Technical debt

Technical debt (also known as **design debt**[1] or **code debt**, but can be also related to other technical endeavors) is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer.[2]

As with monetary debt,[3] if technical debt is not repaid, it can accumulate 'interest', making it harder to implement changes. Unaddressed technical debt increases software entropy. Technical debt is not necessarily a bad thing, and sometimes (e.g., as a proof-of-concept) is required to move projects forward. On the other hand, some experts claim that the "technical debt" metaphor tends to minimize the impact, which results in insufficient prioritization of the necessary work to correct it.[4][5]

As a change is started on a codebase, there is often the need to make other coordinated changes in other parts of the codebase or documentation. Changes required that are not completed are considered debt, and until paid, will incur interest on top of interest, making it cumbersome to build a project. Although the term is used in software development primarily, it can also be applied to other professions.

Contents

- Causes
- Types
- Service or repay the technical debt
- Consequences
- See also
- References
- External links

Causes

Common causes of technical debt include:

- Ongoing development, long series of project enhancements over time renders old solutions sub-optimal.
- Insufficient up-front definition, where requirements are still being defined during development, development starts before any design takes place. This is done to save time but often has to be reworked later.
- Business pressures, where the business considers getting something released sooner before the necessary changes are complete, builds up technical debt comprising those uncompleted changes.[6]:4[7]:22
- Lack of process or understanding, where businesses are blind to the concept of technical debt, and make decisions without considering the implications.
- Tightly-coupled components, where functions are not modular, the software is not flexible enough to adapt to changes in business needs.
- Lack of a test suite, which encourages quick and risky band-aid bug fixes.

- Lack of documentation, where code is created without supporting documentation. The work to create documentation represents debt.[6]
- Lack of collaboration, where knowledge isn't shared around the organization and business efficiency suffers, or junior developers are not properly mentored.
- Parallel development on multiple branches accrues technical debt because of the work required to merge the changes into a single source base. The more changes done in isolation, the more debt.
- Delayed refactoring – As the requirements for a project evolve, it may become clear that parts of the code have become inefficient or difficult to edit and must be refactored in order to support future requirements. The longer refactoring is delayed, and the more code is added, the bigger the debt.[7]:29
- Lack of alignment to standards, where industry standard features, frameworks, technologies are ignored. Eventually integration with standards will come, and doing so sooner will cost less (similar to 'delayed refactoring').[6]:7
- Lack of knowledge, when the developer doesn't know how to write elegant code.[7]
- Lack of ownership, when outsourced software efforts result in in-house engineering being required to refactor or rewrite outsourced code.
- Poor technological leadership, where poorly thought out commands are handed down the chain of command.
- Last minute specification changes, these have potential to percolate throughout a project but no time or budget to see them through with documentation and checks.

Types

In a discussion blog "Technical Debt Quadrant",[8] Martin Fowler distinguishes four debt types based on two dichotomous categories: the first category is reckless vs. prudent, the second, deliberate vs. inadvertent.

Technical debt quadrants

	Reckless	**Prudent**
Deliberate	"We don't have time for design"	"We must ship now and deal with consequences (later)"
Inadvertent	"What's Layering?"	"Now we know how we should have done it"

Service or repay the technical debt

Kenny Rubin uses the following status categories:[9]

- Happened-upon technical debt—debt that the development team was unaware existed until it was exposed during the normal course of performing work on the product. For example, the team is adding a new feature to the product and in doing so it realizes that a work-around had been built into the code years before by someone who has long since departed.
- Known technical debt—debt that is known to the development team and has been made visible using one of the previously discussed approaches.
- Targeted technical debt—debt that is known and has been targeted for servicing by the development team.

Consequences

"Interest payments" are caused by both the necessary local maintenance and the absence of maintenance by other users of the project. Ongoing development in the upstream project can increase the cost of "paying off the debt" in the future. One pays off the debt by simply completing the uncompleted work.

The buildup of technical debt is a major cause for projects to miss deadlines. It is difficult to estimate exactly how much work is necessary to pay off the debt. For each change that is initiated, an uncertain amount of uncompleted work is committed to the project. The deadline is missed when the project realizes that there is more uncompleted work (debt) than there is time to complete it in. To have predictable release schedules, a development team should limit the amount of work in progress in order to keep the amount of uncompleted work (or debt) small at all times.

If enough work is completed on a project to not present a barrier to submission, then a project will be released which still carries a substantial amount of technical debt. If this software reaches production, then the risks of implementing any future refactors which might address the technical debt increase dramatically. Modifying production code carries the risk of outages, actual financial losses and possibly legal repercussions if contracts involve service-level agreements (SLA). For this reason we can view the carrying of technical debt to production almost as if it were an *increase in interest rate* and the only time this decreases is when deployments are turned down and retired.

> "As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it."[10]
>
> — Meir Manny Lehman, 1980

While Manny Lehman's Law already indicated that evolving programs continually add to their complexity and deteriorating structure unless work is done to maintain them, Ward Cunningham first drew the comparison between technical complexity and debt in a 1992 experience report:

> "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."[11]
>
> — Ward Cunningham, 1992

In his 2004 text, *Refactoring to Patterns*, Joshua Kerievsky presents a comparable argument concerning the costs associated with architectural negligence, which he describes as "design debt".[12]

Activities that might be postponed include documentation, writing tests, attending to TODO comments and tackling compiler and static code analysis warnings. Other instances of technical debt include knowledge that isn't shared around the organization and code that is too confusing to be modified easily.

Writing about PHP development in 2014, Junade Ali said:

> The cost of never paying down this technical debt is clear; eventually the cost to deliver functionality will become so slow that it is easy for a well-designed competitive software product to overtake the badly-designed software in terms of features. In my experience, badly designed

> software can also lead to a more stressed engineering workforce, in turn leading higher staff churn (which in turn affects costs and productivity when delivering features). Additionally, due to the complexity in a given codebase, the ability to accurately estimate work will also disappear. In cases where development agencies charge on a feature-to-feature basis, the profit margin for delivering code will eventually deteriorate.
>
> — Junade Ali writes in *Mastering PHP Design Patterns*[13]

Grady Booch compares how evolving cities is similar to evolving software-intensive systems and how lack of refactoring can lead to technical debt.

> "The concept of technical debt is central to understanding the forces that weigh upon systems, for it often explains where, how, and why a system is stressed. In cities, repairs on infrastructure are often delayed and incremental changes are made rather than bold ones. So it is again in software-intensive systems. Users suffer the consequences of capricious complexity, delayed improvements, and insufficient incremental change; the developers who evolve such systems suffer the slings and arrows of never being able to write quality code because they are always trying to catch up."[1]
>
> — Grady Booch, 2014

In open source software, postponing sending local changes to the upstream project is a form of technical debt.

See also

- Code smell (symptoms of inferior code quality that can contribute to technical debt)
- Big ball of mud
- Spaghetti code
- Software rot
- Shotgun surgery
- Bus factor
- Escalation of commitment
- Software entropy
- SQALE
- Sunk cost
- TODO, FIXME, XXX
- Overengineering

References

1. Suryanarayana, Girish (November 2014). *Refactoring for Software Design Smells* (1st ed.). Morgan Kaufmann. p. 258. ISBN 978-0128013977.
2. "Definition of the term "Technical Debt" (plus, some background information and an "explanation")" (https://www.techopedia.com/definition/27913/technical-debt). *Techopedia*. Retrieved August 11, 2016.
3. Allman, Eric (May 2012). "Managing Technical Debt". *Communications of the ACM*. **55** (5): 50–55. doi:10.1145/2160718.2160733 (https://doi.org/10.1145%2F2160718.2160733).

4. Jeffries, Ron. "Technical Debt – Bad metaphor or worst metaphor?" (https://web.archive.org/web/20151111011323/http://ronjeffries.com/articles/015-11/tech-debt/). Archived from the original (http://ronjeffries.com/articles/015-11/tech-debt/) on November 11, 2015. Retrieved November 10, 2015.
5. Knesek, Doug. "Averting a 'Technical Debt' Crisis" (https://www.linkedin.com/pulse/averting-technical-debt-crisis-part-1-doug-knesek). Retrieved April 7, 2016.
6. Girish Suryanarayana; Ganesh Samarthyam; Tushar Sharma (11 November 2014). *Refactoring for Software Design Smells: Managing Technical Debt* (https://books.google.com/books?id=1SaOAwAAQBAJ&pg=PA3). Elsevier Science. p. 3. ISBN 978-0-12-801646-6.
7. Chris Sterling (10 December 2010). *Managing Software Debt: Building for Inevitable Change (Adobe Reader)* (https://books.google.com/books?id=LYQlOaRwpnEC&pg=PA17). Addison-Wesley Professional. p. 17. ISBN 978-0-321-70055-1.
8. Fowler, Martin. "Technical Debt Quadrant" (http://martinfowler.com/bliki/TechnicalDebtQuadrant.html). Retrieved 20 November 2014.
9. Rubin, Kenneth (2013), *Essential Scrum. A Practical Guide to the Most Popular Agile Process*, Addison-Wesley, p. 155, ISBN 978-0-13-704329-3
10. Lehman, MM (1996). "Laws of Software Evolution Revisited" (http://dl.acm.org/citation.cfm?id=681473). *EWSPT '96 Proceedings of the 5th European Workshop on Software Process Technology*: 108–124. Retrieved 19 November 2014.
11. Ward Cunningham (1992-03-26). "The WyCash Portfolio Management System" (http://c2.com/doc/oopsla92.html). Retrieved 2008-09-26.
12. Kerievsky, Joshua (2004). *Refactoring to Patterns*. ISBN 978-0-321-21335-8.
13. Ali, Junade (September 2016). *Mastering PHP Design Patterns | PACKT Books* (https://www.packtpub.com/application-development/mastering-php-design-patterns) (1 ed.). Birmingham, England, UK: Packt Publishing Limited. p. 11. ISBN 978-1-78588-713-0. Retrieved 11 December 2017.

External links

- *Ward Explains Debt Metaphor* (http://c2.com/cgi/wiki?WardExplainsDebtMetaphor), video from Ward Cunningham
- OnTechnicalDebt (http://www.ontechnicaldebt.com) The online community for discussing technical debt
- Experts interviews on Technical Debt: Ward Cunningham (https://web.archive.org/web/20120621171342/http://blog.techdebt.org/resources-links/67/ward-cunningham-interview-about-technical-debt-sqale-agile), Philippe KRUCHTEN (https://web.archive.org/web/20120717083235/http://blog.techdebt.org/interviews/156/interview-with-philippe-kruchten-on-technical-debt-rup-ubc-decision-process-architecture), Ipek OZKAYA (https://web.archive.org/web/20121129073247/http://blog.techdebt.org/interviews/189/technical-debt-interview-with-ipek-ozkaya-on-technical-debt-sei-ieee-software-architecture-agile), Jean-Louis LETOUZEY (https://web.archive.org/web/20120714053317/http://blog.techdebt.org/interviews/118/interview-with-jean-louis-letouzey-on-technical-debt-and-sqale)
- Steve McConnell discusses technical debt (http://www.construx.com/10x_Software_Development/Technical_Debt/)
- TechnicalDebt (http://www.martinfowler.com/bliki/TechnicalDebt.html) from Martin Fowler Bliki
- Averting a "Technical Debt" Crisis (https://www.linkedin.com/pulse/averting-technical-debt-crisis-part-1-doug-knesek) by Doug Knesek
- An Andy Lester talk entitled (http://www.media-landscape.com/yapc/2006-06-26.AndyLester/) "Get out of Technical Debt Now!"
- Lehman's Law (http://www.inf.ed.ac.uk/teaching/courses/rtse/Lectures/lehmanslaws.pdf)

- Managing Technical Debt Webinar by Steve McConnell (https://www.youtube.com/watch?v=lEKvzEyNtbk)
- Boundy, David, Software cancer: the seven early warning signs (http://dl.acm.org/citation.cfm?id=156632) or here (https://www.academia.edu/2303865/Software_cancer_the_seven_early_warning_signs), ACM SIGSOFT Software Engineering Notes, Vol. 18 No. 2 (April 1993), Association for Computing Machinery, New York, New York, US
- Technical debt: investeer en voorkom faillissement (http://www.emerce.nl/opinie/hoge-ontwikkelkosten-eigen-schuld) by Colin Spoel
- Technical debts: Everything you need to know (https://gauravtiwari.org/2016/12/10/technical-debts-basics/)
- What is technical debt? (https://deepsource.io/blog/what-is-technical-debt/) from DeepSource blog

---