Quantifying the Cost of Technical Debt

Carlo Bottiglieri

September 24, 2017

1 TL/NR

To make sure a cleanup effort produces the maximum positive impact on the code base, in the past I used a heuristic that centers on frequently changed code that also suffers from Technical Debt and poor automated test coverage. The idea is to minimize Technical Interest, which I define as the effort lost due to Technical-Debt-generated resistance to change. In order to quantify Technical Interest accrued for a set of changes over a period, without measuring actual development time against a benchmark, I propose the following rough formula :

$$TechnicalInterest = \sum_{changes} ChangeComplexity + ChangeUncoveredCode$$

2 Which Technical Debt?

In this article I mention Technical Debt frequently, but it's far from a well-defined term, moreover, the common understanding of this concept has diverged from what its creator, Ward Cunningham, originally meant[1].

Cunningham defined Technical Debt as the misalignment between a team's current best understanding of the problem domain and what is instead expressed in the code: imagine a cab fleet management system written in perfectly clean code, its design expressively describing the concepts of drivers, vehicles and car positions. Let's then imagine that at some moment the system's developers discover that by introducing the concept of 'areas of availability' they would be able to both enrich and simplify their complicated cab-selection features. If they decide they can't afford to do the necessary refactoring to introduce this new concept right now and instead keep this insight in their mind, but not in their code, for a while, they are accruing Cunningham's definition of Technical Debt.

The common understanding of Technical Debt is instead related to code which is just poorly implemented: obscure and convoluted logic that does not express the underlying problem domain at all; poor responsibility distribution; absent, inconsistent or not isolated components and more hallmarks of poor

[1] https://www.youtube.com/watch?v=pqeJFYwnkjE

technique. By this definition the cab fleet management system would suffer Technical Debt when the driver module owns and manages the current car's mileage, the car position is a plain string containing gps coordinates that get passed around in every other entity of the system and slow calls to remote resources are all synchronous with the UI.

If the first definition is concerned about expertly painted portraits not catching facets of a complex personality, the second is about accidental brush strokes and misplaced facial features.

This article is concerned with the Technical Debt by the common definition: the thing that is not just preventing software to excel on the long term, but that is able to calcify its evolution to the point where, after just a few man-months of work, dozens of man-days are needed to add a new drop-down box, while refactorings are as traumatizing as full rewrites, and even less likely to succeed.

3 Debt and Interest

For years now people have been talking of Technical Debt: developers wail about it; seniors prove their salt attacking it with sweeping refactorings whenever the stakeholders are looking the other way, often loosing themselves and their credibility in the crusade; many a failed project's corpse has been imputed to a manager letting this pest breed in it 'until after the release'. Regardless on how frequently it's blamed, calcifying low quality pervades the industry.

I believe that one of the reasons for stakeholder complacency in generating Technical Debt (a complacency that starts with obviously low-quality-producing staffing practices) and developers' ineffectiveness in repaying it, is the fact that, while Technical Debt is quantified, its effects are not.

We know (or at least believe we know) how much we would need to work to 'fix it all', but we have no clue how much we are being slowed down by not fixing it right now. This also means that, provided a large landscape of debt in a codebase, we don't know where reducing the Technical Debt will produce the greatest benefit for future efforts. Which part of our indebted code is generating the highest interest, the highest amount of attrition to our limited development resources?

4 Paying Interest

Unlike monetary debt, whose interest rate is expressed over time, Technical Debt does not generate interest linearly, nor continuously, with time. The most hideous working-mess-of-code will not generate extra effort if it never needs to evolve. Much like a game where the situation stays still until one of the players makes a move, in software development nothing happens unless you have to act on the code.

When you do have to act though, depending on the depth of debt in the area you are working on, you'll pay more or less interest. This will happen in the

form of time spent understanding complicated code, manually testing untested code and bug-fixing regressions.

So, when are we paying Technical Interest? When we change code. For every modified statement there's a price to pay.

How much do we pay? We could measure it empirically by developing a feature in the system as it is, then refactoring the system until it gets to near-zero Technical Debt and re-developing exactly the same feature. The difference in effort is the Technical Interest. The problem with this approach is obvious: from a commercial point of view it is an exercise in futility. In the absence of such empirical data I propose a formula that I believe might approximate truth:

- The effort to 'understand the code' is linear to the Cognitive Complexity of the function containing the changed statement : $U * CC$
- U can be considered constant in most cases and can be set to 1 until the amount of effort actually spent in developing the change is available, at which point it can be set to $U = \frac{0.5 * Effort_{change}}{CC}$.
- The effort to 'bug-fix' the change and induced regressions is linear to the lack of automated test Branch Coverage of the function containing the changed statement $T * (1 - BC)$
- T can be considered constant in most cases and can be set to 1 until the amount of effort actually spent in developing the change is available, at which point it can be set to $T = \frac{0.5 * Effort_{change}}{(1-BC)}$.

Finally, the overall interest paid for a given period is the sum of all bits of interest paid for each software change applied in the period:

$$Interest = \sum_{c=change_0}^{change_n} (U * CC_c + T * (1 - BC_c))$$

4.1 When U and T cannot be assumed constant

If system module boundaries are fuzzy (signatures are not there, or just wrong...) or entirely absent, a developer who needs to understand the code that he needs to change will not be able to stop at function calls, but actually move around reading implementations of things that are being called by the target code. Since efferent coupling gives an indication of how many things you depend on from your code, in the case of weak boundaries it also gives an indication of how many things the developer will have to understand beyond the immediate scope he has to act upon. This is why I suggest a second approximation that defines the U factor as a function of the Efferent Coupling from the function we are changing : $U = k_u * f(C_e)$

Similarly, if test isolation[2] is very poor the obvious symptom is that many tests break at every change and the more tests break, the harder to find the

[2] http://wiki.c2.com/?UnitTestIsolation

source of the break. Presuming[3] homogeneously poor test isolation, the more a change is depended upon, the more the test failures. I thus suggest a second approximation that defines the T factor as a function of the Afferent Dependencies to the function we are changing : $T = k_t * g(C_a)$

Combining everything together this gives the following second approximation formula :

$$Interest = \sum_{c=change_0}^{change_n} (k_u * f(C_e) * CC_c + k_t * g(C_a) * (1 - BC_c))$$

5 Final Thoughts

1. In this article I've used Cognitive Complexity[4] as a way to evaluate the effort to understand code; while it is not perfect, I prefer it over Cyclomatic Complexity[5]. I believe that, by dropping the relation to logical branches, Cognitive Complexity better models how a human brain is impacted by code structures; but since Cognitive Complexity has only recently been defined and it is supported only by some code quality tools, all my direct experiences on the topic of Technical Interest are based on Cyclomatic Complexity.

2. To reiterate, the gist of the second section: what I'm proposing relates to targeting a cleanup effort, raw code quality improvement, not a conceptual refactoring (again, referring to Cunningham's definition of Debt). Refactoring from one conceptual model to another one should not be conditioned by change-frequency considerations, but rather by how salient the new model will be for the future of the system being developed.

3. While I've used variants of the formula proposed above on code-bases I was intimate with, in the absence of a dedicated tool it's impractical to go through the history of a project to find the hotspot. As a result, I often find myself (and others) taking a shortcut to find the next cleanup target: pick the most highly complex code. The rationale for this is that, considering that the high complexity induces high change cost that such a large amount of logic is very likely to attract future changes, high complexity functions are a safe bet. I've recently developed a set of scripts to find out exactly this: are high-complexity functions a good target for refactoring if we don't have the luxury of a full historical analysis of Technical Interest? I hope to report the results soon.

[3] We might avoid making this assumption by using a metric that I call 'test distance' and which I've not yet documented

[4] https://www.sonarsource.com/docs/CognitiveComplexity.pdf

[5] http://en.wikipedia.org/wiki/Cyclomatic_complexity