand the limits under which TD management activities can be performed. Instead of focusing only on strategies that deal with existing debt, knowledge about causes make possible the definition of strategies that consider the prevention of future occurrence" (16)

There are a number of factors that lead up to the build-up of these development overheads. One factor of particular note is difficulties encountered in planning the development of the system (3). Management of the development strategy can vary depending on the working environment inside of the company. That said, failure to properly prepare for the requirements of a system's formation can impact the rate of which the friction increases. This mismanagement can include setting unrealistic targets and deadlines for certain features, thus prompting errors in the code or how aspects of the system are intended to run. There also needs to be a consideration for how the team plans to allocate the work among each individual involved in the project. Stacking the burden of many key features among certain members of the team can lead to a suboptimal performance from these members and prevent them from meeting their targets.

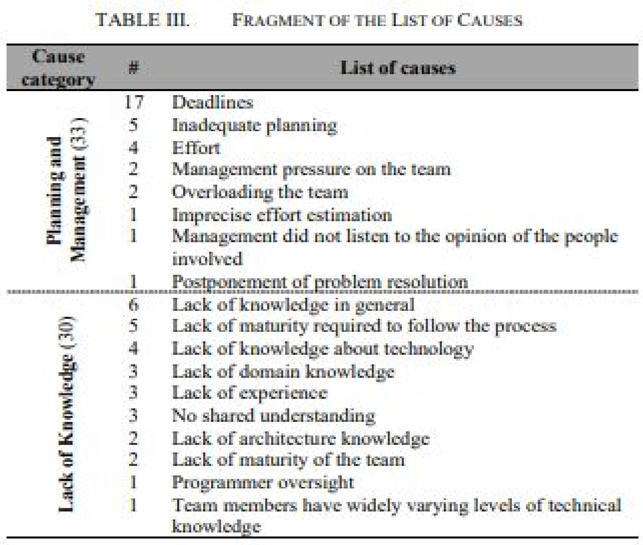


TABLE III. FRAGMENT OF THE LIST OF CAUSES

Cause category	#	List of causes
Planning and Management (33)	17	Deadlines
	5	Inadequate planning
	4	Effort
	2	Management pressure on the team
	2	Overloading the team
	1	Imprecise effort estimation
	1	Management did not listen to the opinion of the people involved
	1	Postponement of problem resolution
Lack of Knowledge (30)	6	Lack of knowledge in general
	5	Lack of maturity required to follow the process
	4	Lack of knowledge about technology
	3	Lack of domain knowledge
	3	Lack of experience
	3	No shared understanding
	2	Lack of architecture knowledge
	2	Lack of maturity of the team
	1	Programmer oversight
	1	Team members have widely varying levels of technical knowledge

Fig 4. List of TD Causes - Federal University of Bahia 2018

Another example of mistakes that can lead to an increase in the friction of a group development cycle is that of team understanding. Certain projects require knowledge and ability with the technologies required to optimise or expand upon the existing systems or to create a new system from scratch(3). Each member of the team needs to have the necessary ability to contribute code to the software that can perform the actions and features required by their clients and customers. If each individual has the capability to complete their necessary tasks, then the build-up of development costs is quite minimal. However, as the scope and technicality of the project expand, so too are the resources needed to complete it. Staff will need to be trained properly in the technical areas needed for their work to be finished in a timely, high-quality manner. That technical experience will be needed alongside experience in working on projects of a similar size. This can be hard when a large portion of up to professionals in the industry have less than ten years of experience in programming in any language or system, much less with advanced systems & programs used in industry circumstances (14).

Planning and experience are some examples of common, costly occurrences of issues that can cause an increase in friction for a business. Although they are not the only instances of problems that cause an increase in technical debt. Much like planning, testing is a key phase in the life cycle of a software's development. Admittedly, it does normally take place towards the end of the development cycle but it's still an integral part of creating a system(21). And much like with avoiding crufts in the design and structure of a software system, failure to test your software effectively can have an impact on your technical debt. It can nearly be as fatal to finances as structural problems since problems not found from testing can potentially appear in the final product. Thus applying more interest on top of the existing debt due to potential dissatisfaction from clients and customers.

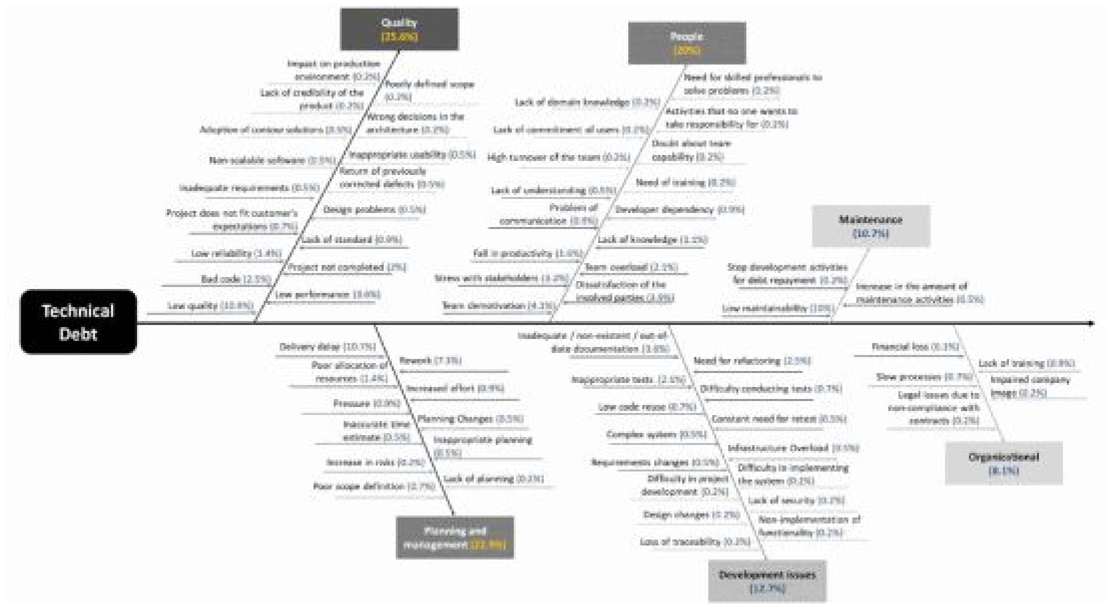

Technical Debt
Quality (25.6%)
Impact on production environment (0.2%)
Poorly defined scope (0.2%)
Lack of credibility of the product (0.2%)
Wrong decisions in the architecture (0.2%)
Adoption of contour solutions (0.5%)
Non-scalable software (0.5%)
Inappropriate usability (0.5%)
Return of previously corrected defects (0.5%)
Inadequate requirements (0.5%)
Design problems (0.5%)
Project does not fit customer's expectations (0.7%)
Lack of standard (0.9%)
Low reliability (1.4%)
Project not completed (2%)
Bad code (2.5%)
Low performance (3.6%)
Low quality (10.8%)
People (20%)
Need for skilled professionals to solve problems (0.2%)
Lack of domain knowledge (0.2%)
Activities that no one wants to take responsibility for (0.2%)
Lack of commitment of users (0.2%)
Doubt about team capability (0.2%)
High turnover of the team (0.2%)
Need of training (0.2%)
Lack of understanding (0.5%)
Problem of communication (0.9%)
Developer dependency (0.9%)
Lack of knowledge (1.1%)
Fall in productivity (1.6%)
Team overload (2.1%)
Stress with stakeholders (3.2%)
Dissatisfaction of the involved parties (3.5%)
Team demotivation (4.1%)
Maintenance (10.7%)
Stop development activities for debt repayment (0.2%)
Increase in the amount of maintenance activities (0.5%)
Low maintainability (10%)
Delivery delay (10.7%)
Rework (7.3%)
Poor allocation of resources (1.4%)
Increased effort (0.9%)
Pressure (0.9%)
Planning Changes (0.5%)
Inaccurate time estimate (0.5%)
Inappropriate planning (0.5%)
Increase in risks (0.2%)
Lack of planning (0.2%)
Poor scope definition (0.7%)
Planning and management (22.9%)
Inadequate / non-existent / out-of-date documentation (3.8%)
Need for refactoring (2.5%)
Inappropriate tests (2.1%)
Difficulty conducting tests (0.7%)
Low code reuse (0.7%)
Constant need for retest (0.5%)
Complex system (0.5%)
Infrastructure Overload (0.5%)
Requirements changes (0.5%)
Difficulty in implementing the system (0.2%)
Difficulty in project development (0.2%)
Lack of security (0.2%)
Design changes (0.2%)
Non-implementation of functionality (0.2%)
Loss of traceability (0.2%)
Development issues (12.7%)
Financial loss (0.1%)
Lack of training (0.8%)
Slow processes (0.7%)
Impaired company image (0.2%)
Legal issues due to non-compliance with contracts (0.2%)
Organizational (8.1%)

Fig 5. Plot of TD Cause categories - Federal University of Bahia 2019

The clients that the company works with are not immune to causing friction either. Much like the development team for the system, the partners looking for the system need to provide the developers with a clear idea of what actions they want the software to perform and how they want those actions to be performed. This can come through conscious documentation and clear communication in calls or meetings between both parties. As projects become wider in scale with parties having language and geographical barriers in the way(16). Failure to do so will not just have a significant impact on the developers' cost of change margins, but also impact the finances of the clients themselves. The amount of risk involved with this or any occurrences of friction to the company will need to be measured by them effectively.

3.2 RQ2 - How do you measure Technical Debt?

TD is recognised as a critical issue in the software development industry (3), and thus the ability to identify, monitor, measure and pay such debt is vital to the longevity of the project/company. Measurement enables the identification of potential underlying issues and ergo is a resource for decision-makers to examine the prioritization of TD items to pay off and implement strategies to reduce or prevent the effects of debt items. Yet due to TD's extensive list of causations, with (3) describing over 100 and (4) identifying 57, the measurement of debt is inherently difficult. A paper published by MIT explains how architectural health can serve as a proxy for the technical debt - a code base that has good architectural health is both modular and hierarchical in nature (4). TD describes the sacrifice of long-term quality for short-term value, while architectural debt is the sacrifice of modularity and hierarchy of the code in order to meet faster deliveries. In the following sections modularity and hierarchy will be discussed in detail and used to measure TD.

3.2.1 Modularity

"The concept of modularity is used primarily to reduce complexity by breaking a system into varying degrees of interdependence and independence across and hide the complexity of each part behind an abstraction and interface" (5)

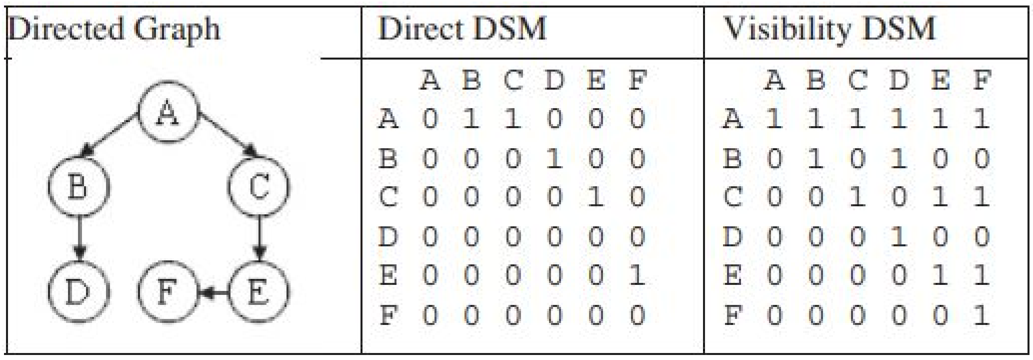
In other words, modules are units in a larger system that are structurally independent of one another but work together. Implementing loosely coupled architectures that are easy to modify or replace enables changes to an individual component without causing any changes to those that depend on it. Thus, this enables a developer to hide the complexity of each individual module, meaning other users only need to be concerned with the inputs and outputs of the module, essentially treating it as a black box. However, the splitting of modules carries the caveat of an integrative framework that allows for independence of structure while integrating its features. Considering the advantages to modularity, including the increase in range of manageable complexity and allowing different parts of the system to be developed concurrently, the time taken to provide a framework is minimal.

For modularisation to be effective, the designers are required to partition the design information into separate categories: visible and hidden information. Hidden information only affects their own piece of the system without triggering any external changes in the system. Modularity can be measured as units within a system can be grouped, based upon their levels of coupling, and used to analyse system structure and performance.

3.2.2 Hierarchy

"Design hierarchy refers to a specific ordering of components in a system, such that dependencies flow in a uniform direction. For example, if file A depends on file B, and file B depends on file C, there is a natural ordering of these components" (6)

If all components depend on C, then C consequently belongs at the top of the hierarchy - as any changes to C may impact those below it. Vice versa, A is placed at the bottom of the hierarchy as changes to it should not impact others. Hence there is a "building-block" approach to the development. The hierarchy of a software architecture is important to control unwanted coupling and propagation effects (6).



Direct DSM

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	0	1	0	0
C	0	0	0	0	1	0
D	0	0	0	0	0	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

Visibility DSM

	A	B	C	D	E	F
A	1	1	1	1	1	1
B	0	1	0	1	0	0
C	0	0	1	0	1	1
D	0	0	0	1	0	0
E	0	0	0	0	1	1
F	0	0	0	0	0	1

Fig 6. Design Structure Matrix

To measure the degree of coupling between components all direct and indirect dependencies are captured. Measures of component coupling are derived from a DSM (Design Structure Matrix). A DSM is a network modeling tool used to represent the elements comprising a system and their interactions and consequently aids decision making by showing potential system health vulnerabilities.

In the example above used by A. MacCormack in "Technical debt and system architecture", the A component is dependent upon components B and C. As C is dependent on E, a change to this component may have a direct impact on C, and an indirect impact on A. Thus the DSM displays all direct and indirect dependencies between components in a system. This is calculated via the transitive closure of the Direct DSM.

Visibility Fan-In (VFI) is obtained by summing down the columns for each component. Visibility Fan-Out (VFO) is obtained by summing along rows for each component. This allows for the calculation of the Propagation Cost, which can be used to compare the mean level of coupling across the system. This result measures the fraction of a system's components that could potentially be at risk.

3.2.3 Modularity & Hierarchy Health Assessment Tools

Above shows that both modularity and hierarchy are essential in the metrification of a system's technical debt. The four main providers of architectural health tools are SonarQube, Lattix, Silverthread Inc. and CAST (7). Each of these supply analysis methods for codebases and will be discussed in detail below.

SonarQube empowers developers with automated Static Code Analysis rules - catching bugs, fixing vulnerabilities and providing a linting standard to ensure code is written in a clean and safe manner. Code smells are used to represent a maintainability-related issue in the code (7), often an indication of a deeper problem. Code smells are not technically incorrect pieces of code, however, they point to flaws which may introduce bugs or cause further technical debt. SQ extends its reach to metricising complexity, duplications, issues, maintainability, reliability, security, size and tests (8) which provides developers with a visual representation of the technical debt within a system.

CAST Application Intelligence Platform categorise business' functions into measurable units. Focusing on the identification of deficiencies at the architectural level - distinguishing the relationships between different elements, layers and functions. It's engine is designed to measure software health, size, flaws and to generate architectural blueprints of multi-tiered, multi-technology software (9). The system-level analysis is used to analyse individual units of software, the interactions between said units and the overall health of the system. These health measures include; Robustness, Efficiency, Security, Changeability, Transferability. CAST AIP has the ability to identify critical problems such as difficult architectural flaws which may then lead to unpredictable stability, performance, security and data integrity issues.

The Lattix Platform offers a scalable, DevOps enabled way to manage software architecture across an entire application portfolio. Lattix is the only solution that can provide visibility into the software's design in one centralized view (10). Lattix Architect is a desktop application that enables the creation of dependency models. This allows customers to understand detailed dependencies of low-level elements, decompose and understand hierarchical relationships, control how 3rd party libraries are used and utilise metrics to measure complexity, stability, coupling and other measures.

SilverThread is a product based on 15 years of applied research at MIT and Harvard Business School (11). Their CodeMRI Portfolio measures hidden technical debt and quantify economic value across a system. The CodeMRI Diagnostics enables deep insight into the technical health of a specific system and exploration of both current and projected economic outcomes. SilverThread's CodeMRI Care improves technical health and lowers risks with developer tools that provide continuous real-time insight into the architectural health of a system. (12)

In each of the above, architectural health analysis aids to convert difficult relationships and structures in a codebase to visual representations. Lattix and Silverthread both provide DSM graphs.

3.2.4 Metrics

As seen above there is a broad range of metrics used to determine the TD within a system, however, there is a large overlap in metrics between the different providers; complexity, duplications, relationships between entities, maintainability, reliability, security, size, and tests.

3.3 What is the cost of Friction?

By underestimating the consequences of friction in software development and allocating too little time or too little resources to meet deliverables, it can affect the overall costs of running and cannot be ignored. If done so, it will degrade the software system making it difficult to manage. "*issues in reaching performance requirements of the software (due to the degraded internal quality of the software*" (22).

From a technical debt cost definition,*" it is the cost of fixing structural quality problems in production code that must be eliminated to control development costs and avoid operational problems"*(23). It is an optimisation problem with no approach for real-world applications due to the difficulty in finding an optimal solution, i.e. we cannot measure productivity. For this reason, it is an unknown quantification with traces of documentation. Decision management of technical debt is based on managers experience and software measurement data and may aid in controlling technical debt *"Schmid proposed a simplified approach that considers only evolution cost, refactoring cost, the probability that the predicted evolution path will be realized "*(24)

The figure below shows technical debt growing over a time period if not resolved. As technical debt increases, the interest will do so also. In the absence of technical debt maintenance costs remain optimal (25).

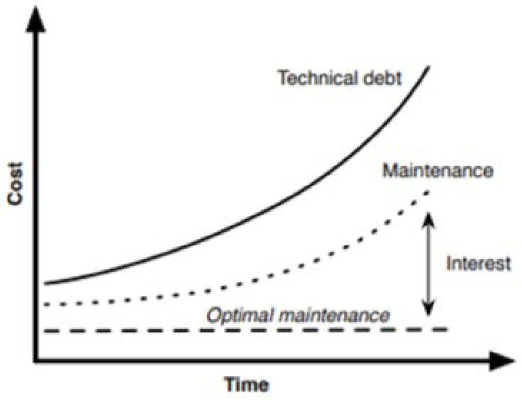


Fig 7. Cost of TD over time

3.3.1 Technical debt management costs

When dealing with technical debt, quality is less favourable to compensate for productivity. Maintenance time or cost is reduced in the short term which is the main advantage but is achieved at the cost of extra costs/work in the future

When orchestrating technical debt management, the decision to maintain or pay off debt is reliant on the *"present and future cost of the debt to maintain sustainability of a system and is measured by effort required to pay off the debt therefore an idea of the cost is essential for software maintenance"* (26).

Such core costs include the cost of meetings, effort spent managing the technical debt and time spent collecting data for technical debt management such as adjourning an educated guess as to the cost of fixing software quality issues, classified by the values of software metrics.

3.3.2 Effects and corresponding costs

The below Figure (50) displays the top 10 significant effects taken from questionnaires on technical debt within large organisations alongside their overall frequency of occurrence and level of significance

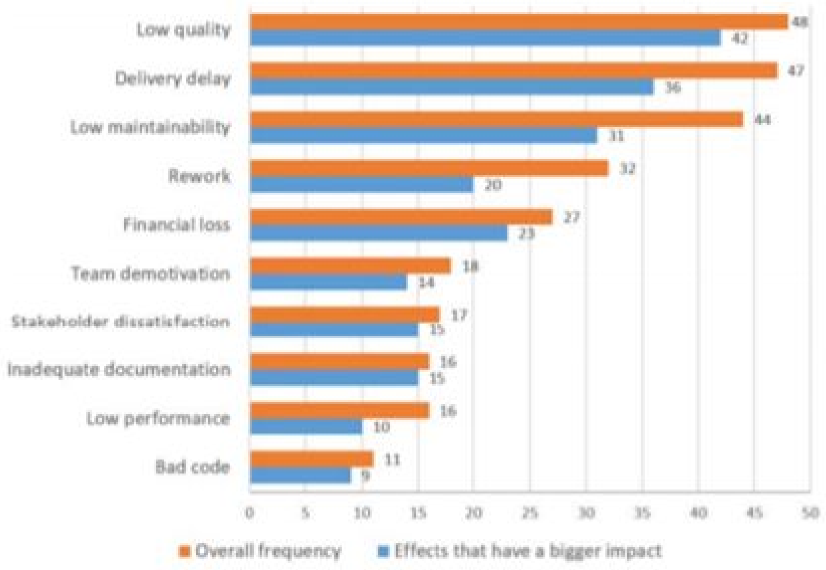


Fig 8. Effects of TD questionnaires vs. their frequency

Correction costs to problems in production code are roughly calculated from the *"number of hours required to remediate must-fix problems in production code, multiplied by the fully burdened hourly cost of those involved in designing, implementing, and testing these fixes"* (23). The following effects on members of the system can be a delay of patches, low quality, low maintainability, rework, financial loss, team demotivation, stakeholder dissatisfaction, inadequate documentation, low performance, bad code, updates and new versions or assets of the platform.

Code creates greater maintenance effort or computing resources that's costs represent interest on the debt. Interest being defined as *"the extra maintenance cost spent for not achieving the ideal quality level. Maintenance encompasses activities such as adding new functionality and fixing bugs"* (25). Maintenance is different from that of practical repairs or interest cost as it *" involves visible changes and their impacts are immediately visible (e.g., fixing bugs or adding new features)' (25).* interest of technical debt is not the same as maintenance costs and is somewhat indirectly proportional to each other. *"Systems without technical issues will still spend some effort on maintenance"* (25).

The substantial costs associated with technical debt and its importance is underlined by the SEI (Software Engineering Institute) *"For systems on which software architectural quality has been allowed to degrade, especially in its modifiability and maintainability dimensions, dealing with the stream of continual changes becomes increasingly less cost-effective, as more and more effort is required for comprehending and sustaining the system, leaving fewer resources for implementing new capabilities. This results in cost and schedule slippage or a diminished ability to field new capabilities"* (28). The problem of Technical Debt is its business risk. When these risks evolve into unfavorable operational situations it results in the creation of liabilities.

An example case study from a 2016 information house of representatives subcommittee report *"The federal government spends the majority of its $80 billion technology budget on maintaining and operating legacy systems." [20] While legacy programs are not inherently bad, there is significant technical debt that is inherited in those programs that must be dealt with just to keep the programs operating. "Money, time and manpower that are devoted [to these efforts]... are unavailable for other efforts, and this* *crippling debt can impact agency performance and jeopardize the success of IT modernization. The key to successful modernization is paying off technical debt by automating outdated workflows and processes..."* (29).

The below figure shows manpower costs associated with maintaining a software system (An analysis from a $1B firm with over 1000 developers). Code quality, represented on the y-axis, improves as you near the origin. The design quality is shown on the x-axis, with the complexity of code increasing moving further from the origin point (27).

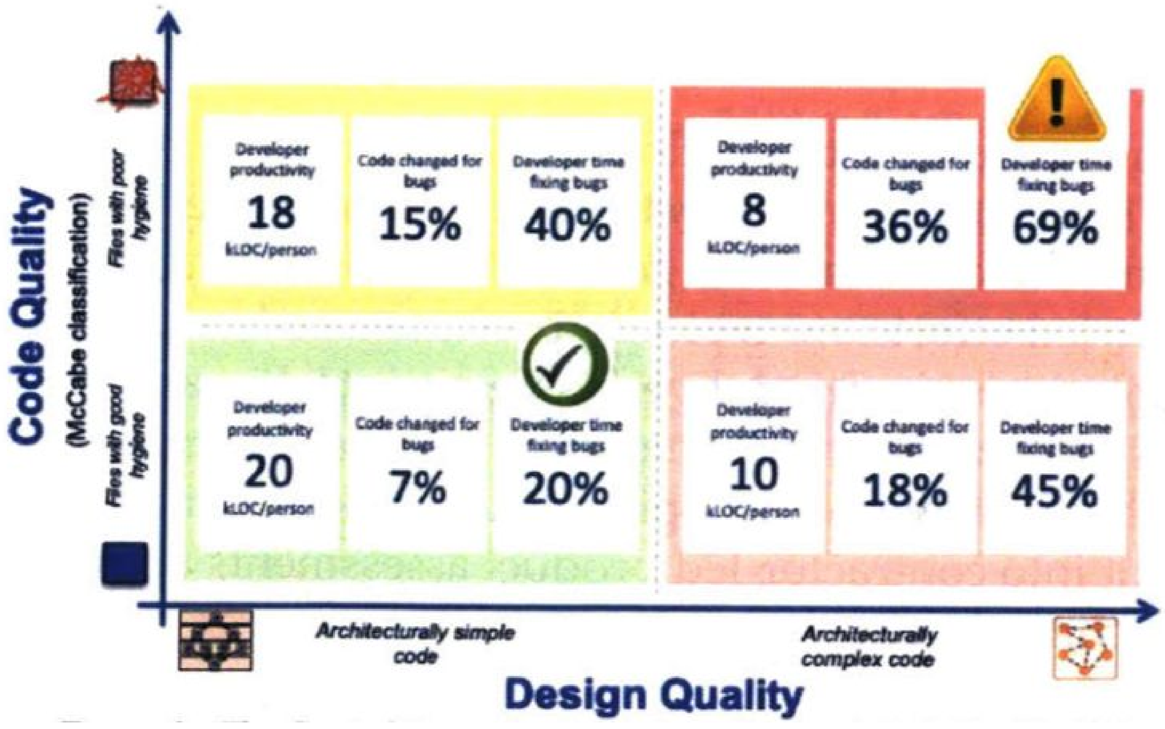


Fig 9. The cost of operating in Various States of Code Health

The diagram gives insight into the manpower required to debug, develop, and deliver new features in a system with good coding practices and a strong architectural structure being more cost-efficient.

3.4 RQ3 - What tools or methods can be used to reduce Friction?

3.4.1 Reduction of Technical Debt

When it comes to reducing the buildup of technical debt it seems that there are a number of strategies to be considered.

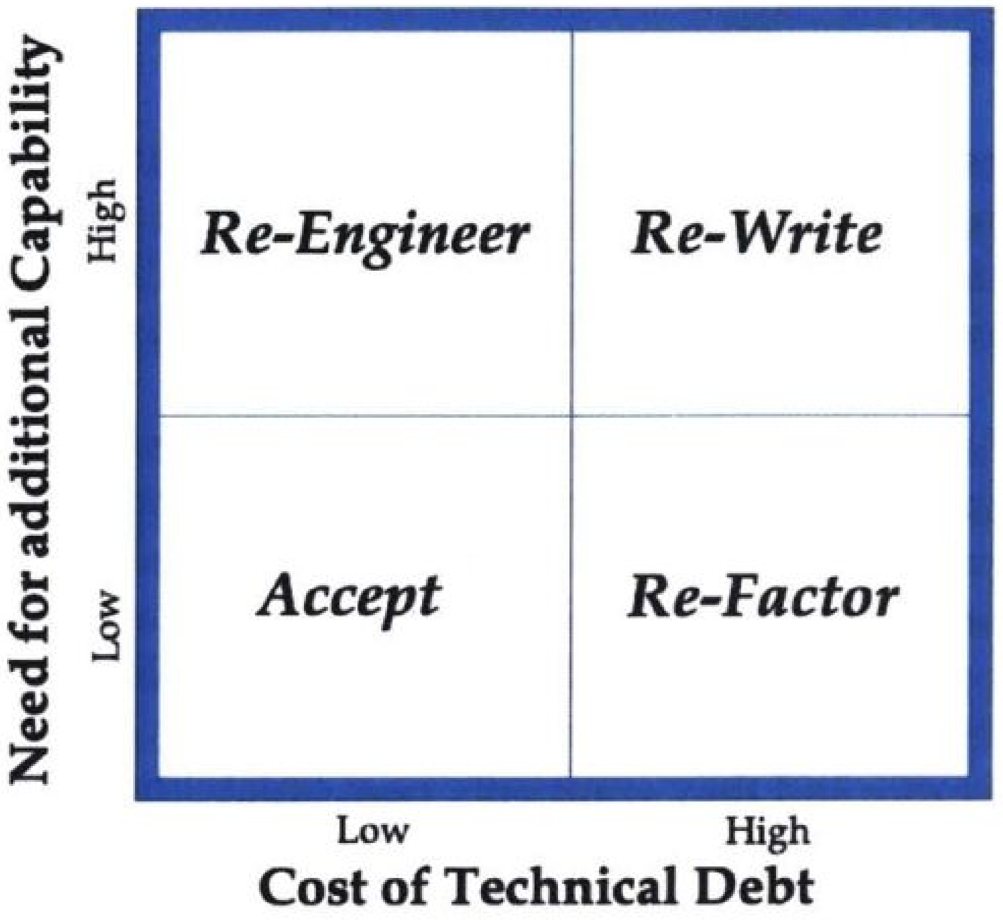


Fig 10. Cost vs. Capability Matrix for Resolving Technical Debt (33)

As can be seen above (33), the approach taken greatly depends on both the need for additional capability (Y-axis) & the cost of the technical debt item (X-axis).

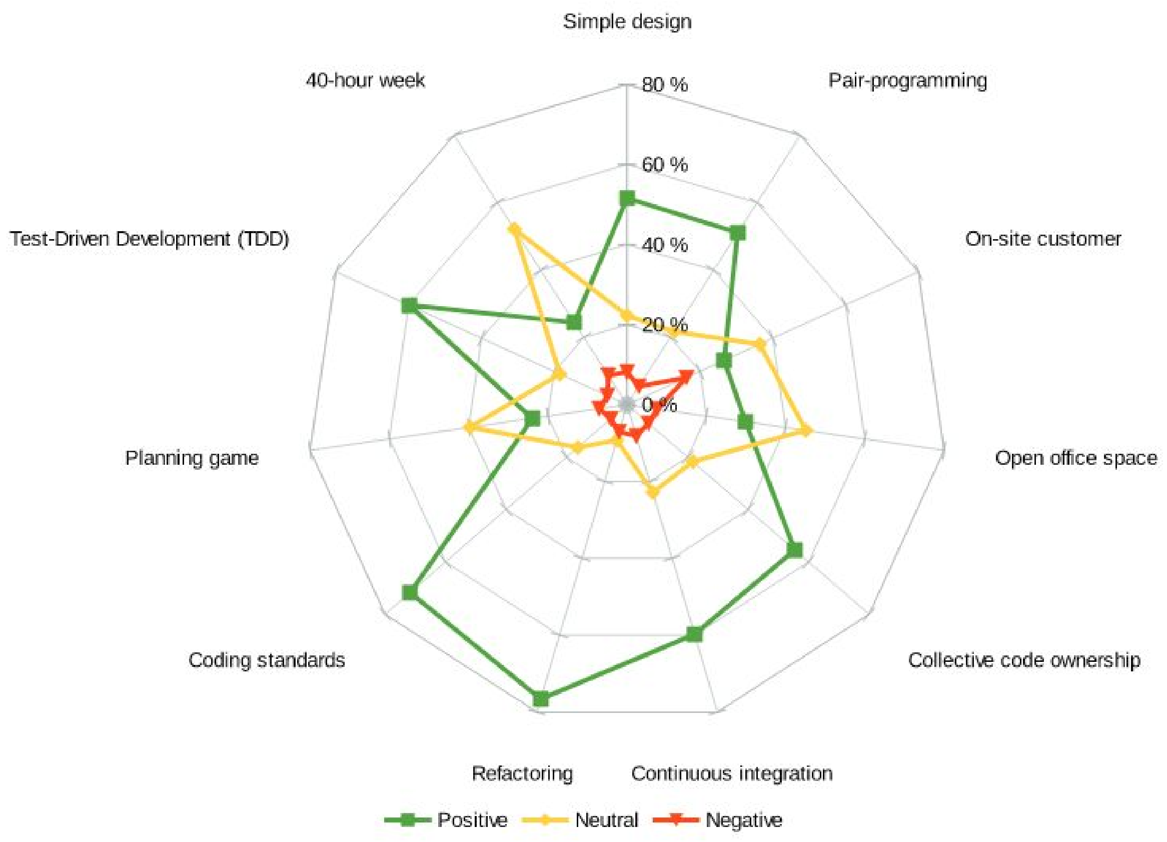


Fig 11. Agile practices & effect on TD (39)

The above diagram (39) is a representation of agile practices and the perceived effect on technical debt. It can be seen that many of the practices with the greatest effect on the amount of debt are preventative or ongoing measures rather than post-development work. This demonstrates the idea that prevention truly is better than cure when it comes to dealing with a difficult problem such as the buildup of technical debt.

Processes such as Coding Standards, Continuous Integration, Test-Driven Development and Collective Code Ownership (amongst others) appear to have the greatest impact on the resulting debt. This suite of preventative processes can be adopted by virtually any non-safety critical team to lower their cost of change in projects.

However with that being said it is still commonplace to need to reduce debt, as some level of debt is virtually inevitable. In general, a good high-level approach to reducing technical debt post-development consists of a number of tasks, the first of which being actually identifying the technical debt present *(32)*. This can be done through the use of static code analysers or by using stakeholder workshops to influence design decisions *(31)*.

The next step taken is to measure each piece of debt in terms of both benefit & cost. The benefit can be approximated subjectively by those undertaking the task of reducing the debt, while the cost can be measured in a number of ways including metrics mentioned previously. Measuring both the cost & benefit can stir discussion and can be a good reference point for gauging progress towards the goal of reducing the overall debt *(33).*

The debt items can then be prioritized in order of the highest payoff (i.e. highest value). The team can optimally allocate the limited resources they possess to the most pressing debt items. By doing this it can be ensured that the highest payoff debt items are repaid first, therefore providing the most value.

At this stage the debt is repaid through refactoring. This will be discussed in Section 3.4.2.

Lastly, any items that are not repaid in each iteration of the above process are monitored, as the cost or value of a piece of debt may increase or decrease over time. It may also be possible that certain technical debt items might escalate to being unmanageable. By iteratively reviewing the list of debt items it can be ensured that the team is optimally repaying debt at all times.

3.4.2 Refactoring

Refactoring is the process of rewriting code such that technical debt is reduced whilst maintaining the existing functionality of that code. If the functionality is changed, or new features are added, this is not refactoring and is referred to as Rework.

"Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior." (34)

Fowler *(35)* says in his book that *“Before you start refactoring, make sure you have a solid suite of tests. These tests must be self-checking.”*

The most important facet of refactoring is the concept that the code will continue to function correctly after the refactoring process has taken place. This is a vitally important aspect of refactoring that must not be overlooked. The tests that Fowler recommends provide piece-of-mind to the programmer(s) that the functionality of the code has not changed (as much as possible). Fowler describes it eloquently by saying *“By writing what I want twice, in the code and in the test, I have to make the mistake consistently in both places to fool the detector.”*

Fowler also proposes (35) that the best way to go about refactoring is to perform it in small steps so as to minimise the chance of introducing bugs or errors, thereby changing how the code executes.

“Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, each of which "too small to be worth doing". However the cumulative effect of each of these transformations is quite significant. By doing them in small steps you reduce the risk of introducing errors. You also avoid having the system broken while you are carrying out the restructuring - which allows you to gradually refactor a system over an extended period of time.”

By beginning to refactor code with the idea that the functionality will remain the same, developers can derive a number of goals or expectations from the process. They can expect all unit tests to pass both before and after the modification, they can expect to not have to change any tests or write any new ones, they can expect cleaner code when they are finished and they can expect no new behaviour (36).

In order to benefit fully from the refactoring process it’s crucial that there is a solid understanding of the specific process focus paired with a disciplined implementation. Otherwise it can do more harm than good in the long run to a system’s quality (33).

One massive side benefit from refactoring is the fact that developers will likely gain a better understanding of what constitutes good or clean code in the first place, thus somewhat reducing the buildup of technical debt before it even becomes a problem.

3.4.3 Clean Code

"Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control." (37)

"You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem." (Ward Cunningham)

The essence of clean code is that it is readable, predictable and reliable. It is a goal that all developers should strive towards, even if it is not always wholly attainable. Robert C. Martin (38) describes code as representing the details of the requirements. Martin also says that it is not enough for code to simply "work", i.e. all of the requirements are implemented or functional. The reason behind this is that simply working code, that is designed or implemented poorly, incurs more technical debt than idyllic clean code. As a result, by seeking to develop better code the first time around the resulting amount of technical debt will be reduced.

3.4.4 Zero Technical Debt

Although having absolutely zero technical debt in a company or a project may seem like an ideal to strive toward, it has been suggested that it may do more harm than good (40). One reason for this is the idea that the world has a finite set of resources that can be used at any one time. As a result of this, some tasks must be prioritised over others and trade-offs are made (41).

Therefore it is perfectly understandable to accept some technical debt. Even by knowing what debt is being accepted, it is a step in the right direction.

Going forward the accepted debt can continue to be monitored and supervised where necessary.

Conclusion

In conclusion, Friction or Technical Debt will likely be a concern for the majority of companies and project teams going forward. Insiders such as Ward Cunningham and Martin Fowler use the idea to convey the importance of satisfactory development practices. As we move towards a more and more Agile development environment more focus will need to be applied to “preventative maintenance”. Being proactive with processes such as Test-Driven Development, Coding Standards and Code Reviews can be of great benefit in lowering Technical Debt build-up. Also important is a focus on clean code from the outset, with due diligence paid towards the principles of software design and architecture as well as refactoring sprints in Agile environments which can have a huge impact on future endeavors. Whilst some technical debt may be acceptable, it will take prudent management to keep the cost of change at an optimal level.

Acknowledgements

[1] The WyCash Portfolio Management System, W. Cunningham, March 26, 1992. http://c2.com/doc/oopsla92.html

[2] E. Murphy-Hill, C. Parnin and A. P. Black, "How We Refactor, and How We Know It," in IEEE Transactions on Software Engineering, vol. 38, no. 1, pp. 5-18, Jan.-Feb. 2012. https://ieeexplore.ieee.org/abstract/document/6112738

[3] N. Rios, M. Gomes de Mendonca Neto, R. O. Spinola “A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners”, October 2018.

[4] N. Rios, R. O. Spinola, M. Gomes de Mendonca Neto, “A study of Factors that Lead Development Teams to Incur Technical Debt in Software Projects”, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8498243&tag=1

[5] C. Y. Baldwin, W. L. White, “Design Rules: The power of modularity”, 2000, https://books.google.ie/books?id=oaBOuo4mId8C&pg=PA63&redir_esc=y#v=onepage&q&f=false

[6] A. MacCormack, D. J. Sturtevant, “Technical debt and system architecture: The impact of coupling on defect-related activity”, 2016

[7] SonarQube, “Concepts | SonarQube Docs”, 2019 https://docs.sonarqube.org/latest/user-guide/concepts/

[8] SonarQube, “Metric Definitions | SonarQube Docs”, 2019, https://docs.sonarqube.org/latest/user-guide/metric-definitions/

[9] Cast Software, “Application Intelligence Platform”, 2020, https://www.castsoftware.com/products/application-intelligence-platform

[10]

[11] Silver Thread Inc, https://www.silverthreadinc.com/, 2020

[12] SilverThread Inc, https://www.silverthreadinc.com/codemri-platform, 2020

[13] A. Martini, J. Bosch and M. Chaudron, "Architecture Technical Debt: Understanding Causes and a Qualitative Model," 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, Verona, 2014, pp. 85-92. https://ieeexplore.ieee.org/abstract/document/6928795

[14] Stack Exchange Inc, Developer Survey Results 2019, https://insights.stackoverflow.com/survey/2019, 2019

[15] M. Fowler “Technical Debt”, https://www.martinfowler.com/bliki/TechnicalDebt.html, 2019

[16] N. Rios, R. Oliveira Spínola, M. Mendonça and C. Seaman, "Supporting Analysis of Technical Debt Causes and Effects with Cross-Company Probabilistic Cause-Effect Diagrams," 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), Montreal, QC, Canada, 2019, pp. 3-12.

https://ieeexplore.ieee.org/document/8785063

[17] Z. Codabux and B. Williams, "Managing technical debt: An industrial case study," 2013 4th International Workshop on Managing Technical Debt (MTD), San Francisco, CA, 2013, pp. 8-15. https://ieeexplore.ieee.org/abstract/document/6608672

[18] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes and R. O. Spínola, "Towards an Ontology of Terms on Technical Debt," 2014 Sixth International Workshop on Managing Technical Debt, Victoria, BC, 2014, pp. 1-7.

https://ieeexplore.ieee.org/abstract/document/6974882

[19] S. Watkins, S. McConnell, D. West, “Steve McConnell: How to Categorize & Communicate Technical Debt”, 2012.

https://www.castsoftware.com/blog/steve-mcconnell-on-categorizing-managing-technical-debt

[20] F. Stephan, “What Is Technical Debt”, 2016

https://www.excella.com/insights/what-is-technical-debt

[21] R.K. Bhujang, V. Suma, Risk Impact Analysis across the Phases of Software Development,2014
https://s3.amazonaws.com/academia.edu.documents/57016699/Risk_Impact_Anlysis.pdf?response-content-disposition=inline%3B%20filename%3DRisk_Impact_Analysis_Across_the_Phases_o.pdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWOWYYGZ2Y53UL3A%2F20200318%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20200318T124215Z&X-Amz-Expires=3600&X-Amz-SignedHeaders=host&X-Amz-Signature=0b338e3ba48cb60f8fb53c146b789ad187deddad4ec1f42fa06c109f8f27cf18

[22] Managing technical debt: An industrial case study

https://ieeexplore.ieee.org/abstract/document/6608672

[23] Estimating the size, cost, and types of Technical Debt

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6226000

[24] Exploring the costs of technical debt management – a case study

https://link.springer.com/article/10.1007/s10664-014-9351-7

[25] An empirical model of technical debt and interest Share on

https://dl.acm.org/doi/abs/10.1145/1985362.1985364

[26] https://link.springer.com/content/pdf/10.1007/s10664-014-9351-7.pdf

[27] D. Sturtevant, "Computer-Implemented Methods and Systems for Measuring, Estimating, and Managing Economic Outcomes and Technical Debt in Software Systems and Projects". United States Patent US 2017/0235569A1, 17 August 2017.

[28] F. Schull and I. Ozkaya, “Recommended Practice for Application of Quantitative Software Architecture Analysis in Sustainment”, Carnegie Mellon University: Software Engineering Institute, 2018

[50] The Most Common Causes and Effects of Technical Debt: First Results from a Global Family of Industrial Surveys

https://dl.acm.org/doi/abs/10.1145/3239235.3268917?casa_token=12PERIqG7WQAAAAA:BsN5lEs51JXcHeenNrEEmGYUIovXTHXji99Kz94sK-ahFSsRmbq0ZHgrSXhhrKP5KQLPT1DhI6rH

[29] Red Hat, "Paying Off Technical Debt for Successful IT Modernization," Federal News Network, 18 December 2018. [Online]. Available: https://federalnewsnetwork.com/open-first/2018/12/payingoff-technical-debt-for-successful-it-modernization/. [Accessed 28 December 2018]

[30] V. Garousi, M. Felderer, Mika V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering" in Information and Software Technology, vol. 106, pp. 101-121, 2019 https://www.sciencedirect.com/science/article/abs/pii/S0950584918301939

[31] P. Avgeriou, P. Kruchten, R. L. Nord, I. Ozkaya and C. Seaman, "Reducing Friction in Software Development," in *IEEE Software*, vol. 33, no. 1, pp. 66-73, Jan.-Feb. 2016. https://ieeexplore.ieee.org/abstract/document/7367977

[32] P. Kruchten, R. L. Nord and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," in IEEE Software, vol. 29, no. 6, pp. 18-21, Nov.-Dec. 2012. https://ieeexplore.ieee.org/document/6336722

[33] A. M. Page, "Technical debt : the cost of doing nothing". 2019. [ONLINE] Available at: https://dspace.mit.edu/handle/1721.1/121795. [Accessed 10 March 2020].

[34] Refactoring.com [ONLINE] Available at: *https://refactoring.com/* [Accessed 9 March 2020

[35] M. Fowler "Refactoring: Improving the design of existing code", 2018
https://martinfowler.com/books/refactoring.html

[36] Stack Overflow Post [ONLINE] Available at:
https://stackoverflow.com/questions/1025844/what-is-refactoring-and-what-is-only-modifying-code [Accessed 9 March 2020]

[37] Grady booch, "Object-Oriented Analysis & Design with Applications", Third Edition, 2007, Available at:
https://www.amazon.co.uk/Object-Oriented-Applications-Addison-Wesley-Technology-Hardcover/dp/020189551X

[38] Robert C. Martin, "Clean code: A Handbook of Agile Software Craftsmanship", 2008 Available At:
https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship-ebook/dp/B001GSTOAM

[39] Holvitie, J, Sherlock, L, Spínola, RO, Hyrynsalmi, S, MacDonell, SG, Mendes, TS, Buchan, J & Leppänen, V 2018, 'Technical Debt and Agile Software Development Practices

and Processes: An Industry Practitioner Survey', *Information and Software Technology*, vol. 96, pp. 141-160. In: Information and Software Technology, Vol. 96, 04.2018, p. 141-160. https://doi.org/10.1016/j.infsof.2017.11.015

[40] An exploration of technical debt - ScienceDirect. 2020. An exploration of technical debt - ScienceDirect. [ONLINE] Available at: https://www.sciencedirect.com/science/article/pii/S0164121213000022. [Accessed 11 March 2020].

[41] F. Shull, "Perfectionists in a World of Finite Resources," in *IEEE Software*, vol. 28, no. 2, pp. 4-6, March-April 2011.