The Mikado Method

Ola Ellnestam
Daniel Brolund

Foreword by
Tom Poppendieck

SAMPLE CHAPTER
The Mikado Method

by Ola Ellnestam
and Daniel Brolund

Appendix A

Copyright 2014 Manning Publications
brief contents

PART 1  THE BASICS OF THE MIKADO METHOD .................................1

1  ■  Meet the Mikado Method  3
2  ■  Hello, Mikado Method!  15
3  ■  Goals, graphs, and guidelines  38
4  ■  Organizing your work  54

PART 2  PRINCIPLES AND PATTERNS FOR IMPROVING SOFTWARE ..........69

5  ■  Breaking up a monolith  71
6  ■  Emergent design  100
7  ■  Common restructuring patterns  123
appendix A
Technical debt

This appendix covers
- Understanding technical debt
- Focusing on sustainable development and avoiding shortcuts
- Sources for redesigning and restructuring

Outside of the software world, you take out loans to finance different sorts of investments or expenses. You then have to pay back the principal (the money you borrowed) and interest (a time-based fee for borrowing the money). You take out loans both privately and in business-related situations.

The basic idea behind taking out a loan is to delay the payment for a purchase, from a time when you don’t have the purchasing power to a time when you do. Sometimes this means you pay back a little at a time over a longer period, perhaps over decades for expensive things such as a house. Sometimes it means you pay back everything in one lump sum quickly, within a week or sooner. Now, what has this got to do with software development?
Imagine the following scenario: The product owner of the site you’re developing comes in shouting, “We need to add social media integration to our customer support flow! And we need it by next week! We have a potential Top 100 customer we must impress!” This is the break you’ve been waiting for. Everybody on the team drops what they’re doing and starts hacking away at adding the new feature.

No one cares that the code isn’t designed for adding social media integration, and that there aren’t any abstractions that support it. With grease and pure force, the new functionality is implemented using a global flag and if statements like if(flag=="SocialMedia")... across the codebase, among other code good-practice violations. Everyone knows that this will take a lot of time to clean up later, but before that happens there are other important things that will suffer from this code frenzy.

**Good practice or best practice?**

Often in business you’ll find people using the term “best practice” for a way of doing something—for a process or a method. In our opinion, that term is misleading in several ways:

- It implies that if you follow the best practice you’re all good. But the reality is that if you do the wrong thing in the right best-practice way, it’s still the wrong thing to do. For example, you could take code that isn’t used and cover it with tests and refactor it, instead of just throwing it away.
- Which practice is “best” is so highly context-dependent that there’s no universal solution that deserves being called “best practice.”
- It implies that there’s no better way to do something, no room for improvement. In a complex field such as software development, there’s usually a better or equally good practice, and there’s always room for improvement.

For these reasons, we use the term “good practice” to express that we find something beneficial, but likely with room for improvement or with alternative ways to get the same, or a better, result. The Mikado Method is a good practice for restructuring legacy code.

By behaving this way, you have, metaphorically speaking, put yourself in debt—*technical debt*. You now have to spend a little extra time to understand what the code does, and you have to put some extra effort into making changes. This *extra time* is equivalent to *interest*. The *principal* of the loan is the time it will take to get the codebase back into shape again. This is why these types of problems in code are often called “technical debt.” The phrase *technical debt* was originally introduced by Ward Cunningham in the experience report “The WyCash Portfolio Management System” at OOPSLA ’92 (http://c2.com/doc/oopsla92.html).

So far, you’ve learned how the Mikado Method can get you out of a big mess and, in other words, repay your technical debt. Now it’s time to learn how to spot the early warning signals and see where the technical debt traps are.
But as in real life, not all debt is bad. You’ll learn what type of debt you must take on, or should take on, in order to hit the market window or seize an opportunity. You’ll also learn to see the big picture—that some technical debt doesn’t originate from the technical team, but from the business side, or even from outside of the company.

### A.1 How you get into debt

Like debt in the real world, the reason for taking on technical debt is to cut yourself some slack now, such as to get the social media integration working for the important client. The cost is having to clean up afterward, and living with the mess you’ve created until you’ve cleaned up.

---

**Definition of technical debt**

*Technical debt* is any existing technical structure that prevents you from effectively and efficiently making relevant and necessary changes to your software. The interest on the debt is the extra time you spend to deal with that structure. The principal is the structure itself. When repaying, the payment is the time it takes to refactor the structure.

As a metaphor, debt is quite easy to understand, but it has a bad connotation relating to loss of freedom, and of owing something to someone. While debt is debt, regardless of where it originates, understanding where the code problems come from can help you mitigate the root cause, or realize that the cause was a good investment, or perhaps just let you come to peace with the stressful situation of having to deal with it.

There are several ways to categorize debt builders. Table A.1 and the following sections categorize debt builders as acceptable, unavoidable, unnecessary, and bad. These are ways of looking at technical debt from a utility perspective. Note that it’s the debt builders we’re talking about, the cause of the debt, and not the resulting code. In some cases, a specific builder may produce certain types of problems, but in general the problems created can be of any kind. Table A.1 lists different categories of technical debt builders. The different builders will be explained in detail in the following sections.

**Table A.1** Categories of technical debt builders

<table>
<thead>
<tr>
<th>Category</th>
<th>Characteristics</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptable</td>
<td>Short-term loss for long-term gain</td>
<td>- Making a trade-off to hit a market window</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Waiting for the right abstraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Reflection and learning about your domain or technology</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Receiving new requirements that render the current solution insufficient</td>
</tr>
</tbody>
</table>

---
Table A.1  Categories of technical debt builders (continued)

<table>
<thead>
<tr>
<th>Category</th>
<th>Characteristics</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unavoidable</td>
<td>Caused by uncontrollable events</td>
<td>■ Working on the edge of chaos</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Inherited debt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Staff turnover</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Complying with new regulations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Changes in demand</td>
</tr>
<tr>
<td>Unnecessary</td>
<td>Good intentions produce bad outcome</td>
<td>■ Working under stress</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Having insufficient knowledge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Making premature optimizations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Adhering to the not-invented-here syndrome</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Adhering to the proudly-found-elsewhere syndrome</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Creating vendor lock-in or framework leakage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Writing one-way code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Writing too-clever code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Suffering from technical policies</td>
</tr>
<tr>
<td>Bad</td>
<td>Totally without benefit</td>
<td>■ Lack of communication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Unprofessionalism</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Habitual corner-cutting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Creating excuses for writing bad code—the broken window syndrome</td>
</tr>
</tbody>
</table>

A.1.1 Acceptable debt builders

Acceptable debt is when you take on debt as an investment to enable a gain of some sort, in the short or long term, that has a greater value than the cost of the debt. It would be better, of course, to get the gain without getting into debt at all, but in the context of debt, this is the good kind.

MAKING A TRADE-OFF TO HIT A MARKET WINDOW

This category of technical debt builder is where you have a known conflict between creating the most apt technical structure and meeting a delivery deadline. If there are considerable financial or market-position gains to be made from meeting the deadline, the cost of having to deal with a less appropriate implementation might be worth it.

Adding social media integration in the example earlier is a good example of this type of debt. Debt can be accepted to meet a deadline, but the debt is best repaid immediately after the deadline. Alternatively, you accept paying interest on that piece of code for a longer period of time before you pay back the loan by refactoring. If deadlines are constantly stacked up and there’s never any time to repay accepted debt, this isn’t an investment anymore—it has turned into credit-financed interest payments. This is generally a bad idea. It will keep you floating for a while, but sooner, rather than later, the debt will accumulate and you’ll risk spending most of your time paying interest, and very little time creating and delivering new features. In
some businesses, not being able to deliver new features means closing up shop, and for most companies it means getting a decreased return on invested capital.

Given a reasonably well-structured codebase, a rushed delivery often forces you to violate the abstractions in the code. When the new information, encoded in the mess, is to be placed in proper abstractions, substantial rearrangements are often needed, and such changes are well suited to the Mikado Method.

**WAITING FOR THE RIGHT ABSTRACTION**
Reducing debt often boils down to finding the right abstraction for your application. In some cases, the abstractions are obvious from the domain. At other times, you’ll need to try a few variations on your implementation to see where you can abstract and generalize it. You might have to put up with some temporary debt as you look for a good abstraction.

We once worked on a system with an object data structure that was programmatically navigated for aggregation, validation, transformation, and presentation with several different traversal implementations across the codebase. After a while, we realized that the underlying structure was a composite pattern, and the traversal could be implemented with a visitor pattern. After implementing that, with a great deal of help from the Mikado Method, the logic that previously was duplicated across the codebase was implemented in a single place. There were also a lot of conditionals for the traversals that were only interested in objects of certain types, or with certain attributes, so we added a filter pattern. This allowed us to remove all those conditionals and place each type of conditional in its own reusable class. By finding those abstractions, we were able to replace pages of code with one or a few lines. As time went on, we found more and more things we could implement with those abstractions, simplifying the codebase and removing technical debt even further. Realizing what we were looking for took us a while, and we couldn’t have been rushed into this epiphany.

**COMPOSITE AND VISITOR PATTERNS** The composite and visitor patterns are described in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et al. (Addison-Wesley Professional, 1994). The composite pattern is a way of creating an object structure, usually a tree, of related but heterogeneous objects without requiring the objects to know exactly what other objects are in the structure. The visitor pattern is an abstraction of how to traverse such a heterogeneous structure.

As in this example, you might have to incur some temporary debt while waiting for the epiphany of a good abstraction. Sometimes waiting is a conscious choice, and sometimes you wait because you have no choice. In either case, this is usually a good thing, because jumping to conclusions can cause an even worse situation with overly complex structures. This process can’t be forced, but it’s essential to software development, so it’s most definitely acceptable.

**REFLECTION AND LEARNING**
Many developers have a different idea about how they should have implemented their solution after they’ve completed it. They did the best they could at the time, but the
more they learn about the domain and the system, the more ideas they have about how their system could be even better.

Developers learn things from discussions with colleagues, from reading on the internet, and from attending sessions at conferences. With the incredible amount of knowledge available, a single person or team just can’t know everything, or even know how to apply the things they already know about in the best possible way. The things you learn can be minor details, or they could suggest a major, Mikado Method–sized overhaul of the application. Learning is an integral part of software development, and when you learn something and realize that your solution wasn’t up to par, this is totally acceptable.

Learning is essential. It implies that you reflect on your work and your solutions. You discuss your problems and solutions with others, and every now and then you get a pointer to something you didn’t know before. With this knowledge, and sometimes using the Mikado Method, you can morph the system into a new and better shape.

NEW REQUIREMENTS FROM STAKEHOLDERS
In order to keep up with competition, extend an advantage in the market, or accommodate new sales, your company stakeholders will update their vision and goals for the product you’re developing, resulting in new requirements. If you’re unlucky, you could end up in a debt situation, where the previous solution worked just fine, but the new requirements invalidate the previous designs and architecture. Until that moment, the application was fit for its purpose, but from that point on, adding more features in the old way just adds to the trouble.

Ola once worked with a start-up online bank that decided to add another type of account to their offerings. What before was a codebase rather fit for its purpose immediately felt a bit more cumbersome and unwieldy. By changing the current abstractions to accommodate the new account type, he once again streamlined the application for what it was supposed to do.

This debt builder usually comes from learning at the business level, where the product you’re developing gets better, so it’s an acceptable debt builder.

A.1.2 Unavoidable debt builders
Some things in life just can’t be avoided, and especially not in real-time business situations. Unavoidable debt builders are those that are just a normal part of developer life.

WORKING ON THE EDGE OF CHAOS
Software development involves constantly making decisions about which path to choose. Generally, the less chaotic the environment, and the more informed the decision-maker, the less risk there is of making bad decisions. Unfortunately, software development is a relatively chaotic field and the problems are ill-posed, at best. This makes it very hard to create the solutions that you, in hindsight, will consider the best, or even good. The debt that comes from this chaos is far from optimal, but you can’t slow the world down, or simplify it, as you please.
If you’ve ever been working hard, but with that uneasy feeling that your solution might not be exactly right, or that it is right but you’re not sure how it should work, and no one has the time to explain, that feeling might be a sign of a chaotic situation. Our best suggestion for avoiding or mitigating trouble is to be transparent about what you do and why, to expose problems early, to reflect upon them, to get feedback on them, and to take action without unnecessary delay. Symptoms often get harder to see the longer you’re exposed to them. Take a step back for a moment and look at the situation—how you got where you are today—and maybe you’ll see things you’re currently unaware of.

INHERITED DEBT
Often, you’ll get to handle troubled legacy code written by others— inherited debt. This isn’t a delight, but it’s something you have to deal with as a software developer. If the people involved in creating the inherited version are still around, you can talk to them to get a hint of why things are the way they are. The reasons may still be valid, and knowing what they are may help you avoid ending up in the same situation again.

On the other hand, the reasons may not be valid anymore. Once, Ola was asked to take over a system where the original developers were about to quit. The system was written using an object-relational mapping framework, and the performance degraded horrifically as the amount of data grew, partly because of the framework itself, and partly because of the objects created by it. Using this framework was the easy path functionality-wise, but only for the initial load. It took about a year and the assistance of a colleague to start getting the functionality performant, slowly removing the old structures piece by piece, making extensive use of the Mikado Method.

STAFF TURNOVER

Whenever you visit code for the first time, you can feel awkward. You’re not 100% sure what it does, or how to implement things. It always take some time to get up to speed on a codebase. The same thing happens when you replace members on a team. The departing people take some unique knowledge with them, and the new
people, especially when under deadline pressure, might add things in ways that add to your problems.

If you’ve ever come in as a replacement for someone on a team, you know the hopes everyone has for you, and you’ve probably felt like you were letting people down when your solutions haven’t matched their expectations for that environment. It takes some time to get up to speed, and in the meantime debt might be created. When you’re new to a codebase, take some time to experiment with the code and document your findings with the Mikado Method. If you can pair with someone who’s familiar with the codebase, or just discuss your findings with others, you can get on a fast track to knowing how the application is stitched together. You can also teach others about the Mikado Method and possibly create even more mutual benefit.

Staff turnover is unavoidable, and it isn’t bad per se. In the short term, there might be a dent in productivity when you’re getting someone up to speed, but in the long term, the added knowledge and perspective often improves speed and structure in a solution. It could be viewed as learning, but the reason for this learning is that knowledge was lost when the previous team member left, so it’s placed in the unavoidable category.

NEW EXTERNAL REGULATIONS

Changed regulations or laws can sometimes require significant changes to the architecture and implementation of a product. This is a sort of technical debt that’s not caused mainly by the development team, but by forces outside of the team, and even outside of the company.

When the European market opened up for online gaming, such as poker, governments all over Europe required that every transaction be approved by a central national authority. The companies that wanted to compete in those markets had to adapt their implementations to those rules.

These kinds of changes are hard to predict. When you know you’re working in a market that’s heavily regulated, you can only prepare for it—you can’t anticipate it. By automating the building, packaging, documenting, and verifying of your product as much as possible, you’re best prepared for whatever changes come along.

New regulations can have a significant impact on your system, requiring large changes. With regulatory compliance as the Mikado Goal, and a clear graph, there should be little argument over any changes tied to that goal.

CHANGES IN DEMAND

The market, or the customers, do change their minds from time to time. Things come in and go out of fashion, and new technologies make existing products obsolete. There’s usually little you can do about it but try to adapt. Sometimes the associated changes are just a cost, but sometimes they could harness possibilities. Change isn’t always bad.

For example, Flickr, one of the largest online photo-sharing sites, started out as a photo upload tool in the massively multiplayer online Game Neverending. The tool proved to be much more popular than the game, and it was extracted to a chat room with real-time photos. The chat room was also later discontinued, in favor of the current website for uploading and storing photos. The game, like the chat room, was
discontinued as the photo service was refined. During those discontinuation transitions, the amount of technical debt—code that was in the way—in their codebases should have been substantial.

A.1.3 Unnecessary debt builders

Some debt originates from the excessive ambition of developers or management, from lack of experience, or from systemic problems. These are unnecessary debt builders, but because there’s a positive and ambitious undertone in the activities that caused the debt, we don’t place them in the bad-debt category.

STRESS

![A situation of stress](image)

Stress makes you less aware of the current situation, leading to tunnel vision where decisions are often based on an unusually incomplete or distorted set of data. Those decisions tend to create solutions that aren’t optimal. You can be pretty sure that in the social media integration example, the extra stress will cause additional debt to be created, beyond that created by consciously cutting corners.

All people have different thresholds when it comes to stress; at low levels of stress, humans tend to produce slightly better results, but high levels invite you to take shortcuts and make unnecessarily short-sighted solutions, adding to the problems that already exist in the solution. Sometimes you’re aware of the stress, but other times stress creeps up on you. Excessive stress, frequent or extended, is often a systemic problem and can only be solved by making changes to the system. Are the developers in the team pushed to meet deadlines beyond reasonable expectations? Are the teams not permitted sufficient control over their workload? The underlying systemic problems must be dealt with.
Stress comes from caring about the outcome of the situation, and caring is a good thing that should be nourished. Indifference or apathy would be much worse. Management is always responsible for creating a healthy environment, and the excellent book *Behind Closed Doors* by Johanna Rothman and Esther Derby (Pragmatic Bookshelf, 2005) gives great insight into how to become a better manager. But the problems can’t be blamed exclusively on the system or the managers; all people involved have a personal responsibility to warn their managers and colleagues when working conditions are so unhealthy that they cause bad code.

**INSUFFICIENT KNOWLEDGE**

When you lack tool or programming-language knowledge, you risk creating an implementation that’s unnecessarily bulky, or that does the right thing but in an awkward way, thereby adding to the technical debt. In addition, if you have no understanding or a poor understanding of the business and the domain you’re working in, you’re likely to build up debt. Just as a lack of technical knowledge can contribute to bad constructs, insufficient domain knowledge can lead to similar problems because you’re missing important pieces of information. This is also true if the person specifying what to do lacks knowledge of the domain.

The lack of correct information can lead to problems ranging from minor defects to using incorrect assumptions as the basis for important parts of a product. The cost of fixing this at a later stage might be high. Try to verify your assumptions with real code, real tests, and real users as early as possible. Try to get together with the business side, or the customers, to learn how they talk about and interact with the domain. If you find that things have gone horribly wrong with your implementation, the Mikado
Method may help you out of that, but it’s not a replacement for communicating with the people involved.

**PREMATURE OPTIMIZATION**

Making optimizations before you’ve tested your application with real data often leads to an overly complex design that rarely targets the right bottleneck, and that needless complexity is just added technical debt. In a complex system, it’s virtually impossible to foresee where the bottleneck will be, or even to predict where the next bottleneck will be if you’ve already identified your current bottleneck.

We’ve seen several incomprehensible pieces of code that were the result of premature optimizations, sometimes in parts of the code that had neither performance nor load requirements. It was optimized only because “code should be fast.” Untangling that code can be a challenge, and in ugly cases you might find the Mikado Method helpful.

Structure your applications well, write the clearest code you can, and then optimize when you must, based on real usage, or preferably on crafted load and performance tests (based on expected or real usage). Good structure often leads to good performance as well. If you have extreme performance or load requirements, you’ll need to consider performance at an early stage in your architecture, but even in those cases, the majority of the application won’t be on the critical path of performance and your focus should be on clarity.

**NOT INVENTED HERE**

The not-invented-here syndrome manifests itself as an unwillingness to use existing software components, frameworks, products, or knowledge. Time and energy is spent on writing proprietary code for components that are commonly available, instead of focusing on the core business. Because the component isn’t part of the core business, its maintenance is likely to suffer, sooner or later making it a debt rather than an asset.

When you find a local implementation that can be replaced with an external library, it’s often easiest to encapsulate the functionality behind local interfaces, and then replace the implementation with the external library. This is almost exactly what
you saw in section 2.2, where we encapsulated the database calls. As in that example, the Mikado Method is often useful when the implications are many, or when they go deep into the codebase.

**Proudly Found Elsewhere**

The opposite of *not invented here* is *proudly found elsewhere*. This leads to a *framework frenzy*, where no piece of code can be developed; the focus is always on analyzing and choosing an existing library or component that contains the feature. Developing a system becomes a *scrapheap challenge*, where the system is built by fitting together different libraries that do what’s needed and a million things more. Navigating and improving such a system is often difficult because the libraries are entangled with the system, and just penetrating such a system means wading through unnecessary complexity.

The decision to introduce an existing library must be balanced not only against the cost of developing your own code for that functionality, but also against the cost of decreased control and increased complexity. In the book *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison Wesley Professional, 2004), Eric Evans discusses strategic domain-driven design: when is an existing library more effective, and when should you consider developing your own? It comes down to the strategic mission statement for the product or service and following the consequences of that statement; if the code is what creates your competitive edge, according to the statement, you should develop it yourself, even if there’s software available that does the job. If not, you can use a standard library. The important thing is that you *must* be able to adapt and change the strategic parts of your software to keep your edge.

Replacing frameworks can also be a tough job, requiring the Mikado Method. The process is similar to when you replace a local implementation with a library, but the purpose is the opposite. Take the code that does the job and encapsulate it with interfaces, and then make your own implementation of those interfaces.

**Vendor Lock-In or Framework Leakage**

When you use a framework and let that framework leak into most corners of your codebase, you’re suddenly in a *vendor lock-in* situation. You’ve let the third-party library take over your application instead of using the library to solve a problem.

If you want to use a third-party library, the abstractions should come from your domain, and then you should implement those abstractions to encapsulate the library. That way, you don’t let the library dictate your implementation. The library merely becomes a tool to help solve a specific problem, and it’ll be much easier to replace if necessary. If your framework has already leaked, use *not invented here* or *proudly found elsewhere* to contain the problem.

Some vendor lock-ins can be acceptable, such as choosing a platform for development. But care should be taken so that, for instance, licensing for the platform doesn’t limit the development, testing, or deployment of the solution.

**One-Way Code**

*One-way code* is when a codebase is unnecessarily restrained so that you can only do things one way. Where abstractions are used, they’re used to limit the freedom of an
How you get into debt

implementation rather than create options. Performing any task, however simple it may be, might involve making changes to the entire codebase. If a customer asks for a new feature, it may not be feasible to implement it within the existing architecture. This is not a template situation for being productive in creating good software.

Ola and Daniel remember

We were once in a situation where dynamic, context-dependent menus were considered for an application, but the existing framework for the menus didn’t support that, so the team had to change the framework to allow for more dynamic handling. As they did that, they realized that the initialization of the menus was tangled up in the initialization of the application, so the initialization of the application had to change as well. But that could only be done if a lot of global variables and singletons were removed first. Those global variables were in turn used throughout the application, so the team had to refactor all the code where these globals were used. This, of course, took quite a while longer than the initial estimate indicated it would.

One-way code is riddled with limitations. Either the author deliberately restrains the flexibility without having a business case for that, or it happens as a result of the programming style of the author. In either case, if these restrictions aren’t part of the business case, and they rarely are, this puts unnecessary limitations on future development.

The opposite of limitations is options. Good code has abstractions that put the fewest possible restrictions on future development, in order to preserve options. This is not the same thing as making something “future-safe” by implementing code for all future scenarios that can be imagined. This is putting as few constraints as possible on the code to minimize the risk of having to change it when adding more code.

To resolve such limiting code, you’ll often need to deal with a lot of dependencies. This is a situation where the Mikado Method can help to get you on the right track.

TOO-CLEVER CODE

Too-clever code is code where the author has used all available tricks known, creating a solution that’s way more complex than the problem solved.

Ola remembers

Ola was once involved in a project where he and his pair-programming partner came up with a really clever solution. They used built-in abilities in the programming language and managed to create a very flexible solution. The code parsed an XML file and objects were created from the XML structure, depending on what elements it contained. The code was compact and highly efficient, but it lacked something. Two weeks after the code was deployed, a bug appeared. Two other team members looked at the code and were blown away. Not by its cleverness, but because they had such a hard time understanding the code. They just handed the code over to its originators, saying “We don’t know what to do. You guys will have to fix this yourselves!”
One would think that a clever solution would be preferable. After all, “clever” has a positive connotation. But programmers would take a clear, easy-to-read, and understandable solution over a clever one any day. As a rule of thumb, *one point of clarity is worth more than one hundred points of cleverness*. Sometimes, this rule is in conflict with keeping the code free of duplication. When that happens, clarity is more important than removing all duplication.

The computer is just as good at executing clear code as clever code, and sometimes even better. The Java Virtual Machine (JVM) is a good example of this. The JVM has built-in optimizations for about 200 common programming patterns. Clever code can throw it off and might even result in worse performance because it can’t find an optimization for the code.

To deal with clever code, change the program piece by piece by making sure the intentions of every piece of code are clear. This is like unrolling the implementation. You might have to use the Mikado Method when doing this, but more often it involves working on a smaller scale, method by method or class by class.

**TECHNICAL POLICIES**

Many organizations and corporations have technical policies that state what products are allowed to be used. Often the idea behind this is to simplify the work for IT operations, in an attempt to limit the number of technologies they have to handle.

In the eyes of the operations manager, this seems a rational decision. But different tools are good for different tasks, and using the wrong tool will force developers to create technical debt in the shape of workarounds. This means that interest has to be paid throughout the lifetime of the product. If you’re trying to provide the best value for the money, or to compete with the best in the market, this could be crippling, or even fatal, for the business. Sometimes it will work just fine, and the products implied by the policy might be sufficient, but in other cases this will incur a debt that will take a toll on the profitability of the product.

In any competitive market, technical decisions should be made based on the product and its usage. Any technical policy should be a preference rather than a dictate. If the technical policies are lifted, the Mikado Method can help when migrating to using more appropriate tools.

### A.14 Bad debt builders

Some debt is bad in the sense that it comes from a lack of respect for customers, coworkers, managers, employees, or shared resources. When you see these types of debt builders, you can be fairly sure that the problems go deeper than just the technical parts of the application.

**LACK OF COMMUNICATION**

When different parts of an organization isolate themselves or excessively formalize their communication, other parts of the organization build structures or behaviors around that. Instead of people just talking to each other, you get not one, but two unnecessary filters between the people who are trying to communicate. This leads to
distrust, disrespect, an us-versus-them mentality, and a poor flow of information. This, in turn, leads to the different parts of the organization trying to solve problems in isolation, creating multiple structures for the same purpose within the organization, often including technical solutions. That leads to technical debt situations.

**UNPROFESSIONALISM**

Technical debt isn’t an excuse for letting your system or development environment decay into a mess. You should always do the best you can, and apply good development practices such as using just-in-time design, incorporating test-driven development or the like, writing clean code, taking care of the continuous integration environment, and so on. To dismiss this responsibility for a system is just plain unprofessional.

**HABITUAL CORNER-CUTTING**

It’s common to be asked for estimates and then face pressure to reduce them in the hope that this will result in work being done more quickly.

Manager: *How long do you think this task will take?*
Programmer: *Well, I looked into it, and I’d say four days.*
Manager: *Four days?!*
Programmer: *Yeah, this new editor that we’re creating looks a lot like two others…*
Manager: *So…*
Programmer: *…so I was thinking we might pull out the similarities in a base editor…*
Manager: *Isn’t there a faster way?*
Programmer: *I guess we could finish one, then just copy that and do some small changes…*

To be done sooner, do less instead.

When you cut corners, you start on a downward spiral that steadily adds to the needless complexity of the application. It’s very easy to end up in a situation where you compensate for previously cut corners by cutting new corners, adding complexity upon complexity. This is like taking credit to pay the interest on your loans. If you don’t start improving the code, but keep cutting more corners, you’ll get deeper and deeper in debt with every change you make.

**BREAKING WINDOWS**

In criminology, there’s a phenomenon often referred to as the *broken window syndrome*, first presented in 1982 in an Atlantic article by James Q. Wilson and George L. Kelling called “Broken Windows” ([http://mng.bz/Cw7O](http://mng.bz/Cw7O)). The title comes from studies of houses left unattended. When a well-cared-for house is abandoned, it can stay untouched for a long time, perhaps even years. But if someone breaks a window in the house, it isn’t long before the house is completely vandalized. The phenomenon is manifested in our unwillingness to break or stain something nice, but when the first scratch on the surface is made, the rest will soon be ruined. This principle holds true with code: deterioration often starts with one small thing, like a broken window.

Similarly, working on code that’s already messy can be demoralizing, and it’s easy to just wreck some more of the code, creating more technical debt.
Creating trouble-free code in that situation is very cumbersome. The easiest way is to fix the codebase by fixing one piece at a time, and then to not break them again.

**LACK OF RESPECT**
Sometimes a team member knows what’s expected, and is capable of complying with those expectations, but they still play by private rules. This can mean anything from checking in code that doesn’t compile to refusing to share knowledge with other developers. It’s a way of saying, “I’m more important that you,” and that kind of behavior is bad for a software development team for several reasons.

First, it will make the current state and the expectations more unclear. This, in turn, will have an impact on the code. It increases the risk of having different rules for different parts of the codebase, adding to your problems. Developers risk becoming more keen to take ownership of certain parts of the code and to build protection against the other parts, causing technical debt to grow.

**UNRESOLVED DISAGREEMENTS**
When there are unresolved disagreements, such as about how to implement certain things, there’s a great risk that you’ll start building technical debt. If a team can’t settle on an implementation, the application is likely to have two or more ways to implement a feature. It may be that all the suggested ways are perfectly valid, and that the difference is a matter of taste. But even then, having to learn and navigate several different implementation styles takes a serious toll on the development effort.

Try to resolve any disagreements early. If the situation becomes difficult to manage, bringing in an outside facilitator for that particular task is probably worth the cost. As a last resort, consider making changes to the team if important problems aren’t resolved.

**A.2 More about technical debt**
We often use the metaphor of *technical debt* when we talk to non-technical people and try to explain how bad code affects progress and profitability, especially when we communicate with people who work with management, economy, and financing rather than technology. Often, they’re unaware of the consequences of an ill-structured codebase, but they’re very familiar with the do’s and don’ts of debts. In general, they’re terrified by the fact that they were put in debt without even knowing it, grateful for the new insights, and they slowly become frustrated when they realize they have to pay it back.

**A.2.1 Lactic acid**
Debt isn’t inherently bad, but if it isn’t handled properly, it’ll eventually introduce more problems than you bargained for. To illustrate this, let’s look at software development through our sports glasses for a moment.

There are situations where a short sprint can win the race. In a bike race, riders use this strategy to gain advantage over their competitors. A perfectly timed spurt at the
very end of the race, or a sprint during a critical part of the race, can literally leave the
competition behind.

The price of this strategy is an increased load on the system. If the racer doesn’t get
a breather shortly thereafter, the lactic acid produced during the sprint will trigger
the opposite effect and slow the rider down. If they go for too long without clearing
the acid, they might even hit the wall and collapse.

Deliberately using bad structures or cutting corners can give you a temporary
advantage over the competition. It can be viewed as the equivalent of sprinting, which
will result in the equivalent of lactic acid—technical debt. Used properly, this
approach can truly be a performance booster, but using it as a regular strategy will
bring you to your knees.

The body has a natural process for ridding the system of lactic acid, but software
doesn’t. If you want to keep your product in a healthy state, you need to take into
account the time it will take you to rid the codebase of the accumulated debt.

A.2.2 What is the interest?

Different debts have different levels of interest. To illustrate this, consider Kenny, an
ordinary guy with a weakness for betting. Kenny has several loans: one on his house
and one for his new car, and he also borrowed some money from a relative. On top of
this, he has a gambling debt to a mob-associated loan shark. Not a very nice guy. If
Kenny comes across some money, where should he start to pay it back?

In this case, it might look rather easy. Paying back the loan shark is likely to be a
higher priority because those kinds of loans come with very high interest, along with
very persuasive methods for getting repayments.

Technical debt in software projects comes in different shapes and with different
interest rates, but the interest depends on what you set out to do, not what type of
debt you have. In fact, a single type of debt might have different interest rates at differ­
et times. It can be zero interest if it’s in a part of the code you hardly ever touch, or
“mob interest” if it stands in the way of delivering important features.

When you spend time paying back debt by randomly refactoring code, you might be
prioritizing repaying the loan to mom and dad before the loan to the mob. But in soft­
ware development, the difference between the types of debt is much more subtle. In
order to identify your mob debts, you need a way to find them. The Mikado Method can
help you sort out the critical paths in your change effort to find your mob debts.

ZERO-INTEREST TECHNICAL DEBT If you don’t need to touch or read a part of a
system that’s part of the debt, you don’t need to pay any interest. It’s likely
unwise to repay that debt, because there are usually more urgent needs in
other parts of the system. Focus on the important parts, and the parts that you
change frequently.

From a debt point of view, it doesn’t matter how you got into debt. You still have to pay
the interest every time you come across that part of the system. If you ever want the
interest payments to go away, you need to repay the principal. *Debt is debt, no matter the cause,* but to avoid getting into debt again and again, it’s important to know where it came from and avoid the creation of debt altogether.

### A.3 Influence and type

Technical debt doesn’t only come from technical work. It also comes from the surrounding organization, from the integration of supplied APIs, and from the greater society we live in. Another way to look at technical debt is to assess it in terms of its *influence* and *type*.

*Influence* is the extent to which your organization has control or influence over the process. Was the debt externally imposed (out of your control) or internally incurred (created by your organization)?

The second dimension, *type*, indicates to what extent the debt is related to technical details or has more of a market focus. Is the debt related to technical and operational decisions, or is it the result of a change in laws and markets, visions, or business models?

Figure A.5 shows these factors in a matrix. The matrix consists of four categories. Let’s look at them a bit closer, starting with the upper-left corner.

![Figure A.5 Influence and type quadrant](image-url)
A.3.1 Third party—technical and imposed

Technical and imposed debt is related to technical changes outside of your control. An example is when BigCorporation buys SmallCo, your database vendor, just to discontinue the product. You have no influence over that decision, but you have to deal with the consequences by either changing your implementation or by living with the fact that you no longer get any support.

Debt from this quadrant can be very expensive to pay off, depending on how your system is structured. But still, it must be done if you don’t want to throw your solution away and start afresh, and starting afresh is often is a bad decision.

We were once developing a product in which a map engine was integrated. Unfortunately, there was a bug in the map engine, and the bug fix was only available in the latest major version, in which the APIs had changed significantly. Because the old API had leaked into the application in several places, we had quite a lot of work to refactor those parts of the code. We had put ourselves in a bad situation and ended up in a vendor lock-in or framework leakage situation (see section A.1.3).

If an application is built up by calls to different frameworks, as described in section A.1.3, the risk of getting into trouble increases. The risk of running into problems with discontinuations or other major changes should be balanced with the gains of introducing a framework.

A.3.2 Bigwig—market and imposed

In this quadrant, the rules or regulations have changed and there’s not much you can do about it. You either comply or your product is unusable, or in the worst case, illegal.

Things keep changing, and you’d better get used to it. New laws are created, which could mean that an application that was really fit for its purpose yesterday may have to change a lot today in order to comply. The market can also change, and customers may change their behavior. You can try to change the course of events, but unless you work in a very large organization, you’ll probably not be able to influence the decision.

In the northern parts of Europe, especially Scandinavia, young adults are less and less likely to get a phone number tied to a physical address (a number for their house or apartment). They grow up with a mobile phone and stick with that. This slight change in behavior isn’t something a service provider can control or predict. But consider how this change is imposed on companies. A simple registration form that worked perfectly well a couple of years ago might now need attention because the phone number validation has changed or the mobile number that once was optional is now considered the most important one.

More examples of debt builders that relate to this category are new external regulations and changes in demand (section A.1.2).

A.3.3 The boardroom—market and incurred

Software developers often experience the effects of management making quick decisions and swift changes in order to keep up with competitors. The challenge is to keep up with and embrace the changes.
Some business decisions can have a bigger impact than initially expected, such as corporate policies. These decisions (or rather, the lack of options) often steer you toward early limitations in projects, sometimes even before the projects have started. For example, “We must use BigRelationalDatabase, it’s a corporate policy.” BigRelationalDatabase might be a really good relational database, but when the project needs to store big chunks of loosely structured data in a massively distributed system, it might not cut the mustard. By forcing technology onto a project, a debt is created that will diminish the returns, or even push the project over the edge.

Examples of debt builders that relate to this category are making a trade-off to hit a market window (section A.1.1), technical policies (section A.1.3), and new requirements from stakeholders (section A.1.1).

Consider a loan institution, like a bank. They’re humming along nicely, and one day the owners of the bank stumble across an opportunity. They receive an offer to buy their competitor’s customers at a very reasonable price. The deal is quickly closed, and soon the only thing left is to import the competitor’s customer database into the existing back-office system.

Most of the new customers have contracts similar to the ones already in place, but some are different. Some have a loyalty bonus in the form of an individual, significantly lower, interest rate. There are also customers who aren’t amortizing on their loans during their first six months—they’re just paying interest. Neither of these two contract types can be accommodated in the existing system. Deciding to expand its business has put the bank in a situation where structural changes need to happen or a lot more manual work will be introduced. If the former is chosen, the back-office system that was working fine before just isn’t any more.

Even though turning down or following through on a deal is a business decision, its impact can be more readily handled if the technical side is involved early on.

**A.3.4 Propeller hat—technical and incurred**

This quadrant is where most of the technical debt we’ve seen comes from. It can be the result of anything from sloppy coding, to a don’t-touch-it-it-works mentality, to ambitious and well-intended over-engineering. They all create technical debt.

Early choices on architecture and design that once looked like good decisions can become the biggest hurdles when it comes to effective software development, forcing you to create workarounds that add complexity to your solution without giving anything in return. But it’s a balance, and some decisions have to be made. Choosing a database too early is often a bad idea, but so is staying too long with saving your data to a flat file.

The most sensible technical changes often start with a hunch and maybe a feeling that you’re missing a point. The code may feel awkward, inflexible, and stale like yesterday’s bread. Sooner or later that feeling is followed by a new way of looking at things and usually an idea of how to improve the system.
When you’ve decided to make an improvement, make sure you implement it all-out to avoid leaving the codebase semidisrupted. We’ve seen our fair share of semidisruption; one case that comes to mind is where Apache Struts was almost replaced by WebWorks, which in turn was almost replaced by the Spring Framework, leaving the system with no less than three different ways of presenting and accessing data. Like growth rings on a tree, the system presents its structural shifts to its caretaker.

Although sound changes stand the best chance of leading to improvement, the change can add to the debt unless everyone is aboard and the change is fully implemented.

Watch out for these examples of debt builders and red herrings that relate to this category of debt: waiting for the right abstraction (section A.1.1), insufficient knowledge (section A.1.3), premature optimization (section A.1.3), not invented here (section A.1.3), proudly found elsewhere (section A.1.3), one-way code (section A.1.3), too-clever code (section A.1.3), and breaking windows (section A.1.4).

A.3.5 Borderliners

Some of the sources of technical debt can’t easily be put into a single category. Rather, they span several of these quadrants, or their origin differs from time to time. This is due to the fact that technical debt is often created when internal or external borders of communication are crossed.

A.4 Attacking the technical debt source

Poor internal structures constitute a multitude of problems that might all need to be handled sooner or later. If you look closer, you’ll find that there are no less than four different ways to deal with a problem:

- Absolve—Do nothing; just forgive the people or the process.
- Resolve—Take knowledge acquired in the past from a similar situation, and apply it to the current situation. This is the preferred way to deal with problems, and it’s what we’re taught and tested for in school.
- Solve—Sit down and analyze the problem, and try to find a solution that’s better that the current one.
- Dissolve—Eliminate the cause of the problem.

In order to show how a problem can be approached in several different ways, we’ll illustrate it with a story, once told by Russell L. Ackoff in *Idealized Design* (Prentice Hall, 2006). Throughout the story, you’ll see examples of absolving, resolving, solving, and finally dissolving the problem.

In London, double-decker buses have a driver and a conductor. The conductor collects fares and issues receipts from the back, as passengers board the bus. When all have boarded, the conductor signals to the driver through the front mirror that everyone is on board and they’re ready to go.
To make things a bit more complicated, there are incentive programs in place—one for the driver and one for the conductor. The driver is paid more if they stay closer to schedule; the conductor is paid more the fewer fares he misses collecting, which is controlled by undercover inspectors. The driver is highly dependent on the efficiency of the conductor.

To prevent delays during peak hours, the conductor lets everyone board the bus and collects fares and issues receipts between stops. Sometimes the conductor isn’t fast enough, or fails to signal the driver when everyone has boarded. This results in a growing hostility between the drivers and conductors.

When this problem surfaced, management chose to ignore it, hoping that the problem would disappear on its own. This is the problem-absolving approach. In an ever-changing environment, problems can disappear by themselves, but they didn’t in this case. The problem actually got worse, so they had to do something.

They decided to change the incentive system, which they thought was causing the problem. Ackoff calls this reversion to a previously working situation resolving. But the conductors and drivers had gotten used to their extra money and rejected the idea.

Next, management decided to solve the problem and find the best solution. After thinking about the problem for quite some time, they came up with the idea that the drivers and conductors should share the incentive payments. This wouldn’t affect their paychecks. The company wouldn’t have to pay more, and it would encourage the conductors and drivers to work together. If they had started out with an incentive system that looked like that, the problem might not have surfaced or would have taken longer to discover. But at this time, conductors and drivers were reluctant to work together, and even more so when it came to sharing money.

Management was on the verge of giving up, and as a last resort they consulted an expert, who happened to be familiar with problem dissolving. He dove into the problem—actually, he stepped back and took a broader view of the system, which is really important when you want to get to the bottom of a problem.

The consultant found something really interesting, which made him propose a redesign of the system. During rush hour, there were more buses on the streets than there were bus stops. Conductors should get off the bus and stay at the bus stops! If they got off the bus and collected fares from people at the bus stops during rush hour, they could still signal to the driver that everyone was on board and ready to go. But more importantly, the passengers could get on the bus a lot faster. When the peaks were over, the conductors could work on-board again. The problem had been dissolved.

His proposition had another advantage. During peak hours, fewer conductors were needed because there were fewer stops than buses, which meant a huge savings in salaries for the company!

### A.4.1 Get to the bottom of the problem

In general, you want to solve the essential and novel problems that are at the core of your mission statement and domain, because that’s what your customers are paying
you for. You usually do that by applying a series of resolutions—previously applied and known solutions—to parts of the problem you’re solving. In addition, you want to dissolve any unnecessary or accidental problems.

For the parts of the system with technical debt where you don’t have to pay any interest, you can do nothing and absolve the problem. To pay back debt, you can apply technical solutions or resolutions to the problems. The Mikado Method is a tool for helping you solve your novel debt problems, while utilizing any resolutions you have. The solutions are typically a graph or a subgraph, whereas a resolution is often what you write in a single node, like breaking a circular dependency by using the Dependency Inversion Principle.

In contrast, the creation of technical debt is usually the kind of problem you’ll want to dissolve. To do that, you’ll likely have to change the system you’re working in, how work is arranged, how change requests enter the organization, how decisions are made, and so on. A shift in attitude is required, and you’ll need to step back and take a good hard look at the situation and the work system.

This can be difficult when you’re deeply involved in the details of the implementation. This is true not only because of the level of your involvement, but also because going to the root of the problem and dissolving technical debt requires a slightly different approach for each of the previously discussed quadrants. Figure A.6 identifies some phrases that suggest where the origin of the debt might be.

![Diagram of technical debt quadrants]

**Figure A.6** Phrases that hint where technical debt originates from
A.4.2 Third party—create defensible space

The disadvantage of being tangled up in API calls or tied to a specific solution to a problem becomes real when you’re forced into change after change due to changes in the underlying API. The trick when you work with third-party software is to minimize the collateral damage of any changes you can’t control, and identify situations that could lead to problems that might spread.

If you’re not careful in your use of third-party libraries, for example, the external dependencies can leak into your codebase, much like a wildfire spreads over a prairie or savanna. And much like a wildfire, third-party libraries can be hard to contain once they’ve started spreading. This wildfire is also fueled by the broken window principle. ORMs, 3D engines, or report tools can easily become the wildfires of a codebase.

Firefighters know that stopping a fire is a lot easier if the correct prevention techniques are implemented, such as education, careful handling of open fires, and defensible space. Defensible space acknowledges that a fire needs fuel. Cutting down corridors of trees in a forest, or requiring enough space between buildings in a suburb are examples of defensible space across which a fire can’t easily spread.

Abstracts or layers are the defensible space of a computer program. Eric Evans calls them anti-corruption layers in his book Domain-Driven Design (Pearson 2004). When you’re able to see the fire early on and use abstractions and layering properly, many of these problems will be dissolved, thus minimizing the effects of third-party API changes later on.

A.4.3 Bigwig—probe and prepare

The problems that come from when the bigwigs, or the markets, change their minds are very hard to dissolve, and instead you need to probe and prepare for this type of change. You need to do your market research, keep in touch with customers, listen to the news, and keep up to date with current political decisions to anticipate what will happen.

In general, regulatory and legal changes can be hard to anticipate. What you can do is make sure your organization is able to respond quickly to any change. This means letting the people doing the job make the decisions about how to do it, have spare work capacity (slack) in the organization, have automated validation and verification processes in place, and have as little work in process as possible at any given time. When you’re working with code, using the Mikado Method with frequent check-ins is a good way of keeping work in progress low.

When your decisions are heavily influenced or regulated by external or governmental instances, we recommend making those variation points easily configurable. Keep your options open by separating the logic that varies depending on external decisions from the logic that stems from internal decisions.

A.4.4 The boardroom—talk and learn

Changes to business models and great new ideas are probably the most natural and sane causes for change in a codebase. But when the program manager repeatedly makes decisions that cause large changes to your program, you have a problem.
If technical and non-technical people don’t have an understanding of each other’s work, friction will appear and the nimble solution will continue to be unidentified, or unimplemented. The non-techies need to get an understanding of how their decisions affect implementation details, and techies need to understand the relevant parts of the business domain to make good technical decisions.

To achieve this, non-techies and techies need to teach each other and learn from each other, to build the knowledge and the respect needed to create awesome products. This communication can’t be expected to be built on an occasional argument.

The easiest way to get started is for one group to invite the other to talk about both business and technical details, such as technical debt, with the goal of learning and understanding each other’s work and challenges. Once you’re speaking the same language, both sides are more likely to make the right decisions. This also helps mitigate the problem of creating technical policies that damage development.

A.4.5 Propeller hat—form an opinion and align

Awkward technical solutions, sloppy coding, absent or inadequate coding standards, and different opinions on what code should look like, or even what paradigms should be used, are also drivers of technical debt.

The road to better code starts with forming an opinion about what great solutions look like. This opinion shouldn’t be static, even though some people tend to believe so. It’s actually a work in progress; it’s always subject to improvement. By continuously and actively working on a team’s opinion, and aligning development and design ideas with that opinion, it’s easier to come to agreement on where to take the existing code, what parts to improve, and how to implement new things. Forming a team opinion takes some time and a lot of effort, but unless you talk about it, you’ll be running without direction, which is an even worse use of your time.

Looking at other people’s code or other paradigms for inspiration and ideas is an excellent way to broaden your perspective. You should look for the qualities you want in your own work, and then figure out what parts of your current opinion to replace.

A.5 The way out of technical debt

Sustainable change can’t be forced; it requires a willingness to change, and this goes beyond the software development team—it includes the whole organization. If you want different output from the system that created what you have today, you need to change it, which essentially means a change in behavior. You can’t expect different results from a system without a change.

Apart from different behavior, you also need discipline and integrity from the people who develop the software. Real improvements to a messy codebase can only be made by moving away from the current situation and then continuing in that direction for a long time. For this, the developers need support and commitment from management.

You also need to be patient. You can’t expect an organization to change over the course of a week. Rewiring a human brain for new habits and ways to do things normally
takes around a year with plenty of practice, and it’s easy to see that discussing and changing old habits should be pretty high on the agenda, continuously.

Making use of an effective process for the change helps a lot. This is where the Mikado Method can help in repaying the current debt and getting the product in shape at a competitive price.

Technical debt is a tricky thing. You can’t just put up a lot of money to pay the debt and hope that the problems are gone the next day. You do need financing for your activities, but primarily you must pay with sweat and hard work.

Technical debt isn’t limited to the code; it can manifest itself in anything from code to build scripts, folder structures, version control systems, tests, environments, operating systems, configuration files, and more. When we’re using the Mikado Method, we don’t limit the nodes to code. The nodes can be anything in the system that we need to take care of.

A.6 Summary

Just fixing problems as you run into them isn’t enough; that should be pretty clear by now. If you just fix problems, you’ll end up feeling like you’re Sisyphus, the ancient Greek king who, according to the mythology, was condemned to perpetually roll a immensely heavy boulder up a hill, only to have it roll down again before reaching the top, and having to roll it up again. Unless you change the system, addressing the way the work works, you’re doomed to the same fate as Sisyphus.

Changing the process of how software is developed will look different depending on a lot of things, such as the type of application you’re developing, the size of the team, and a lot of other parameters. It’s also important to remember that reworking, redesigning, and restructuring your system can never be avoided entirely, but examining where your changes are coming from can help you create strategies for minimizing the disruption. Try to mitigate your debt builders as early as possible, and pay back your loans with the highest interest first.

Try this

- What alarm bells are ringing for you in your current environment, and which are sounding most frequently?
- What alarms have you experienced in previous projects? What did you do? What should you have done differently?
- Can you think of times when you introduced technical debt into a system?
- What types of technical debt can you see in your current, or a previous, system? Can you find some that resulted from hitting a market window, unprofessionalism, ignorance, external events, or learning?
- What types of events caused your debt?
- What parts of your code are troublesome to read or change? Do you read that code often? Do you change it often?
- For a piece of code that has some debt, can you estimate how much less time you’d spend there if you could improve the code? What would that improvement constitute? What stops you from changing it?
The game “pick-up sticks” is a good metaphor for the Mikado Method. You eliminate “technical debt”—the legacy problems embedded in nearly every software system—by following a set of easy-to-implement rules. You carefully extract each intertwined dependency until you expose the central issue, without collapsing the project.

The Mikado Method presents a pragmatic process to plan and perform nontrivial technical improvements on an existing software system. The book helps you practice a step-by-step system for identifying the scope and nature of your technical debt, mapping the key dependencies, and determining a safe way to approach the “Mikado”—your goal. A natural by-product of this process is the Mikado Graph, a roadmap that reflects deep understanding of how your system works.

What’s Inside

- Understand your technical debt
- Surface the dependencies in legacy systems
- Isolate and resolve core concerns while creating minimal disruption
- Create a roadmap for your changes

This book builds on agile processes such as refactoring, TDD, and rapid feedback. It requires no special hardware or software and can be practiced by both small and large teams.

Ola Ellnestam and Daniel Brolund are developers, coaches, and team leaders. They developed the Mikado Method in response to years of experience resolving technical debt in complex legacy systems.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/TheMikadoMethod

“Enables you to deal with unavoidable complexities in manageable pieces.”
—From the Foreword by Tom Poppendieck

“An engaging and authoritative source of information and best practices that you need to succeed.”
—Alex Garrett
Hot Towel Consulting, LLC

“A great guide to safer refactoring.”
—Mark Sponsler, SintecMedia