Chapter 2. What Is Technical Debt?

Drawing from a financial metaphor, the concept of technical debt shifts the conversation about decision making from a technical standpoint or an economic standpoint to a place where developers and managers can better understand the trade-offs and compromises in software development and decide on the way forward. In this chapter, we describe the technical debt landscape through the forms technical debt takes in different types of development artifacts across the software development lifecycle. We explore more thoroughly the concept of a technical debt item and its causes and economic consequences as principal and interest. We introduce the technical debt timeline to help you understand how technical debt unfolds over time.

Mapping the Territory

In Chapter 1, "Friction in Software Development," we defined *technical debt* in software-intensive systems as the "design or implementation constructs that are expedient in the short term but that set up a technical context that can make a future change more costly or impossible." We added that "technical debt is a contingent liability whose impact is limited to internal system qualities—primarily, but not only, maintainability and evolvability."

Technical debt is mostly invisible when looking at or using a software product. It manifests in two main ways: difficulty and additional cost in evolving the system (that is, adding new functionality) or maintaining the system (that is, keeping the system running when the technical environment changes). But concretely, opening the box and looking at technical debt at the software level reveals that it takes many different forms.

In this chapter, we survey the software development landscape with respect to technical debt and then dig a bit deeper into the technical and economic implications of this definition.

The Technical Debt Landscape

Figure 2.1 illustrates a typical technical debt landscape showing the software development issues that developers work on to improve the system. We distinguish the visible issues, such as new feature requests and defects that need to be fixed, from the mostly invisible issues, which are visible only to software developers. Issues related to evolution appear primarily on the left side of the figure; issues related to maintenance and quality appear primarily on the right.

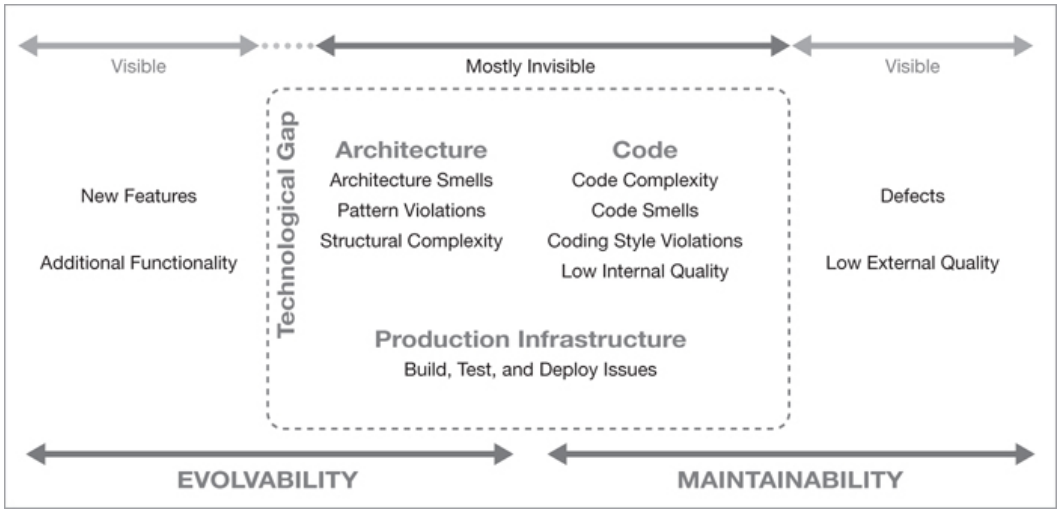


Figure 2.1 *The technical debt landscape*

Our focus is on the mostly invisible aspects of evolution and maintenance. Technical debt takes different forms in different types of development artifacts, such as the code, the architecture, and the production infrastructure. The different forms of technical debt affect the system in different ways.

The source code embodies many design and programming decisions. The code can be subjected to review, inspection, and analysis with static checkers to find issues of finer granularity: violations of coding standards, bad naming, code clones, unnecessary code complexity, and misleading or incorrect comments. Many of these symptoms of technical debt are referred to as *code smells*. When a system incurs technical debt at the source code level, the debt tends to hinder maintainability so that it will be hard to make corrections to the system when needed.

Other technical debt items are more encompassing and pervasive. They involve choices about the structure or the architecture of the system: choice of platform, middleware, technologies for communication, user interface, or data persistency. Static code checkers don't find technical debt caused by these types of choices. For these elements, the principal and the interest are often higher than for technical debt in the code. When a system incurs technical debt at the architectural level, the debt tends to hinder evolvability: The system will be hard to extend to new features, with their functional and quality attribute requirements, such as scaling to a larger number of users, processing different kinds of data, and the like.

Not all technical debt is associated with bad internal quality. Technical debt incurred by the passage of time and the evolution of the surrounding environment is not the result of bad quality. Your system could have had the best possible design (or code) at the time you built it; five years later, it is deep in technical debt because of changes in your environment—not because the system has degraded. A technological gap has grown between the original state and the current environment. For example, perhaps you picked AngularJS as your front-end web application framework, but starting with the most recent release, the release documentation announces that AngularJS will not continue to support Internet Explorer. You ignore this version incompatibility for a number of releases, focusing on implementing other functionality, until you discover that the number of customers using Internet Explorer was not as small as you initially thought. You incurred debt just because time passed and you didn't revisit your initial choice, not because you took a shortcut in that initial choice.

Finally, some technical debt items are associated not with the code of the product but with the code of other closely related artifacts in software production: build scripts, test suites, or deployment infrastructure.

The constant characteristic across this landscape of technical debt is its invisibility. Technical debt is not visible outside the system's development organization; it is mostly invisible to customers, purchasers, and end users. These parties observe the systems. They are affected by a reduced ability of the development organization to evolve or maintain the software product and, in more dramatic cases, a degradation in the overall quality. In the financial world, you drive the BMW, and there is no observable evidence

that you still owe 50% of it to your bank. In software development, end users use your software without knowing how much technical debt your organization owes on the product.

Some developers (and tool vendors and researchers) argue that defects—or any other form of visible low external quality—are technical debt. Some development teams even argue that unimplemented requirements are technical debt. We think that this makes technical debt too vast a category, rendering it a more or less useless concept. Defects and low external quality—such as poor performance, a cumbersome user interface, instability, and security holes—are not technical debt. They are just poor external quality; the system is not operating properly, and its problems must be addressed. However, the poor quality may be a consequence of technical debt. Software practitioners already know how to track and manage defects and unimplemented requirements. Technical debt refers to a class of issues not historically tracked or managed that represent the trade-offs among technical decisions and their changing consequences as the system grows.

As explained in later chapters, defects, new requirements, and technical debt must all be considered when planning upcoming work because all three compete for resources during software development. They all require the effort of the development organization and contribute in different ways to the value of the software product. But for now we limit our attention to what is inside the technical debt landscape.

Technical Debt Items: Artifacts, Causes, and Consequences

All software-intensive systems, regardless of their domain or size, suffer from some form of technical debt that negatively affects their evolution if it is not managed in a timely manner. This technical debt is not atomic or monolithic, but it can be decomposed into dozens or hundreds of items, which we call *technical debt items*, that accumulate over time.

A technical debt item is associated with the current state of a development artifact: a piece of code, build script, or test. It is a concrete development artifact that you can point to. What connects this item to technical debt is that

the state of the artifact makes further changes to the software system more difficult: Evolving the software system is slower, more costly, more error-prone, riskier, or even impossible. Technical debt adds some kind of friction to further development, making it harder. In practical terms, how do you manage technical debt items? They can be mapped to entries in the same tool you use to manage your backlog, or they can be mapped to your issue tracker.

Technical debt is a state of your software system, and it has multiple causes and multiple consequences. Each technical debt item has one or more causes. The most likely cause we have observed is schedule pressure. A development team takes a path that is expedient now to save time and effort and, very often, because of some imminent delivery deadline. But there are other reasons to take on technical debt, as the stories in Chapter 1 illustrate. For example, you may want to investigate a possible product solution with minimal initial investment. Or the developer may not know a better approach at that point. While most technical debt can be traced to some decision made by the development organization, consciously or not, other causes are not linked to any decision made by developers or anyone from the business side. Some technical debt is caused by changes that occur outside the system, when some other element evolved in such a way that your system now suffers from technical debt; your software system has aged. We will look more closely at this "technological gap" in Chapters 6, "Technical Debt and Architecture," and 10, "What Causes Technical Debt?"

Be careful not to confuse technical debt with the cause of that technical debt. The necessity of meeting a hard deadline is not technical debt. But that necessity may lead you to make an expedient choice that changes the state of an artifact. Or you may miss a deadline because the current technical debt slows you down, and this is a consequence of technical debt.

The consequence of most technical debt is the additional costs that the development organization will potentially incur in the future. "Let us do <this> now, and we'll decide later if we can afford to do <it> better." Essentially, the debt is not borrowed money but borrowed time—or, more precisely, borrowed effort—which the organization can translate into monetary terms. These additional costs do not always appear clearly associated with specific technical debt items. Instead, they manifest in the

form of reduced velocity (or productivity), longer release cycles, or even their effect on the development team's morale.

Technical debt, however, may have consequences other than costly future development; it may also manifest in more defects, by making the evolution of the system more error prone for other developers. For example, if the technical debt takes the form of missing documentation or code that is hard to read or overly complex, a developer might make changes to this code and introduce a bug inadvertently. In turn, this bug may have some impact on the value of the software and lead to future remediation costs.

Often, these consequences—unintended development, lower productivity, and system fragility—are first visible only to the development team. They are the externally visible symptoms of technical debt. By themselves, symptoms are not complete technical debt items, although some development teams and software managers mistakenly refer to them as technical debt. They require some deeper investigation to identify the actual state of the related development artifacts, as we will show in Chapters 4, "Recognizing Technical Debt," 5, "Technical Debt and the Source Code," 6, "Technical Debt and Architecture," and 7, "Technical Debt and Production." Using another analogy, if technical debt were a health issue, then stomach ache, coughing, or high body temperature would be the consequences, also called symptoms, while eating contaminated food or sitting next to someone who is ill on a packed six-hour flight would be possible causes. Relieving the symptoms, such as taking a fever reducer, often does not resolve the issue. To get the complete picture, it is necessary to determine the state of the lungs or the stomach to effectively diagnose the illness and treat the patient.

Principal and Interest

Using our metaphor of financial debt, such as the mortgage on a house, financial consequences can be articulated in terms of principal and interest. The principal associated with a technical debt item is proportional to the effort that a development team would expend to eliminate it. Similarly, the interest incurred by this technical debt item is the effort expended in additional development if the team leaves the technical debt in the system. Moreover, both the principal and the interest grow over time, as more

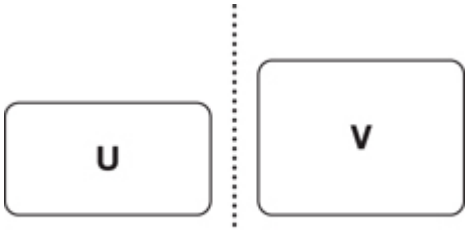
development that depends on the related development artifact is done, which ultimately makes paying off the debt more and more expensive.

Let us illustrate these concepts with a simple example.

Step 1: Incurring Some Initial Debt

You need to implement a new feature, inventory management, in your system and the underlying software stack. You can choose one of two design strategies:

- **Design option U:** A home-brewed stack based on the MEAN stack (Mongo DB, Explore.js, Angular.js, Node.js), which is expedient but not quite extensible; it has a low cost, say, six person-days.
- **Design option V:** A commercial middleware product, which is a much better design and is extensible and elegant; it has a higher cost, at ten person-days.



Solution U is cheaper than V

Note

The size of the box indicates development cost: The larger the box, the more effort in its development.

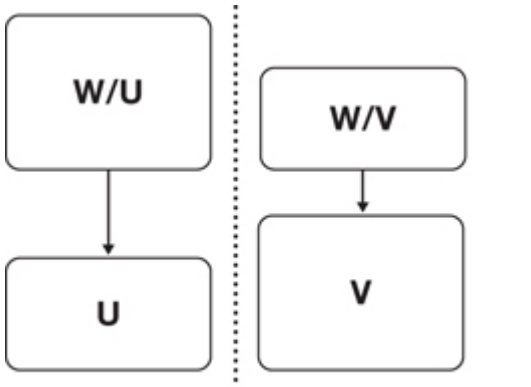
Because of schedule pressure, you choose the home-brewed stack, design option U, for the first release. Your home-brewed stack is not "buggy," and your inventory management feature will work perfectly well in either option.

Step 2: Evolving the System and Facing the Debt

For the second release, you want to implement a new feature, an order entry function, which depends on inventory management and therefore on U. Let us call the implementation of this order entry feature W.

Implementing order entry, W, on top of the quick-and-dirty home-brewed stack, U, is costly: W/U. It is more costly than it would have been if you had chosen the middleware product option, V, in the first place: W/V.

So, the code associated with the home-brewed stack, U, has incurred some technical debt. The interest you pay on this technical debt is the additional effort it now takes you to implement the new feature on top of U, relative to implementing it on top of the middleware product, V.



W over V is cheaper than W over U

Step 3: Deciding How to Treat the Debt

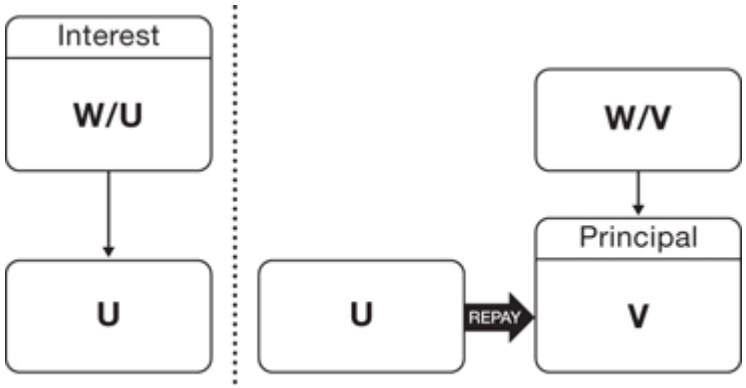
You now have another choice:

- You can repay the debt by discarding U and implementing the commercial solution V. Since U cannot be refactored, this choice involves paying the full price for V: the original principal that was the cost savings plus the cost of replacing U. By choosing this option, you avoid incurring any interest on new features developed on it.

- Or you can decide to implement the order entry feature W on top of U, with interest.

Your choice will likely be driven by immediate considerations of cost and schedule pressure and how much of each is involved. The interest—the additional cost of implementing W on top of U—is small compared to the cost of replacing the home-brewed solution U with the commercial middleware solution V and is not offset by the lower cost of implementing new features, such as order entry W on top of V.

If replacing U with V costs ten person-days, but the difference between implementing W on top of U instead of V is only one person-day, you may be tempted to choose the cheaper option: Accept the interest and postpone the decision to change U into V to some later day.

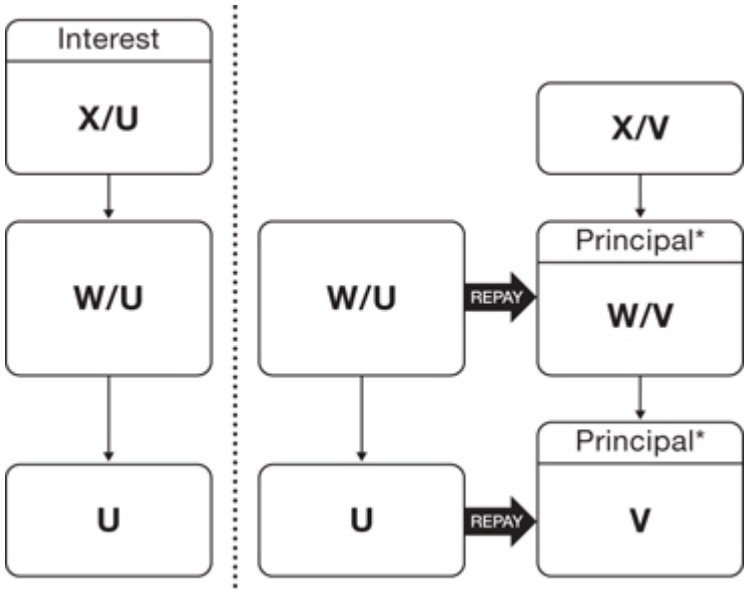


Pay interest, or repay the principal

Step 4: Just Paying Interest

If you do not repay the principal, your technical debt will continue to accrue as you add new features, implemented by X, as shown in the next figure. You have not added a new technical debt item, but you have made the current item more costly. At some later time, if you decide to replace the quick U with the nicer V, you will also have to retrofit W/U and X/U to make W/V and X/V; in other words, your implementation of each feature—W, X, and any others—on top of U would also have to be adjusted to the better V. The cost associated with moving from U to V therefore increases (principal* in the figure).

Interest is one of the key concepts of technical debt. In fact, understanding how and when interest accumulates in technical choices helps determine whether an issue should be managed as a technical debt item. Sometimes for good reason you choose to accept the interest. But often this choice is not conscious, and development costs increase. Managing such issues as technical debt items will increase their visibility so you can better assess the consequences and make informed decisions about the time and effort for treating the debt.



Pay more interest, or repay the higher principal

Cost and Value

Our description of technical debt so far has revolved around the costs, expressed as principal and interest. But like financial debt, technical debt has some value. You gain value in taking a mortgage to finance a home: You can enjoy the home now rather than wait until you are 60 years old to afford one. Similarly, software projects take on technical debt, consciously or not, because doing so creates some immediate value. And as in all other economic endeavors, there are trade-offs to be made. Cost must be constrained and value maximized.

Although we think of both value and cost in monetary terms—dollars, euros, or yen—they are different, and we will be careful not to confuse them.

Cost means any development costs: what it takes to get a system into the hands of its end users. For software-intensive systems, cost is primarily driven by compensating software developers. To estimate the cost of software developers, you must be able to estimate the amount of time they will spend developing. The cost comes in two forms:

- **Recurring interest:** The cost of the constant additional, possibly also growing, effort incurred because of the technical debt whenever the system must evolve.
- **Principal and accrued interest:** The cost of changing the design and the cost of retrofitting the dependent parts (the workarounds) in order to repay the debt.

A way to illustrate the difference between recurring interest and accruing interest is to look at a typical credit card statement:

Credit Card	Cost	Analogue in Software Development
Principal (at start of month)	$1,000.00	Current remediation cost
Interest for the month	$50.00	Accruing interest (coding a workaround)
Financial charge	$35.00	Recurring interest (slowdown of the team)
Balance due	$1,085.00	

The interest for the month corresponds to accruing interest; it is added to the principal as an obligation to pay in the future. The monthly financial charge of $35 corresponds to recurring interest; you pay it whenever your balance is not zero (a feature of credit cards in North America that you may not have experienced elsewhere!). Let's assume that you pay only this financial charge and do not repay the principal. The next statement will show the following balance:

Credit Card	Cost	Analogue in Software Development
Principal (at start of month)	$1,050.00	Remediation cost going up (accruing)
Interest for the month	$52.50	Retrofitting the workarounds
Financial charge	$35.00	Still slowing down general progress
Balance due	$1,137.50	

If you continue to defer payment, at the end of the year you will have paid $35 per month, for a total of $420 in financial charges, and the principal will have grown to $1,795.86 due to interest compounding monthly.

Where the analogy breaks a bit is that in software development, the interest is not a defined percentage of the principal. You may not accrue interest all the time; you may have only recurring interest. The principal may change for various reasons, and it may not be the principal incurred at the beginning. Other slight breaks in the analogy include the debt incurred by the passing of time that we described above, although you could think of it as not maintaining the cedar shingle roof of your house: It was perfect when you built it, but 15 years later, it has decayed and must be rebuilt.

Value is what the business draws as profit from selling the software, or the value perceived by the end users or the acquirers of the system. Value is

even more difficult to estimate and forecast than cost. The accounting department can only tell you about the actual value of sales so far.

When you evaluate, assess, and decide what to do about technical debt, you must deal with forecasting. Forecasting is difficult because it requires comparing different scenarios about the future, with different values and costs associated with them. One method for getting past the question of whether you are including the right details in your estimation is to use some proxy for monetary value. Many software development processes and organizations use points for cost: function points, use-case points, story points, and similar terms. There is no well-established proxy for value.

Let us revisit our simple example, this time to look at both cost and value. Whether you choose the quick-and-dirty U or the more elaborate V in the first release, the value of delivering a feature implemented with W in the second release is the same. But using design option U is cheaper. Remember that technical debt is not an externally visible defect, so the end user has received the same value at this point, regardless of the cost.

In the next release, you add the new feature, X. The total value delivered is the same whether you use design option U or V. But again, the cost with V is less than the cost with U.

To optimize value for a given cost over time through multiple releases, you should have chosen the more complex V. However, there is more to value than what features are delivered. Value is also influenced by when they are delivered. On one hand, choosing V would have delayed the initial delivery of W, potentially reducing its market value. But the investment in V would have reduced the recurring interest in adding a new feature as the system evolves. On the other hand, the value of incurring technical debt by choosing U is the time you gain by delivering additional value early. This is a valuable use of debt—that is, an investment resulting from understanding the business trade-offs and actively managing the technology roadmap—if the software development endeavor is highly risky and if you are ready to walk away from the system, never implementing feature X.

The Installment Plan
by Ben Northrop

There are often many options for paying down financial debt, and this is the case for technical debt as well. Though we might like to pay off each loan (or technical debt item) in its entirety, given the constraints, risks, and context of the project, it's often more practical to pay it off in installments.

A few years back, I was working on a kiosk application for a food service client. The system supported a number of features, from browsing the menu to calculating nutrition facts, but the main function was to allow hungry and time-conscious customers to place their orders quickly. This efficiency component was crucial, so the overriding design philosophy was to keep the user interaction as quick and easy as possible—so the customer would need only tap and swipe, never type.

Well, philosophies are made to be broken. About a year into the project, it was determined that a little typing was needed after all. Power customers would like to be able to log in to the kiosk to see their favorite orders and receive personalized recommendations, and until fancy QR code readers could be installed for instant authentication (á la Starbucks), we would need to implement a simple email/password login screen. We were assured, however, that this feature was a one-time thing, and it would be the only typing interaction we'd need to support.

With marching orders in hand, off we went to implement a virtual, on-screen keyboard for our new login feature. And trusting in the assurance that this typing feature was an anomaly, we happily embedded the code for the virtual keyboard right into the login screen. It was quick and dirty but effective.

Of course, a few months later, things changed. Focus groups showed that customers who had not originally logged in might still want the

ability to enter their email address after the order process, so, as we should have expected, we were now tasked with supporting a second instance of typing via the virtual keyboard. Had we initially implemented the keyboard generically (for example, abstracted it into some common widget), then reuse would have been a snap. Instead, our code was tightly bound into the login screen, and the solution would not be so easy. To further complicate the matter, we were in the final sprint of our release cycle, and capacity was stretched thin.

The decision before us was clear: Either we could go back and extract the virtual keyboard into a common component (as perhaps we should have done in the first place!), and incur the extra cost associated with the code refactoring and extra testing. Or we could take the easy road and simply copy and paste the original virtual keyboard code and embed it again into the feature code, this time the email entry screen. With the clock ticking, we took out a loan.

In the following months, our decision proved sound. We not only made our deadline but also found that the debt was manageable. Sure, there were a few minor defects or style enhancements in the virtual keyboard that required duplicate fixes, but overall the interest payments were low, and we were aware of the debt we would need to later pay (that is, it was on our backlog).

A few months later, however, a third requirement came for typing into our kiosk. At this point, we knew we did not want to increase the size of our debt by taking the copy/paste route again, but with a recently reduced team, we also knew we did not have the ability to pay down the entire principal in a single sprint. Was there a way we could avoid getting ourselves into more debt but also not pay off our loan in its entirety?

Our approach was to pay down our debt in installments. We recognized that the virtual keyboard was composed of three pieces: the style (CSS), the template (HTML), and the controller logic (JavaScript). Further, it was clear from the past few months of maintenance that the majority of changes were isolated to the look and feel and that the logic was generally stable.

Given all this, we decided to pay our debt in three chunks. In the first sprint, we extracted the CSS classes into our common stylesheet but left the duplicate template and controller logic in place. A sprint later, we tackled the template code, pulling it out into a reusable HTML snippet. In the third sprint, we abstracted a few of the nontrivial blocks of logic in the controllers into a common JavaScript library, and then we were done. In the end, this was not the purest design, and it was certainly not what we would have created if we had had perfect knowledge from the start, but it was pragmatic. In three smaller debt payments, we were able to consolidate about 80% of the solution, leaving only a very tiny debt of duplicated code that we were comfortable never paying off.

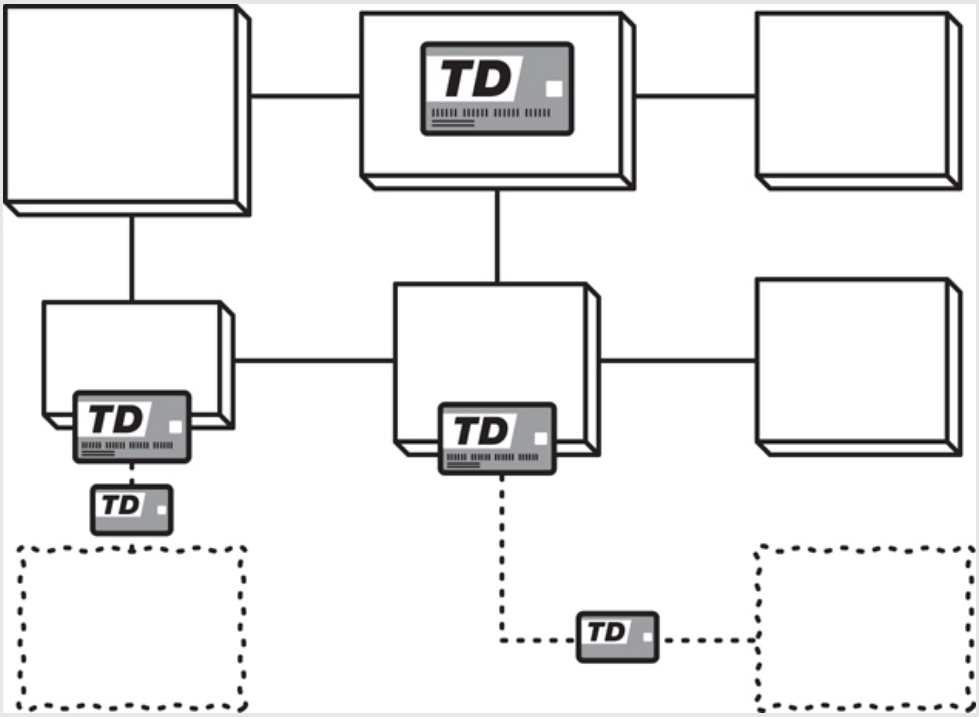
The point is that for any particular technical debt item, there is often a spectrum of options between the obvious poles of "do nothing" and "pay it all off." For us, though, either would have been a less-than-perfect end solution, and paying a technical debt in installments was a practical way to meet the demands of the business and also keep our technical debt under control.

Potential Debt versus Actual Debt

Not all technical debt items have the same impact. Whether an issue is actual technical debt or not depends on what future evolution a development team wants to make to the software system. If a technical debt item resides in a part of the code that is not affected by a future evolution, then this technical debt item is only potential debt. When it affects the evolution, it becomes actual debt.

The financial analogy again helps make this relationship to time and evolution concrete. If you borrow money from a bank with 0% interest initially, it is like free money; it is not incurring additional cost to you. Likely you will plan to repay it before the interest rate switches on. But until then, you may use the borrowed money for other purposes. You have the debt, but you have not started seeing its impact. This brings us to our second principle.

Principle 2: If You Do Not Incur Any Form of Interest, Then You Probably Do Not Have Actual Technical Debt



We use this principle as a litmus test when we classify an issue as actual technical debt or not. This gets tricky as the business and the technical context of your system changes; the likelihood of interest accumulating on design choices may change as well. For example, an area of the system with an abundance of problems may currently have zero interest if it is sufficiently decoupled from the rest of the system and does not need to be maintained. Understanding interest and how it changes is key to understanding and managing your technical debt. The amount of interest is not linear and fluctuates as systems evolve. Hence managing technical debt is not a one-time activity.

A system may have a very large potential debt, but at a given point in time, based on the next evolution increment, only a small part of this technical debt is actual debt. This distinction will drive our prioritization of technical debt remediation (or repayment). We focus mostly on actual debt first, and next on potential debt, depending on its likelihood of occurrence.

The Technical Debt Timeline

As you have realized by now from our description of technical debt, time plays a big role: Technical debt matters only as time flows and the software system evolves. If the system were to never evolve, you would never have to pay any interest, and therefore technical debt would not matter. Let us look at the technical debt timeline, which shows how technical debt unfolds over time (see Figure 2.2).

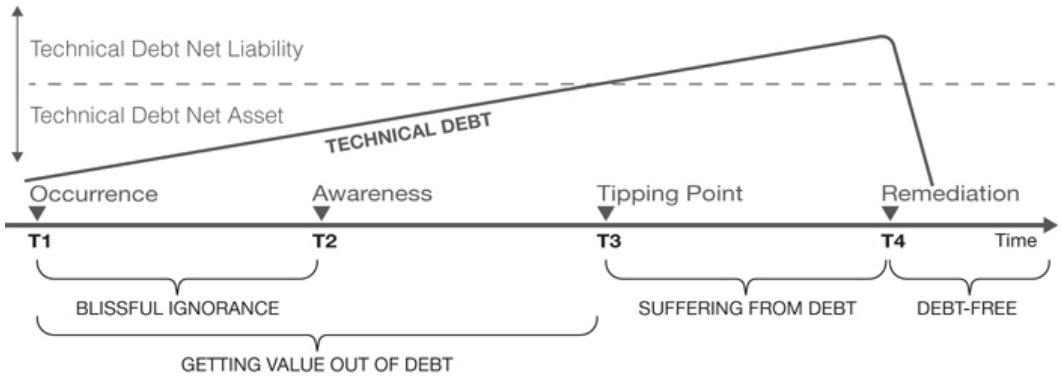


Figure 2.2 *Technical debt timeline*

T1: Occurrence

Occurrence is the point in time when a technical debt item is introduced into the system, for whatever good or bad reasons.

T2: Awareness

Awareness is when the development organization sees the symptoms of the technical debt item. For technical debt that was incurred intentionally,

through a deliberate decision and for some clear immediate benefit, T2 = T1. A development team made an explicit decision and records it. But for many development projects, lots of technical debt items just happen, unintentionally, and they will be discovered only later, when symptoms of slow development or defects point to strange workarounds, “fix me” comments, or “todo” labels in the code. The period between T1 and T2 is blissful ignorance.

T3: Tipping Point

The tipping point is the time when the cost of having technical debt starts to overcome the original value of incurring the debt. In the interval from T1 to T3, the slower progress due to recurring interest and the accruing interest of not-quite-right code that will need to be retrofitted lead to a situation where you could be better off repaying the debt. Before T3, you might just as well live with it; you actually get some value from the technical debt. T3 is an inflection point: you now pay more than you gain.

T4: Remediation

Remediation is removing the technical debt item from the system. The cost of remediation includes the initial principal and all the accrued interest. In the period from T3 to T4, the debt continues to accumulate interest. But also (unlike in the financial world), the principal may evolve to be very different from the initial principal. So, the remediation will often be more effort than just undoing the not-quite-right code and implementing what would have been the right solution at T1. The remediation might lead to a very different design from the one you forfeited at T1 because the context has evolved significantly. After T4, you stop paying the recurring interest, too.

…Or No Remediation

The remediation step at T4 may not be chosen. If the tipping point T3 is far in the future, the remediation costs may be prohibitive, so you postpone the

decision further into the future. Therefore, there may be a period from T2 onward during which you just live with the technical debt and accept to pay any recurring interest as you go. This is likely to be the case when the technical debt item is confined to a part of the source code that is unlikely to evolve in the future, so it will have very minimal recurring interest and no accruing interest.

Technical debt-savvy organizations may not wait to hit the tipping point at T3 and instead start remediation early.

In the following chapters, we will tackle key planning questions:

- Is incurring debt worthwhile?
- Once you have technical debt, when should you repay it?
- If you cannot afford to repay it all, which parts of it should you prioritize?

Software development organizations usually operate within budgets that constrain the costs. They try to optimize the delivered value at each step or release, and when they decide what to do at each step, they face competing demands on their budget, including adding new features, scaling up the system, increasing quality (in particular by reducing defects), and reducing technical debt. They need to estimate both cost and value for these competing demands.

What Can You Do Today?

The processes of developing, evolving, and sustaining software systems require making technical, organizational, social, and business trade-offs. The more these tradeoffs are made explicitly and communicated broadly, the more likely resources will be allocated strategically. Start today by understanding the rich vocabulary of technical debt and socializing the concept with the key stakeholders of the system:

- Provide a clear, simple definition of technical debt in the context of your project.
- Educate the team about technical debt.
- Educate the people in the immediate project environment about technical debt: management, analysts, product managers.
- Create a "techdebt" category in your issue tracking system, distinct from defects or new features.
- Include known technical debt as part of your long-term technology roadmaps.
- Extend awareness activities to external contractors if they are part of the project.

As you learn more about different technical debt management principles and practices in the following chapters, you will fill a toolbox that will equip you to deal with conversations about trade-offs and managing your technical debt strategically rather than being overrun by it.

For Further Reading

Steve McConnell (2007) of Construx was the first to establish a classification (or taxonomy) of technical debt, differentiating small, scattered, and mostly unintentional debt from large, intentional, and strategic debt. Martin Fowler (2003) of ThoughtWorks presented a different twist on the taxonomy, highlighting the "prudent but inadvertent debt" caused over time by an evolving environment.

Steve Freeman and Chris Matt (2014) have argued that traditional financial debt is not the best metaphor; software technical debt is more like an unhedged call option in the derivatives financial markets. The buyer pays a premium to decide later if he or she wants to buy. The seller collects the premium and will have to sell if the buyer decides to buy. It is not predictable for the seller. When you incur the technical debt, you collect the

premium: You immediately get benefit from it (shorted time). But as soon as you have to maintain or evolve this codebase, the option is called, and you have to pay an unpredictable amount of effort to achieve your new objectives.

The technical debt landscape emerged as the result of a workshop held at the International Conference on Software Engineering (ICSE) in Zürich in 2012 (Kruchten et al. 2012). The principles of technical debt came out of the workshop at ICSE 2013 in San Francisco (Kruchten et al. 2013).