Editor: George Fairbanks
gf@georgefairbanks.com

Garbage Collect Your Technical Debt

Michael Keeling, Timothy J. Halloran, and George Fairbanks

THERE IS A kind of design distortion that happens when a team chooses to build iteratively instead of looking at all of the requirements at once. Ward Cunningham coined the term *technical debt* to describe those design distortions.[1] By understanding the causes of tech debt and connecting them back to a team's actions (or inactions), it's possible to minimize the buildup of tech debt and keep a system healthy indefinitely. The way to minimize tech debt is to view a software development process as an algorithm, consider several algorithms, and choose the right one for the circumstances.

However, most developers don't think about their process as an algorithm, so let's ease into the idea by looking at garbage collection algorithms. Watching tech debt build up on a project is a bit like watching a program allocate memory. Consider this description of garbage collection:

> *Running a program creates garbage, which is memory that's been allocated but is unused. Garbage creation is unavoidable, so we must occasionally pause to collect garbage. It's nicer when those pauses are predictable and short. There are various garbage collection algorithms that have different properties.*

And here's a description of tech debt, using the same phrasing:

> *Running a timeboxed iteration creates tech debt, which is working code with an obsolete design. Creating tech debt is unavoidable, so we must occasionally pause to refactor the code. It's nicer when those pauses are predictable and short. There are various iterative software development processes that have different properties.*

Consider this: a team's software development process is an algorithm, run by the team itself, that generates and cleans up a kind of garbage that we call tech debt. We know how to analyze algorithms, so let's analyze a team's process just like any other algorithm.

Software development processes control tech debt using two techniques. The first technique is cleaning up existing tech debt. Most teams already do this by refactoring. The second is avoiding the creation of tech debt. This is less common but more interesting. Let's look at each in turn.

Technique: Tech Debt Cleanup

You can control tech debt by cleaning it up after it exists. Often, a team "bolts on" a feature without regard to the existing design, identifies tech debt, and only then refactors to clean it up. Sometimes the cleanup happens immediately, but it could be much later.

Small problems can be refactored in minutes, but bigger problems can take days, weeks, or months to clean up. When developers take a break from writing features to fix tech debt, that's like a garbage collector pausing to clean up garbage. Spending time on refactoring means less time for new features. The bigger the problem, the longer it takes to refactor.

Because it requires stealing time from feature building, teams can find themselves under pressure to do less refactoring, especially large refactorings. As a result, they clean up the small problems but delay cleaning up big problems, such as the system's architecture.[2] Postponing a small cleanup can transform it into a big cleanup because, over time, code builds up around the problem, and it too must be refactored.

Digital Object Identifier 10.1109/MS.2021.3086578
Date of current version: 20 August 2021

Technique: Tech Debt Avoidance

You can control tech debt by creating less of it—that is, by avoiding it. Teams do that by considering design alternatives and choosing the one that creates the least tech debt. When asked to add a new feature, a team considers how well the current design can accommodate that feature. If the design is already suitable, they add the feature. But if the design is unsuitable, they update the design first, then add the feature.

Kent Beck summarized it this way: "[F]or each desired change, make the change easy (warning: this may be hard), then make the easy change."[3] The wordplay in Beck's quote is delightful, but the idea here is not a linguistic trick. Figure 1 shows two possible software development processes to control tech debt. The first allows tech debt to happen, then cleans it up. The second looks for upcoming trouble and avoids it by redesigning before implementing the feature.

To many developers, avoiding problems sounds better than cleaning them up. Be aware, however, that some tech debt is unavoidable. Sooner or later, a new requirement will be an unpleasant surprise. You might wonder if peeking ahead at future requirements would work, but that's not foolproof because the fog of design obscures our view of the future.[4]

Some teams worry that tech debt avoidance is a waterfall process in disguise, or worse, big design up front. That's clearly not the case, as a waterfall process would have the team look at all the requirements and deliver one system to handle them. Tech debt avoidance means that the team works on the requirements iteratively, delivering a working system with each iteration.

Some teams worry that tech debt avoidance will lead to analysis paralysis. Today, we see lots of teams struggling to control their tech debt, but we don't know of any teams using an iterative process that are stuck in analysis paralysis. Perhaps that's because there are strong forces pushing the team to deliver features with each iteration.

Choosing a Tech Debt Algorithm

We've explored two techniques to keep tech debt low: cleanup and avoidance. Expanding the combinations of those two techniques yields four kinds of tech debt algorithms to choose from: none, reactive (cleanup only), proactive (avoidance only), and balanced (both cleanup and avoidance). These algorithms are summarized in Figure 2.

We've seen teams succeed with all of these algorithms. We've also seen teams choose an unsuitable algorithm and suffer, then conclude that tech debt is an untamable monster. Choosing the right algorithm for your team depends on circumstances, including the team and project size, domain knowledge, design experience, technology experience, and schedule pressure.

Let tech debt happen, then clean it up	Anticipate tech debt and avoid it
1) Get new requirement/feature.	1) Get new requirement/feature.
2) Write the test case.	2) Revise the design, if necessary. (Is the architecture OK? Is the domain model OK?)
3) Edit code minimally so the test passes.	3) Write the test case.
4) Later on, refactor to remove code duplication.	4) Revise code to match the design.

FIGURE 1. Tech debt cleanup and avoidance.

None

Some teams don't do anything to control tech debt, and the parallel with garbage collection holds up: there are no-operation garbage collectors. If you write a quick script for yourself, and you don't plan to reuse it, why worry about tech debt? The same thinking applies to bigger projects, such as commercial computer games, where developers know they will start a fresh codebase for the next game. The developers suffer with tech debt only until the game is released.

Reactive

The reactive algorithm, using only tech debt cleanup, is what most teams do today. Teams can focus primarily on the stream of features to build, pausing occasionally to clean up tech debt "garbage." Bigger cleanup efforts are hard, so early mistakes linger because they are too expensive to refactor later. It's easier to recognize problems than it is to avoid them, so reactive makes sense when the developers have limited design skills.

Proactive

The proactive algorithm, using only tech debt avoidance, is uncommon today. If you can avoid tech debt with a bit of thinking, that's more efficient than blundering into obvious problems. On the other hand, if you don't have experience with the technology being used, you may waste time based on bad assumptions. Despite efforts to avoid tech debt, it will happen, so teams that start with the proactive algorithm may switch to the balanced algorithm to clean it up.

Balanced

Most teams wish their tech debt were lower, so they should use the balanced algorithm because it includes both cleanup and avoidance. Depending on the circumstances, they can do more or less of each technique.

Here's an example of a balanced algorithm that we find pragmatic. At the start of each iteration, the team discusses how the feature requests will affect the current design. That keeps the iteration design-focused and the design fresh in everyone's minds.

They may peek ahead at future feature requests, even though they aren't working on them now, because knowing what's coming may help them answer today's design questions. Sometimes a feature is hard to add to the design. It could contradict an assumption about the domain, or it could be hard to build within the current architecture.

If the developers can rework the design and add the feature within the current iteration, that's great. When they cannot, they chat with the product owner. They weigh political, economic, and social forces as well as schedule pressure and engineering risk before deciding. The answer might be to bolt the feature on and clean up the tech debt later, postpone the feature entirely, or something in between.

Finite and Infinite Games

Perhaps the most important factor in deciding which tech debt algorithm suits your team is whether your team is playing a finite or infinite game. Finite games can be lost or won. Infinite games can be lost, but winning just means you can keep playing. Tech debt feels a bit like an infinite game: If you can keep it under control, you can keep playing. Otherwise, you lose and declare tech debt bankruptcy.

Teams with a strict schedule are playing a finite game. One of the authors (Halloran) developed a military wargame simulation, StratWar, that had to be completed so students could use it in the next semester. He met the deadline but built up vast amounts of tech debt.[5]

Start-up companies are playing a series of finite games. They operate in do-or-die mode to reach the next milestone, and failure means the company dies. Halloran also worked at a static code analysis start-up company that scrambled to build a product to show at the JavaOne conference. As you would expect, the demo built up a lot of tech debt, but showing up at the trade show with working software let the company live another day and kept hope alive to switch to playing an infinite game.

Inexperience can force you to play a finite game. If developers don't know the problem domain or the implementation technologies, they are in a finite game until they can build something that works. Prototyping can build experience faster than up-front design or refactoring, but tech debt will make that code unsuitable for the long term.

Switching from a finite to an infinite game runs the risk of tech debt bankruptcy. Sometimes you discard the code from the finite game, as we did in the StratWar example. Other times you nurse the code back to health, as we did in the analysis start-up.

If you declare bankruptcy and decide to rewrite the system, it is critical to

		Clean Up Tech Debt	
		No	Yes
Avoid Tech Debt	No	**None: Ignore Tech Debt** Some code is never touched after it is delivered, so it makes sense to code right up to the deadline, ignoring tech debt.	**Reactive: Clean Up Existing Tech Debt** New features are added in a bolt-on fashion, without regard to the design. Afterward, if the design looks lousy, the team refactors to clean up the problem (very common).
	Yes	**Proactive: Avoid Creating Tech Debt** When starting on a new feature, team members consider how well the current design can accommodate it. If the design is already suitable, they add the feature. But if the design is unsuitable, they update the design before implementing the feature.	**Balanced: Clean Up and Avoid Tech Debt** The balance may change depending on the maturity of the system, with mature systems needing less avoidance because their design is already a good fit for the problem domain. This is the best way to minimize tech debt for most projects.

FIGURE 2. The kinds of tech debt algorithms. A program may clean up garbage once it exists, avoid creating garbage, both, or neither. Similarly, an iterative software development process may guide developers to remove existing tech debt, avoid creating it, both, or neither.

re-evaluate your tech debt algorithm. Don't keep using an algorithm tuned to a finite game and hope it's suitable for an infinite game. It's a good time to try the balanced algorithm, both avoiding tech debt through good design practices and cleaning up the inevitable debt through refactoring.

Minimize Your Tech Debt

Managing tech debt is a bit like managing memory allocation. By choosing your software development process, you can control how tech debt accumulates, just like a garbage collector reclaiming memory. It's helpful to think of your software development process as an algorithm that controls your system's tech debt.

Building software iteratively leads inevitably to tech debt because we choose to deliver systems before we have looked at all the requirements. Not knowing what's next distorts our designs, and that distortion is the tech debt. In theory, waterfall could avoid that distortion, but, in practice, it introduces other design distortions by peering far into a foggy future.[6]

Software processes have a dominant decomposition: either a stream of features or the system's design. Today, most teams focus on a stream of features, and it follows naturally that those teams rely primarily, or even exclusively, on tech debt cleanup.[7]

We have spoken with teams that work differently. In addition to refactoring, they also proactively avoid tech debt. They have flipped the dominant decomposition, making the system's design their primary concern. Their iterations are design-focused, not feature-focused.

Today, teams struggle with tech debt. Some managers believe it's uncontrollable and expect tech debt bankruptcy after a few years. The idea that our own software development process is contributing to tech debt is liberating because our process is under our control. By looking at tech debt as analogous to garbage creation, you change your perspective. Tech debt might be inevitable, but you can minimize it by choosing a suitable algorithm.

ABOUT THE AUTHORS

MICHAEL KEELING is a software engineer at LendingHome, San Francisco, California, 94104, USA, and the author of *Design It!: From Programmer to Software Architect*. Further information about him can be found at http://neverletdown.net/. Contact him at mkeeling@neverletdown.net.

TIMOTHY J. HALLORAN is a software engineer at Google, Pittsburgh, Pennsylvania, 15206, USA and a retired U.S. Air Force Lieutenant Colonel. Contact him at hallorant@gmail.com.

GEORGE FAIRBANKS is a software engineer at Google. Contact him at gf@georgefairbanks.com.

References

1. G. Fairbanks, "Ur-technical debt," *IEEE Softw.*, vol. 37, no. 4, pp. 95–98, July–Aug. 2020. doi: 10.1109/MS.2020.2986613.
2. M. Keeling, "Headwinds to redesign," *IEEE Softw.*, vol. 38, no. 2, pp. 128–132, Mar.–Apr. 2021. doi: 10.1109/MS.2020.3043081.
3. K. Beck, "for each desired change, make the change easy (warning: this may be hard), then make the easy change," Twitter, Sept 25, 2012. [Online]. Available: https://twitter.com/kentbeck/status/250733358307500032
4. T. Halloran, "The fog of software design," *IEEE Softw.*, vol. 38, no. 3, pp. 132–135, May–June 2021. doi: 10.1109/MS.2021.3056937.
5. T. Halloran, "Development of the StratWar Wargame Software," LeMay Center, Maxwell AFB, AL, Practicum Rep., DTIC ADA428794, Feb. 2004.
6. G. Fairbanks, "The rituals of iterations and tests," *IEEE Softw.*, vol. 37, no. 6, pp. 105–108, Nov.–Dec. 2020. doi: 10.1109/MS.2020.3017445.
7. G. Fairbanks, "Why is it getting harder to apply software architecture?" *IEEE Softw.*, vol. 38, no. 4, pp. 126–129, July–Aug. 2021.