Introduction to the Technical Debt Concept

by Jean-Louis Letouzey and Declan Whelan
Contributors: Thierry Coq, Jean-Pierre Fayolle, Tom Grant and Dan Sturtevant
Special thanks to Ward Cunningham for its review

What is Technical Debt? Where does it comes from?

Ward Cunningham, one of the authors of the Agile Manifesto, once said that some problems with code are like financial debt. It's OK to borrow against the future, as long as you pay it off.

Since Ward first used this metaphor, which he called "Technical Debt", it has gained momentum. While people still disagree about the exact definition of technical debt, the core concept identifies a serious problem that many software teams are struggling to manage.

Ward used it the first time when he was developing a financial application in Smalltalk. He wanted to justify to his boss the refactoring they were doing, so he used a financial analogy:

> *"If we failed to make our program align with what we then understood to be the proper way to think about our financial objects, then we were going to continue to stumble on that disagreement which is like paying interest on a loan."*

Later, in 1992, at the OOPSLA conference, Ward provided additional details (slightly paraphrased here based on feedback from Ward):

> *"Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with* refactoring[1]*. The danger occurs when the debt is not repaid. Every minute spent on code that is not quite right for the programming task of the moment*[2] *counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unfactored*[3] *implementation, object-oriented or otherwise."*

Ward elaborated further in a highly viewed video:

> *"With borrowed money you can do something sooner than you might otherwise, but until you pay back that money you will pay interest.*
>
> *"I thought borrowing money was a good idea. I thought that rushing software out the door to get some experience with it was a good idea. But that of course you would*

[1] Ward originally used the term 'a rewrite' as the term 'refactoring' was not in common use.
[2] Ward originally used the term 'not-quite-right' to express code that was no longer quite right for the programming task at the moment.
[3] Ward originally used the term 'unconsolidated' as the term 'refactoring' was not in common use.

> *eventually go back and as you learned things about that software you would repay that loan by refactoring the program to reflect your experience as you acquired it."*

Ward also provided additional clarification during a later interview**:**

> *"We can say that the code is of high quality when productivity remains high in the presence of change in team and goals.*
>
> *"I would say that debt is localized within a team, its goals and the code offered by a team to meet those goals. However, any mechanism that attempts to measure team commitments, goals alignment and code quality across organizational boundaries would be valuable in our increasingly dependent world."*

In reviewing this paper Ward added:

> *There is an implicit assumption when I speak of repaying debt that the owners of the software want to save the value invested in the software for ongoing development. This is not always the case. It might make business sense to pile debt into software if a liquidity event is on the horizon. This is where business strategy meets engineering strategy. A startup, for example, is building both a product and a company. Can the investors be criticized for maximizing their returns? Or is this just another version of bait and switch?*

We could summarize the metaphor as follow:

> *When taking short cuts and delivering code that is not quite right for the programming task of the moment, a development team incurs Technical Debt. This debt decreases productivity. This loss of productivity is the interest of the Technical Debt.*

Why do we use this metaphor?

The Technical Debt concept is an effective way to communicate about the need for refactoring and improvement tasks related to the source code and its architecture.

If you are able to estimate roughly the time needed for fixing what is not right into your code, the principal of your debt, you could compare this information to other project data, like remaining days before release date. This estimation will help you to understand your situation and plan repayment activities.

What incurs Technical Debt?

Code that is not quite right may include many types of issues. These issues may be related to architecture, structure, duplication, test coverage, comments and documentation, potential

bugs, complexity, code smells, coding practices and style. All these types of issues incur technical debt because they have a negative impact on productivity.

Technical Debt may emerge during the life of a project. At time progresses you may understand something new about your application domain. You may now view your initial architecture as having acquired technical debt.

Are they other types of debt?

Not all software projects issues are Technical Debt:

- Identified defects are not Technical Debt. They are *Quality Debt*.
- Lack of process or poor process is not Technical Debt. It is *Process Debt*. An example is Configuration Management Debt.
- Wrong or delayed features are not Technical Debt. They are *Feature Debt*.
- Inconsistent or poor user experience is not Technical Debt. It is *User Experience Debt*.
- Lack of skills is not Technical Debt. It is *Skill Debt*.

Is Technical Debt bad?

Taking short cuts in order to put earlier a viable product on the market which delivers business value is generally not a bad decision. But one should be conscious that the Technical Debt incurred will hurt sooner or later.

At some time, the team should pay back at least a part of the accumulated Technical Debt. They are different ways to do that, and there is no magic one that fits all situations. In order to fully understand the situation and establish the relevant strategy, the Technical Debt should be made fully transparent and analyzed.

How to analyze TD?

All the Technical Debt items are not the same. In order to understand the situation, in order to select and prioritize refactoring or improvement activities, it is important to analyze them. The following table provide a list of aspects that shall be taken into account.

Technical Debt item attribute	**Detail, Rational**
Type of impact	Which activities and, abilities will it hurt?
Amount of impact	How much will it hurt? Is the potential negative impact on the business limited or ....very high?
Duration and periodicity of the impact	Will the impact happen once, or will it recur?

Timing of the impact	Will the impact be perceived very soon (i.e. when I will need to achieve a good unit test coverage), or later, after delivery?
The age of the Technical Debt item	Is it “legacy TD,” or did the team add it during the last sprint?
Is it intentional or not?	Involuntary TD may trigger coaching and training for team members.
Refactoring cost	This will help to plan and prioritize repayment activities.
Dependencies with other Technical Debt items	Is the Technical Debt item included within the impact scope of another debt item?

This list represents the good sense that teams should apply, and is largely self explanatory. However, it is worth saying a few more words about the last point.

To prioritize repayment tasks, it is important to identify and assess dependencies between scopes of changes. As an example, it will be mindless to fix two issues located in two instances of a duplicated block. It is more efficient to first fix a copy and paste issue and then to fix the remaining issue within the remaining block.

Other external aspects should be considered for selecting and prioritizing refactoring activities, including:

- The business value of the component.
- History, age and future decommissioning of the component.
- The probability of getting a negative impact (it depends on the usage and the change frequency of the piece of code where the TD item is located).

So, what to do?

The Technical Debt concept is quite simple. When looking into details, it appears that they are many aspects to consider for analyzing and managing the situation.

We have consolidated a list of recommendations and best practices in the document: “Project Management and Technical Debt”