# Beheading the Legacy Beast

## Relentless restructurings with The Mikado Method

Daniel Brolund
Ola Ellnestam

Foreword by Tom Poppendieck

# This is a DRAFT

*Saturday 18th February, 2012, 23:49*

This book is a draft that we provide to you, free of charge, for you to be able to learn about the Mikado Method to handle small and large refactorings.

We would be very happy if we could get feedback on your impression of the book. Such feedback can be sent to:

daniel.brolund@agical.com
ola.ellnestam@agical.com

or posted on the Google group:

http://groups.google.com/group/mikado-method

Please include this version for us to know what version we get feedback on:

**c549383356e6afbf1619424a5aeb8275ae9ba54d**

What we are looking for at the moment are things like:

- Things you don't understand or that are insufficiently explained
- Things that are missing
- Things that shouldn't be in there for one or another reason
- Things that are wrong

Your first impression is very important to us!

Please, also tell us a little bit about yourself and your familiarity with refactorings and software development in large. This helps us put your feedback into a context, and that will hopefully help us make this the best book possible.

Thank you!

*Daniel and Ola*

# Beheading the Legacy Beast: Relentless Restructurings with the Mikado Method

Daniel Brolund, Ola Ellnestam

Saturday 18th February, 2012

# Contents

# Foreword by Tom Poppendieck

Most software discussions, books, and articles seem to assume that development work starts with an empty codebase. The agile literature usually presumes that the detailed learning about what is needed to solve a problem for a stakeholder community can be discovered iteratively and incrementally. This might be a valid assumption if one is starting fresh. However, I seldom encounter teams with the opportunity of starting from scratch; there are always constraints. Most investment in software today involves modifications or extensions to existing applications and environments. Thus, in addition to discovering and implementing solutions to business problems or opportunities of the people we solve problems with; we have the constraint of fitting in to the environment created by earlier development teams. We need to discover and codify not just the applicable domain knowledge, policies, and organizational goals that drive our current development work, but also understand how the changes we make affect the existing application environment. Everyone who has done this kind of work knows that the information we need is only weakly represented in the documentation left behind by preceding teams. One has to look at the source code to get reliable information about the constraints it imposes upon additions and changes. If the source code includes an effective set of automated tests, we are in luck because tests illustrate the behavior required of the code by previous implementers to solve previous problems. If these tests pass when executed, we know that the code behaves as they expected. More often, automated tests were never cre-

ated and we are left with only the source code itself. The question then becomes how to learn what we need to know to avoid breaking previous work.

The barrier is sheer complexity. Analysis of the preexisting code has proven to be a weak tool for tackling this complexity. Much of the code we have to confront no longer communicates effectively to us the nuances we need to understand to avoid breaking it as we implement our changes, and it usually ends up taking far more time than we can afford. Therefore, after we exhaust our patience with analysis, we are forced to go ahead and take a chance at breaking the fragile existing code. What we do next is critical.

'Traditional' approaches have ended with a long, tedious, unpredictable test and fix period which often consumes the second and sometimes third 90% of our project schedule. Agile and lean thinking have taught us to test immediately, integrate continuously, and fix every problem discovered as soon as we discover it. This works well with well-structured code weakly dependent on its environment — provided we have an effective automated test suite. The teams who preceded us, however, have seldom had the perspective, skills, or inclination to leave us with such a foundation as they were driven mostly by the need to get-it-done (project scope focus) even at the expense of keep-it-clean (longer-term product lifecycle focus). So we find ourselves in a hole!

The first rule of holes is this: "When you find yourself in a hole, ***stop digging!***" So agile thinking suggests that for all the code you add to an application, use your refactoring, clean coding, and TDD practices to ensure that you do not make the situation worse. You probably have to add some higher-level automated functional tests to be able to do even this safely. Even then, complexity easily becomes overwhelming and quality, speed and cost all suffer. Managing this unavoidable complexity is what this book is about. Rather than permitting complexity to cascade, the Mikado Method enables you to discover what you need to know and address it in manageable pieces by maintaining the focus on always having a known good code base. Ultimately, this is the deepest core of Agile — Always ensure that the work you have done so far is correct.

While the form of the Mikado Method is simple, it ties closely to and greatly helps with the application of principles and practices we have

learned over the last decade. It addresses individual, pair, and team wide practices to reliably tackle changes in small steps. Ubiquitous language and the SOLID and DRY principles characteristic of clean coding practices guide our Mikado Goals and trees of prerequisites that structure our work. Mikado Graphs help keep our packaging, dependency, TDD, and refactoring work properly focused.

The initial focus of the Mikado Method is effectively dealing with a messy reality. The long term goal; however, is to understand the forces that drive organizations to create bad code in the first place so we can avoid creating more code our successors will have to struggle with unhappily.

When confronted with the normal levels of complexity in today's software, we cannot keep everything we need in our heads at once. People have discovered many ways of effectively working together in complex tasks starting with the scientific method and extending to the Theory of Constraints, empirical control systems, and pull/flow systems. The Mikado approach works with all these strategies to help us stay in control.

*Tom Poppendieck*

# Preface

As a codebase get large and complicated, as they often do, there usually comes a time when we want to improve portions of it to meet new functional requirements, new legal requirements, or a new business model. We may also just want to change it to make it more comprehensible. For small changes we can keep things in our head, but for larger ones the chances of getting lost in a jungle of dependencies, or on a sea of broken code, increases dramatically.

## What is the target code for this book?

When changes and improvements are mentioned in the same sentence as code, it is easy to think about bad code.

As we all know there are lots of names for code, especially bad code that isn't fit for purpose. *Legacy code* is one of the more popular ones, which literally means code someone (else) wrote and you are now responsible for it. Michael Feathers suggested it can be defined as *code without tests* [1], which has since then become the *de facto* definition.

Another name for bad code is *big ball of mud*, popularized by Brian Foote and Joseph Yoder in the paper *Big Ball of Mud* [35]. This paper describes a big ball of mud as: *"a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle."* Some synonyms for this type of code are *crap* or *a mess*, and we can think of quite a few more in our native language (Swedish).

When code like that needs to change and trouble appears, it is easy to

label the code *legacy*, *a big ball of mud*, *a mess*, *crap*, or just *impossible to work with*. We distance ourselves from the code and start lobbying for a rewrite, since changing the code is perceived to be too hard.

In addition to the big ball of mud systems, there are at least two more types of systems that need attention. One type is systems that look well organized, maybe even with high test coverage, but underneath the surface they are hard to work with or to extend. The other type is systems that look good and have served us well, but doesn't anymore.

This book is about *changing and improving code that is unfit for purpose*. It offers a way to regain control over a codebase. We call it **The Mikado Method**.

## ACKNOWLEDGEMENTS

There are a lot of people that have contributed to this book, directly or indirectly.

Technically, it builds on the work of Michael C. Feathers, Martin Fowler, Robert C. Martin, Kent Beck, Ward Cunningham and Joshua Kerievsky, to mention a few. The phrase *standing on the shoulders of giants* feels very appropriate when we think about how our ideas around this work came to us.

We would like to give a big thanks to our colleagues and friends with whom we have discussed the method and this book, which has given us many insights for improvements.

We would also like to thank Laurent Bossavit for noticing the similarities between our method and the pick-up sticks game, and thereby naming the method.

A special appreciation goes to the reviewers of the book. Tobias Anderberg, Alan Baljeu, Olle Dahlström, George Dinwiddie, John Eckhardt, Steve Eckhardt, Manne Fagerlind, Thomas Gustavsson, Jakub Holý, Anders Ivarsson, Colin Jack, Torbjörm Kalin, Jan Mattsson, Luca Minudel, Staffan Nöteberg, Joakim Ohlrogge, Tom Poppendieck, Robert Postill, Måns Sandström, David Sills, Jelena Vencl-Ohlrogge, Ted M. Young, and Stefan Östergaard.

A very special appreciation goes to Tom Poppendieck for writing the foreword, for promoting the book, and for his excellent review work. A very special appreciation also goes to Heidi Helfand for her fantastic and relentless editing of the book.

## ABOUT THE AUTHORS

**Daniel Brolund** is a software developer that always see things to improve, to the joy or grief of his fellow workers. He has successfully worked with global web sites deployed on hundreds of servers, desktop applications for just a few users, and on-line gaming applications just to mention a few. He has also done death-marches and a huge big-bang catastrophe.

**Ola Ellnestam** is a coach and mentor for both business and technical teams. He loves to combine technology, people and business, which is why he finds software development so interesting. He has developed complex computer systems within healt care, defense and online banking and he knows that software must be easy to use, extend and deploy in order to be worth developing.

More than anything else, he likes to share his findinds and knowledge with others because he believes that this is how new knowledge and insight is created.

# Introduction

## A COMMON SITUATION

In our profession, we have had to clean up messy code, refactor smaller and larger parts of systems and restructure complex architecture. We have even tried rewriting code from scratch, also known as *greenfield development*. But we have noticed that the field doesn't stay green very long. Let's face it; we're stuck with *brownfield development*, whether we like it or not.

Computer programs have to improve or they are doomed to a slow death. We, the developers, hold the fate of the code in our hands and we are the only ones that have the power to improve it. It is our responsibility to keep the code clean and fit for purpose. This means we have to be able to add code, improve our own code and the code of others.

So, how do we approach large improvement efforts, especially while the pressure is high because we must keep delivering new functionality once a month, or even more frequently than that? Also, we might have a huge system that is hard to comprehend, and its complexity makes it hard to keep all the necessary details in our heads at once.

What complicates software development even more is the fact that it is more or less the norm to develop software as a team. It is very seldom developed by one single individual. Working in groups makes distribution of the work possible but teamwork also adds another dimension of complexity and it doesn't take long to realize that the need to communicate necessary improvements within a team is essential for any

real progress. We cannot scale the change effort to a whole team if it is inside the head of a single person!

## THE GOAL: MORPHING A SYSTEM

Halting development of new functionality isn't popular in the business side of an organization. They want to see value coming out as money is put in. Therefore, software developers have to be able to cope with the current situation as future problems are encountered, solved, and sometimes avoided entirely.

*If we want to be successful software developers, we need to learn how to morph an existing system to a desired new shape.*

When we say *morph* we mean the necessary actions and steps that take us from one shape of the system to another without breaking things in between. The transitions between the different states need to be smooth and possibly add value themselves.

## THE MIKADO METHOD

The Mikado Method helps us visualize, plan and perform business-value-focused improvements over several iterations and increments of work, without ever having a broken codebase during the process. It enhances communication, collaboration and learning in software development teams. It also helps individuals and programming pairs stay on track while doing their day to day work. The framework that the Method provides can help the whole team to morph a system into a new desired shape.

The Method itself is straightforward and simple: after deciding upon a goal, we start to build a mental model of relevant dependencies and required changes by doing quick experiments.

Naïvely we try to accomplish our goal. When we cannot, we create a graph of prerequisites nodes. We use the information from our quick experiments to determine whether a change can or cannot be implemented right away. If a problem occurs we roll back to the original

state. By solving the prerequisites first, we work our way backwards and the created graph guides us towards our decided upon goal.

It is basically the same effective approach structured people have always used when attacking complex software system problems. It is goal driven, systematic, and it involves doing as little as we can get away with.

## A thinking tool

When we introduce the Mikado Method to people who develop software, we often see how they change the way they look at, approach and talk about large structural improvements. The perspective shifts from a very analytical view of problems, to a more practical approach, where the focus is on removing a minimum amount of obstacles at a time, in order to achieve real results.

The effect is more short, focused conversations that are about finding the changes that can be done without breaking the code instead of rigorous analyzing and guessing. Together we get more insight on how to deal with the unnecessary complexity of a software system and the method serves as a *thinking tool* when we solve difficult problems.

Even though we are becoming more and more seasoned developers ourselves, we still find it challenging to see where to start digging and where to take the code. The Mikado Method not only helps us find a starting point, it also shows us where to go and tells us when we are done.

## Characteristics

We think the Mikado Method:

- Fits nicely in an incremental process
- Is very lightweight (pen-and-paper/whiteboard)
- Increases visibility of the work

- Provides stability to the codebase while changing it
- Supports continuous deployments by finding a non-destructive change path
- Improves communication between people
- Enhances learning
- Aids reflection over the work done
- Leverages different competencies, abilities and knowledge
- Helps collaboration within a team
- Scales by enabling distribution of the workload over the team
- Is easy to use

## ABOUT THE CONTENTS OF THIS BOOK

This book is about getting codebases in better shape using the Mikado Method. We have tried to create a book that can be read from first to last page, and a normal developer should probably read the book in its entirety.

However, we have tried to structure the book using the *inverted pyramid* where we place the most important stuff first, then adding to that throughout the book. This way, a reader should be able to stop reading at any time, and still get the best possible return on time invested.

The initial chapters are hands-on recipes for how to get out of trouble and as the book progresses more and more details and aspects are added. This adds depth to the previous knowledge and helps you, the reader, to make the most out of the Mikado Method and your existing codebase.

Along the way you will get acquainted with *Jake*, a developer working with some messy code, and *Melinda*, a more seasoned developer that helps Jake with his ordeals.

## A description of the chapters

Below is a brief description of the chapters in the book.

***Ch.1 Kick start your improvements (p.1)***

In this chapter, we tell a story about the big failure that led us to the Mikado Method. After that comes the Mikado Method crash-course, a quick run-through recipe of the core of the method, along with some simple rules to follow. This chapter might just be enough to get you started and help you in your restructurings the Mikado way.

***Ch.2 An example in code (p.23)***

Here we apply the Method to a very small, but trouble-ridden, codebase. This is to tie the Method to real code, in case you feel the run-through in the first chapter was a bit too abstract.

***Ch.3 More about the Mikado Method (p.53)***

Here we explain the Method with a little bit more detail. We also describe the Mikado Method, its relations to other thinking models, and some principles we have found resonates well with the Method.

***Ch.4 Patterns when using The Mikado Method (p.75)***

This chapter describes some of the more common patterns we have come across when using the Mikado Method.

***Ch.5 Guidance for creating the nodes in the Mikado Graph (p.103)***

This chapter explains some of the more important principles of software design. Those principles are the *force-field* when working with refactorings and restructurings. We have had so much use for them in our work that we cannot omit them in this book.

***Ch.6 Application restructuring techniques (p.139)***

This chapter compares and puts the Mikado Method in relation with other common ways of changing your code.

***Ch.7 Organizing our work (p.153)***

There are a couple of different ways to organize your restructuring effort. This chapter presents a couple such ways, to give you an idea of the options you have when applying the Mikado Method.

***Ch.8 Setting the stage for improvements (p.169)***

Here we have gathered a number of preparations that have served us before dealing with changes and the Mikado Method. These are preparations both for the individual, the team and the organization.

***Ch.9 Technical debt (p.199)***

Messy code is often referred to as having technical debt. In this chapter we describe technical debt, and what causes it.

***Ch.10 Origins of change (p.223)***

The real origin of perceived code problems is not always from within the development team. This chapter tries to find the root cause of the problems.

## Reading suggestions by role or interest

If you don't have the time to read everything, or you have specific interests, you can chose from the reading suggestions below.

If you are a **developer in an emergency situation**, read *Ch.1 Kick start your improvements (p.1)* and *Ch.2 An example in code (p.23)*. That should give you enough information to perform large scale and relatively low risk improvements to your codebase.

Also for **developers in trouble**, we present more details and advanced topics around refactorings in *Ch.3 More about the Mikado Method (p.53)*, *Ch.4 Patterns when using The Mikado Method (p.75)*, *Ch.5 Guidance for creating the nodes in the Mikado Graph (p.103)*, *Ch.6 Application restructuring techniques (p.139)*, *Ch.7 Organizing our work (p.153)*, and *Ch.8 Setting the stage for improvements (p.169)*.

For **the philosopher** to get familiar with the foundation of the Mikado Method, read *Ch.1 Kick start your improvements (p.1)*, *Ch.3 More about the Mikado Method (p.53)* and *Ch.6 Application restructuring techniques (p.139)*.

If you are **a person managing people** in some way and you want to learn more about how to deal with a messy codebase, read *Ch.1 Kick start your improvements (p.1)* and possibly *Ch.3 More about the Mikado*

*Method (p.53)* to get an idea of what the method is about. Then read *Ch.7 Organizing our work (p.153)*, *Ch.8 Setting the stage for improvements (p.169)*, *Ch.9 Technical debt (p.199)*, and *Ch.10 Origins of change (p.223)* to get a hang of the context of changing an application.

# Chapter 1

# Kick start your improvements

At Pasta Software, a company somewhere in the world:

Jake: Now they've done it.

Melinda: What?

Jake: ... this time they've really done it?

Melinda: Jaaake ...

Jake: Melinda, I didn't hear you come in. What did you say?

Melinda: You were mumbling something about someone ...

Jake: Yeah, it's sales again, they've sold the Mastercrüpt application to a new company.

Melinda: Yeah, I heard something about that, that's great. What do you mean, they've done it?

Jake: We can't ship Mastercrüpt to another client. It uses an algorithm that we're not allowed to distribute.

Melinda: So, why don't you just exclude that from the application and make another implementation?

Jake: Ha! Do you have any idea on how difficult that is. The system is a big ball of mud and we've tried to restructure things but we keep getting nowhere.

Melinda: I know something that I think you could make good use of.
Jake: Really?
Melinda: Yes, the Mikado Method. It's a systematic way of handling large structural improvements.
Jake: Tell me more!
Melinda: Sure!

## 1.1 Software dependencies

Every part of a software program has dependencies. In *Fig. 1.1 (p.2)*, one aspect is illustrated in a very generic way. The application depends on the development environment, the development environment depends on the operating system, and the operating system depends on the hardware through its drivers.

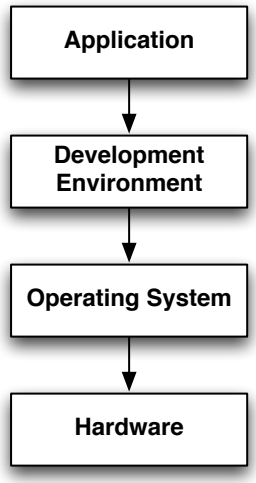


Figure 1.1: Basic dependencies

Within each of these blocks, there are hundreds and thousands of parts that in turn have dependencies between them. For any system at some scale, these dependencies create restrictions that prevent changing that part of the system in exactly the desired way. This means that in order to do what is really desired, other changes must be performed first.

The key to changing any system is to first change the restrictions enough to make the desired changes possible. On a high level it is easy to draw nice pictures of a system. But when getting down to the details, there are dependencies on classes and objects, methods, parameters, external systems, networks, structured and unstructured data and more, see *Fig. 1.2 (p.3)*.

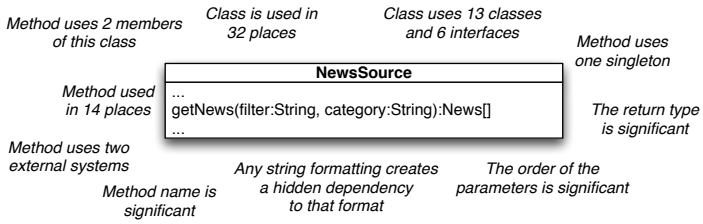


Figure 1.2: There are a lot of different dependencies!

As code and functionality is added to a system, the number of dependencies increase and this solidifies the system which of course decreases flexibility. When we experience rigid systems, it is often these kinds of dependencies that create restrictions and make the system difficult to understand and change. At the same time, it is the very same dependencies that actually let the system do anything useful.

Let us see how those value adding dependencies can get us in some serious trouble.

## 1.2 The Mikado Method - a product of failure

The Mikado Method, like many other discoveries or insights, is the product of a failure.

We would like to tell the story about the first steps and events that led to the discovery, or creation, of the Mikado Method.

Our insights and experiences come from a fairly large restructuring of a software program, which should have been performed as many small steps but were not. Our story is rather the opposite of how to perform such work, and these are the lessons we learned from that adventure.

### 1.2.1 Knee deep in a mess

Melinda remembers:

A couple of years ago, I worked on a project where the codebase had gradually turned into a mess. We made a few slips here and there. Some of us cut corners in order to *save time*, and we also skipped necessary design discussions.

The code was fragile and complex. It consisted of global variables and singletons gluing modules to each other in intricate ways. Circular dependencies, deep and fragile inheritance hierarchies and more caused us problems every day and slowed us down immensely. Pretty much everything we wanted to do took more time than we expected and finally the task of adding very simple functionality could take well over a week.

### 1.2.2 A new client

One day, the company who owned the product we worked with landed a new contract. Fortunately for the team developing the software, the new client needed an almost identical system as the one they had. Just a few minor changes and maybe one or two bigger ones. The big sale was already made, and time wasn't really on our side. We basically

had two options and both would mean extra work, but in very different ways.

We learned early on that there were certain parts of the code that couldn't be shared between our clients. Under no circumstances could one client see the other's secret parts. We had to separate the sensitive business logic into different libraries in order to choose what parts should be delivered to each client respectively.

One way of solving that dilemma was to copy most parts of the codebase to a new project, leaving out the first client's sensitive logic and to modify the new client's code there. That would have solved our immediate problems; no sharing of sensitive code between the two customers.

On one hand, we would be able to deliver really fast as we wouldn't have to think about how to structure the code to minimize duplication. On the other hand, that would mean we would have had to maintain almost twice as much code, including our previously poorly structured code, bugs and all. We would have had to maintain two branches of messy code slowly drifting apart. For every future feature or bug, we would have to analyze both codebases in order to implement a change. As we were already performing badly, more work wasn't very appealing. Duplicating the codebase would mean a lot of extra work, not to mention that it is a very error prone approach. Add to that the fact that it would have been a real pain to work with the code on a day to day basis.

Our alternative was a large up-front restructuring where we would prepare the system for several future customers first and then add our new customer when the system allowed it. This would mean a bigger up front investment and likely a longer time before we could deliver. On the other hand, there would be a lower cost for maintenance. Adding features would be easier, as well as fixing current and future defects.

This was not an easy decision and we argued well over a week before we reached consensus. Our many heated discussions resulted in an agreement to restructure the codebase first – just give us a month, and we would be done.

A colleague and I started working, and guess what?! The work was more complex than we initially thought. Our approach got us into trouble of enormous proportions ...

### 1.2.3 Beheading the beast

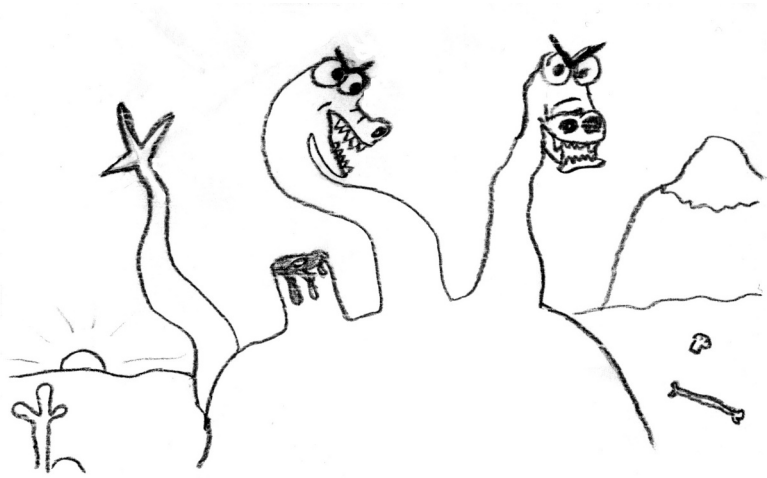

Figure 1.3: A Software Hydra

Melinda remembers more ...

We thought we had a decent plan and a good grip of the situation at hand. Our approach was simple enough and we thought we could get away with just moving some classes to new packages, fix the compilation errors and then add the changes to the versioning system.

As we tried to do the changes we needed in order to split the system in two and separate the sensitive information from the more general system, we ran into trouble. It didn't take very long before we were doing changes to the very core of the system and this was not what we had in mind. Our head-first style proved to be more complicated than we hoped for.

After each and every change we got about 20 compiler errors and immediately we dug into those, patching and adapting to a myriad of unanticipated new errors. When we were just about done with those, the compiler would move one step further, only to reveal a new round

of errors. Fixing those led to more restructurings and more errors. We were in a constant state of broken code, and we had nothing solid to rely on.

Hours turned into days while we valiantly stayed on our course towards more flexible code and the new structure. Day after day we would report our progress in our daily meeting, simply stating, "We're close now. Just a few more hours before we're done. We're probably going to finish this today or tomorrow."

We didn't want to fall behind too much so every day we decided to merge our code with the changes from the others on our team, which was problematic and took a lot of time. The fact that the system didn't compile meant we had no help from neither the compiler nor the tests and we couldn't see if we actually had done the merge correctly. We went so far as to tell people to not change certain parts of the code and even locked certain files in the versioning system, effectively blocking our teammates from changing those parts.

More than two weeks went by and we had well over a thousand files checked out and more than two hundred compiler errors. We couldnt get the code to compile and we had no idea whether the high level system acceptance tests passed or not. It was impossible to tell since we hadn't been able to run any tests for quite a while.

It was time to face the hard facts. We would never make it. We were trying to behead the *Software Hydra*. For every head we cut off, two more grew out.

One day we swallowed our pride and made the tough decision to revert everything we had done so far. We still remember the day when we faced the team during one of our daily meetings and announced that we had wasted several weeks worth of work. At least it felt like that at the time.

### 1.2.4 The dawn of insight

*There is no failure but the failure to learn from failure*

*Kevin Everett FitzMaurice*

Reverting the code would prove to be a defining moment in our software career, in two ways. First, we had learned that it is really never too late to turn back. Second, we thought we had an idea of how we could approach this problem, and other problems like it.

We stated that in order to get the system in shape, we still needed to do almost all of the things we had already done, but we needed to do them in "the opposite order" starting with the small and simple things which would then enable us to do the next change, and so on.

## 1.3 The Mikado Game

> *Mikado is a pick-up sticks game originating in Europe. In 1936 it was brought from Hungary to the USA and was mostly called pick-up sticks. This term is not very specific in respect to existing stick game variations. Probably the "Mikado" name was not used because it was a brand name of a game producer. The game got its name from the highest scoring (blue) stick "Mikado" (Emperor of Japan). The buddhistic Chien Tung also contains a stick called "emperor".*
>
> *From Wikipedia, February 2011.*

The Mikado Game can be played by two or more players. The game is played with a bunch of sticks that are much like dried spaghetti sticks. The game starts by bundling all the sticks in one hand and as they are held in a vertical orientation, over a flat surface, they are dropped. The sticks hit the surface and forms a pile of some kind. In the middle of the pile lies the differently colored, higher scoring stick - *The Mikado*.

The goal is to pick up as many sticks as possible to score points, preferably the Mikado stick since it is the highest scoring stick. If a player is unable to pick up a stick without moving another one, the stick is put back and the turn goes to the next player. While a player picks up sticks without moving other sticks, he or she may continue.

The trick is to pick up the easy sticks first, the sticks that have no other sticks on top of them. Eventually, by using that strategy, the Mikado stick can be picked up, and the person doing so is likely to win the game.

## 1.4 Code changes and the Mikado Game

The similarities between the Mikado Game and restructuring software are striking. There are loads of dependencies and structural complexity in code which has to be taken into account. These dependencies have to be navigated and changed with care until there is an opportunity to do something without breaking stuff.

Just how the other sticks are in the way of the Mikado stick in the Mikado Game, a sequence of changes must be made before the goal of a structural code change is accomplished. In *Fig. 1.4 (p.9)*, there is a very simple game where the red and blue must be picked up before getting the Mikado stick.

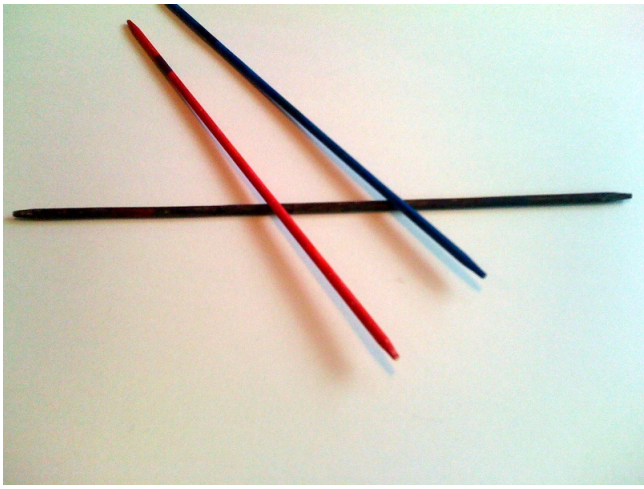

Figure 1.4: A simple pick-up sticks game (The Mikado stick and two sticks on top of it)

In software development, the "pile of sticks" are almost infinite, so we have to settle for one stick at a time and consider things done when that particular stick is removed.

The Mikado Method is about *systematically* working towards that stick.

So, how do we do that?

## 1.5 Our failure as a Mikado Game

We had heedlessly thrown ourselves into an improvement and change frenzy, modifying parts of the system that other parts depended on. Every change left us with an even more stirred up system, with more and more compilation errors.

*Using the Mikado metaphor: The highest scoring stick was at the bottom of our the pile and we had reached for it as the first thing we did, causing sticks to fly all over the place.*

We realized that we needed to do pretty much everything we already had done, but in the reversed dependency order. It was now easy to see that, and if we instead followed those dependencies, or prerequisites, until we reached a change that had no prerequisites, we had something we could actually do. We call such a step a *leaf*. These leaves are the starting points for any change since they can be done without stirring up the system with, for instance, compilation errors or test failures.

*We had to figure out a sequence in which we could pick-up sticks that enabled us to get to the Mikado without moving the wrong sticks along the way.*

Instead of bulling ahead when we found errors, we added the steps on a whiteboard, and drew a line to connect them with the current step. Then we realized we had to revert the code in order get to a known state before trying the next level of steps. Perform one step at a time, find the errors, note the new steps as dependencies, revert, and then find the natural next step.

We drew the dependencies as a dependency graph and the achievable changes were the leaves in that graph, like in the very simple graph in *Fig. 1.5 (p.11)*. By performing all the leaf changes, we fulfilled the prerequisites of the other changes. Those changes had now themselves turned into leaves, ready for execution.

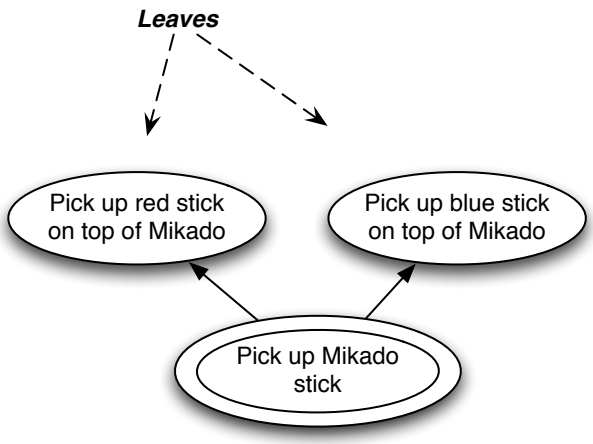


Figure 1.5: A pick-up sticks dependency graph for the simple game

*This is like picking up the top-most sticks from the Mikado pile and by doing so, making new sticks available and easy-to-pick.*

In some cases, performing a leaf resulted in new errors, and new prerequisite leaves. In some cases, we could actually just prune the leaves as they didnt result in any errors. Eventually, we pruned more leaves than we added, and after a while we were able to accomplish our goal.

*Eventually, we were able to pick up the Mikado.*

The Mikado Game continues until all sticks are picked up, but in software development there are usually more sticks than we have time for. In order to stay focused, we need to select one of the sticks as our goal and stop there.

So, how do we change a codebase without having the sticks flying all over the place?

## 1.6 THE MIKADO METHOD RECIPE

Code is not a piece of art. It is supposed to do a job. If the code does not stand in our way for other changes, or if we don't need to understand it, we just leave it be. On the other hand, if it needs to be changed, we need a way to resolve the problems that come with those changes.

Now it's time to introduce the Mikado Method recipe which we use to tackle problematic codebases. We start by describing the flow of the work and the rules for making changes. After that follows a chapter with a more concrete examples which include code and shows how the process is used in a more realistic situation.

### 1.6.1 Step 1: Draw the original Mikado Goal

If the mess affects our work, the best place to start is with a task, or a user story. This time we choose a task that needs to be accomplished, but is difficult to perform because of the mess. This task will be our original goal, and we call it the *Mikado Goal*.

We draw the Mikado Goal on a piece of paper, with a double circle around it to mark it as the original goal, like in *Fig. 1.6 (p.12)*. Instead of writing "Mikado Goal" as is we did in the figure, we would normally write a short description of an explicit goal for our application or system.


Figure 1.6: Start with the Mikado Goal

The double circle is there to indicate where it all starts. This goal will be our focus for the moment, and anything we do from now on is to make this goal happen.

### 1.6.2 Step 2: Implement the goal naïvely

Naïvely trying to implement the goal right away is actually the easiest way to see what obstacles are in the way. We simply try to "pick up the stick". The restrictions or dependencies of our codebase will then immediately be made visible by the compiler or our tests.

Sometimes, we must also analyze the situation first in order to realize what is stopping us. This is not wrong, but it is risky. If we *only* analyze the situation we can easily spend several hours on something that a change followed by a compilation of the code, or a test run, can tell us in a much shorter time.

### 1.6.3 Step 3: Find any errors

Is there anything that is stopping us? Are there compiler errors? Are there tests that don't run, or other obvious problems? All these errors have their origin in the dependencies that restrict us, and they are the reason we are having problems reaching our goal.

If we don't have any errors, we proceed with section *Step 8: Check in if there are no errors (p.16)* below.

An example of an error could be a compiler message which indicates that there are too few arguments to method *X*, probably due to the newly made change. Another common type of problem is runtime exceptions and errors such as null-pointer exceptions. Since these problems are hard to find by just reading the code, an automated test suite is helpful. This test suite minimizes the effort and delay in finding the problems.

### 1.6.4 Step 4: Come up with immediate solutions to the errors

All errors found need to be taken care of in order to reach the goal. We now come up with *immediate solutions* to the errors that will make the goal implementation possible. These immediate solutions will be the links that will solve the problems with the dependencies.

Once again, we don't want to overanalyze the situation and predict where it leads us. We want to apply the solution and see if it actually solves our problem or if the solution hints about new difficulties. Quite often, solutions have underlying restrictions or dependencies that need to be taken care of, which are handled in the next iteration of the process.

For the above example, an immediate solution could be *Add parameter at all places where method X is called*, or for the null pointer, *Initialize null field in method Y*.

### 1.6.5 Step 5: Draw the immediate solutions as new prerequisites

We note those solutions on the paper with arrows from the goal to the newly found solutions, like in *Fig. 1.7 (p.14)*. These solutions are prerequisites to our original goal.

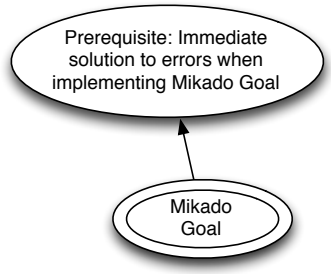


Figure 1.7: The immediate solutions, or prerequisites

A single solution can sometimes take care of hundreds or more of the errors we have. In that case, we only draw one prerequisite in the graph for all of the problems.

As we now start to build knowledge about the system and the dependencies that stand in the way of making our changes, we also find ways

to resolve our dependency problems, one by one.

When we cannot come up with a solution to a problem, we write something like *"Solve the errors with the missing arguments to method X"*. This works as a placeholder and jogs our memory when we iterate over the nodes in our graph in order to find a solution.

### 1.6.6 Step 6: Revert the code to the initial state if there were errors

*When there are errors, we always roll back all of our changes!* This is extremely important! Editing code in an unknown state is very error prone and before we start editing we revert to the last known, working state. Repeatability and predictability trump activity. Hence, roll back!

Sometimes this is perceived as losing all the work but in fact it's not. We have generated loads of information and have drawn all the things we need in order to get back to this state if we would like to do that. We could make a version control system patch and reapply it later on when the prerequisites are in place, but we must get out of the broken state right about now.

### 1.6.7 Step 7: For each of the immediate solutions, repeat the process

For each of the prerequisites, and *one at a time*, we repeat the steps above starting from *Implement the goal naïvely (p.13)*. This means that for each of the prerequisites, we start with a clean canvas and try to implement it naïvely. We find the errors (if any), come up with solutions to the errors, note them as prerequisites, revert the changes, and then continue with the next goal. This will build a graph, possibly resembling the outline of *Fig. 1.8 (p.16)*.

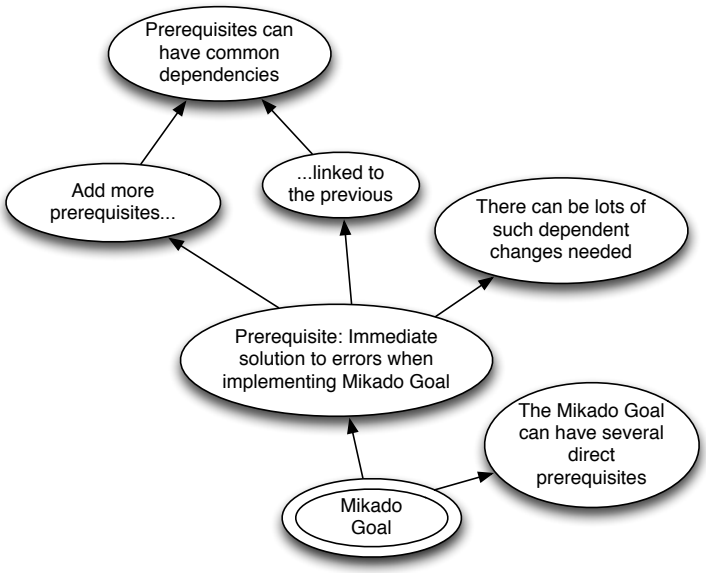


Figure 1.8: Repeat for each prerequisite to build the knowledge about the dependencies

### 1.6.8 Step 8: Check in if there are no errors

When we don't find any additional errors during the implementation of a prerequisite, we have come across a change that had no dependencies. This is probably a perfect time to commit code to the main code repository. In general, we commit if the code compiles, the tests run, the product is all good, and the changes we have made make sense to check in. If it doesn't quite make sense, look at the graph and see if it should be accompanied by a few more changes to make a sensible commit.

We usually note a checkin on the paper by checking off that node, or the nodes, like in *Fig. 1.9 (p.17)*.

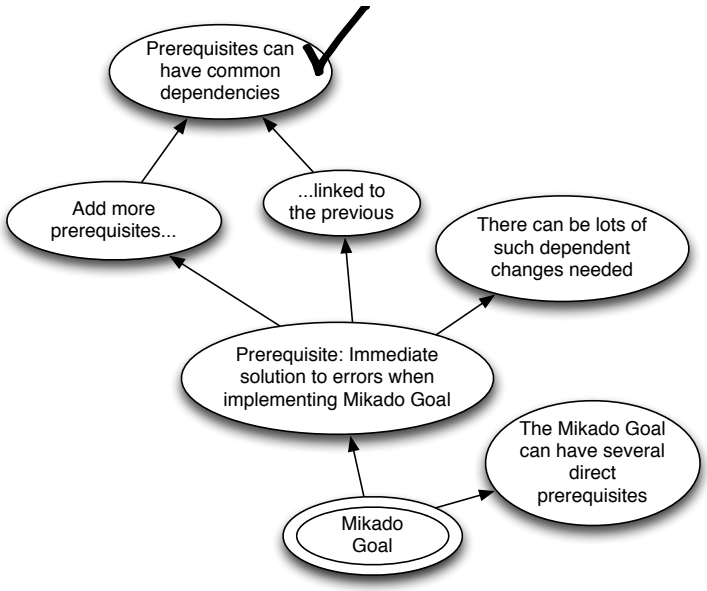


Figure 1.9: Check off the nodes as they are checked in

Now we continue by selecting a new prerequisite to work with as the next iteration, and we repeat the process from step 2.

### 1.6.9 Step 9: If the Mikado Goal is met, we are done!

When we have finished and checked in all of the prerequisites and the Mikado Goal, we are done. We pat ourselves, or our programming partner, or our team, on the back. We take a moment to reflect on what we have just accomplished. Then we realize that now is probably a great time for some celebration.

### 1.6.10 The Mikado Method in a picture

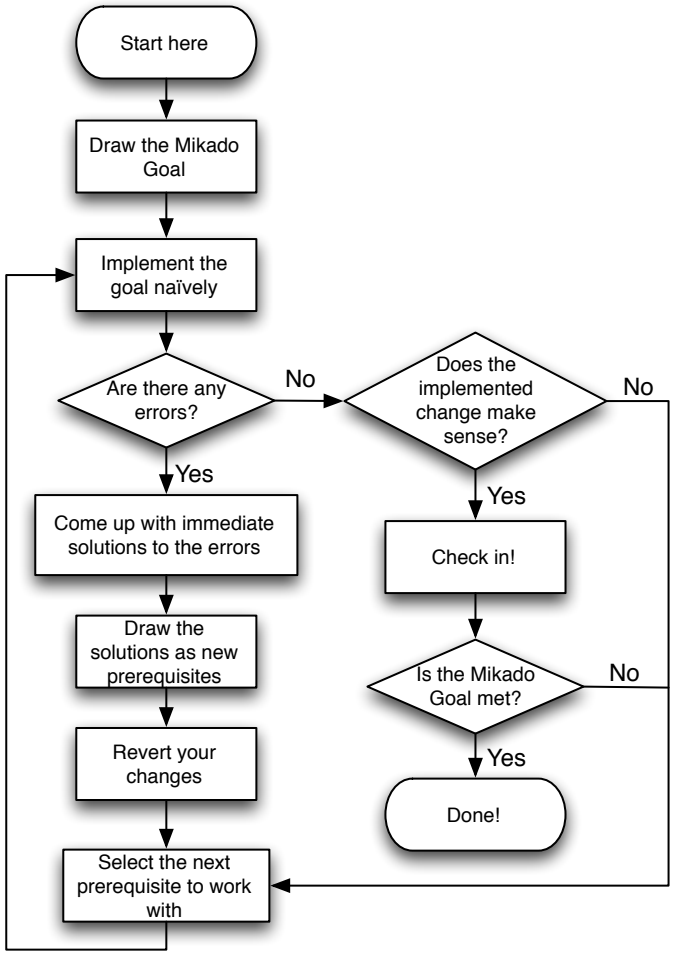


Figure 1.10: The Mikado Method

This is the core of how we find dependencies that create restrictions when we engage in larger restructurings of codebases. *Fig. 1.10 (p.18)* shows what it looks like in a more compound format.

We call the process *The Mikado Method* and the graph is a *Mikado Graph*. It is the basis of how we recommend performing changes to a system that is too large for 'analyze then edit', which is basically any production system in the world.

## 1.7 The Mikado Method rules

*Rules* is a strong word, but there are some core things to keep in mind when using the Mikado Method, without which the method falls fairly flat. When we boil it down a bit further, we find the rules, described in the following sections.

### 1.7.1 Write down the goals

For complex problems, we must aid our memory in order to remember what to do. Even though we all would like to be the person being able to juggle hundreds of things without effort, most of us can only keep track of about a handful of things at a time.

In addition, a goal gets much more real as we write it down on a piece of paper or a whiteboard. It works as an external commitment, and a commitment to ourselves, on what to do. If we cannot formulate and write down a goal and stand for it, maybe we should really be doing something else.

### 1.7.2 Seek things to try

To expand the graph, we use the Naïve Approach to quickly find changes that can give us feedback in the form of test cases that don't pass or non-compiling code. Both are very useful and help us find the next step.

We prefer the rapid feedback from compilers or tests over analysis, which is why we act as naïvely as we do. It's our way of avoiding analysis paralysis, a common but equally useless state that's best described as a strategy to avoid blame.

So instead of analysis we more often go for the simplest possible thing, and often we are surprised over our findings. Our mucking around is normally fruitful and produces results far quicker than hours of analysis.

When our naïvety doesn't seem to contribute to the expected progress, then we stop for a minute and analyze the situation. The initial efforts and attempts still pay good interest though as the information we've discovered is most useful when the analysis start.

### 1.7.3 Back out broken code

The most fundamental rule of the Mikado Method and sensible software development as a whole is to limit changes and additions to situations where the codebase is in a working and known state.

Making improvements and additions to code that is broken is simply asking for trouble – we don't do that.

There are, however, circumstances when code needs to change, perhaps to get it into a test harness, and the only reliable thing we have that supports us is the compiler. In those cases we always go from one known state to another, preferably using automated refactorings.

When we use the Naïve Approach to explore, we can do pretty much anything we want without being scared of making mistakes, as long as we restore the code to its previously working state.

### 1.7.4 Fix the leaves first

Eventually, we will find things we can do without introducing additional errors, but we have to be patient in getting there. We *Write down the goals (p.19)* and *Seek things to try (p.19)* to see where the application breaks. Then we *Write down the goals (p.19)* again using what we've

learned from our latest experiments which further expands the graph. After that we *Back out broken code (p.20)* and fix the leaves in the graph first. If we only have one single goal, this is of course our first leaf.

## 1.8 Summary

| | |
|---|---|
| Melinda: | So, that was the Mikado Method recipe crash-course. How do you feel? |
| Jake: | Hopeful! I know what my goal is, and I have a vague idea of how to naïvely implement it to see where it breaks. |
| Melinda: | Good! That is all you need to get started. |
| Jake: | It is? |
| Melinda: | Yes. The Method is all about that. |
| Jake: | That is simple! |
| Melinda: | Yep, the rules are very simple. Then you need to decide on how to implement the prerequisites, which is where your skill comes in. |
| Jake: | I knew there was more to it... |
| Melinda: | Don't worry. You will never be worse off than you would have been otherwise, more likely better. |
| Jake: | Sounds good ... |
| Melinda: | You've actually learned a whole new way of applying changes to your software. That is awesome! |

## 1.9 Try this

- What parts of your code do you find irritating? Make a short list.
- What goals exist in your code and your business? Make a short list.
- Start with one of the goals and figure out its prerequisites by changing your code using the method described. Do the irritating parts show up in the graph?

- For one or more of the things you want to change, start implementing the change and build your graph using the Naïve approach. Can you find any leaves?
- Do you have any failures that you can learn from?

# Chapter 2

# An example in code

| | |
|---|---|
| Jake: | Melinda, I have this problematic code and I need to add new functionality for a client. Can you help me? |
| Melinda: | I would love to. Do you use version control for your code? |
| Jake: | Yes? |
| Melinda: | Great, we'll use that. Have you ever done small, atomic, refactorings, like the "Move method" or "Extract interface"? |
| Jake: | Yes, a little. |
| Melinda: | Good, we will probably use those as well. |
| Jake: | So, where do we begin? |
| Melinda: | Well, what is your goal? |
| Jake: | Goal? |
| Melinda: | Let me rephrase that. How do you know when you're done? |
| Jake: | When I have a deliverable for the new client. |
| Melinda: | Sweet, that will be our Mikado Goal and we'll work from there. |
| Jake: | Just like that? |
| Melinda: | Just like that. |

## 2.1 New business for Pasta Software.

*Pasta Software* is a small family-owned company. By good luck and coincidence, their pride and joy MasterCrüpt (TM), has been sold to a new customer.

But now, problems arise. The very, very secret obfuscation algorithm in the application cannot be exposed between the old customer Gargantua Inc and the new customer, Stranger Eons Ltd.

Without really knowing it, Pasta Software has just put themselves into technical debt by changing their business model, i.e getting a customer with slightly different needs. They did not see this coming and the code isn't flexible enough to offer an easy way out. The design has to change so that confidential information doesn't leak between their customers. That would be the end of their business.

> *It is virtually impossible to tell how a system will evolve in advance. Trying to anticipate all future changes to a system and make the code flexible enough to handle them only makes the codebase bloated. In fact, that kind of over-design makes the code more complex and will likely become a problem in itself.*

### 2.1.1 The codebase

The code at Pasta Software is written in Java. This is an overview of the classes in the system:

```
package mastercrupt;

import static org.junit.Assert.assertEquals;
import mastercrupt.UI;

import org.junit.Test;
public class AcceptanceTest {
  @Test
  public void testLeeting() throws Exception {
    UI ui = new UI();
    assertEquals("Leeted: S3cr3t", ui.leetMessage("Secret"));
  }
}
```

Listing 2.1: The AcceptanceTest class

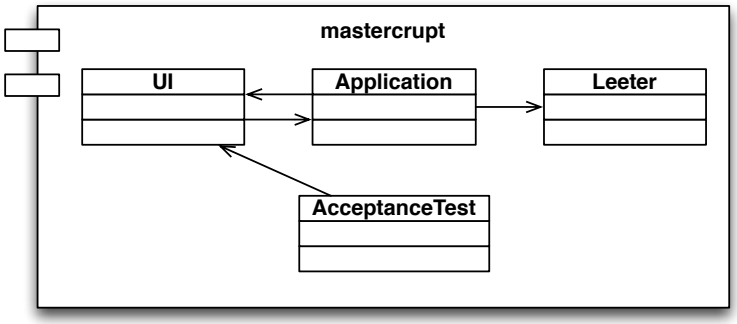


Figure 2.1: Class diagram of the existing system

Lets start with the test case, as in *Listing 2.1(p.25)*.

This is the acceptance test for the application as of today. It simply creates a `UI` instance and uses that to obfuscate a message. The algorithm used for obfuscation is a *leetspeak* algorithm.

> ***Leetspeak*** *is the way hackers and crackers avoided text filters on Bulletin Board Systems (BBS) in the eighties by re-*

```
package mastercrupt;

public class UI {
    Application application = new Application();
    private String leeted;

    public String leetMessage(String unLeeted) {
        application.leet(unLeeted, this);
        return "Leeted: " + leeted;
    }

    public void setLeeted(String leeted) {
        this.leeted = leeted;
    }
}
```

Listing 2.2: The UI class

> *placing alphabetic characters with non-alphabetic, but resembling, characters. For instance, in one leet dialect 'leet' is translated to 'l33t', where '3' is the mirrored capital 'E'.*

The `UI` class in *Listing 2.2(p.26)* creates an `Application` instance.

The `Application` instance is called with the string to leet and a reference to this `UI` class. The `Application` instance, see *Listing 2.3(p.27)*, is supposed to call back to `setLeeted(String leeted)` to set the leeted `String` value.

The `Application` class uses the `Leeter` class, see *Listing 2.4(p.27)*, to leet the given string and calls back to the provided `UI` instance with the leeted value. The `Application` class also contains the main method for the application.

The `Leeter` simply performs the leeting of the incoming message by substituting the character 'e' with the character '3'.

That is all the code there is in this application. As we can see, it only takes a few classes to create a mess. For instance there's a circular dependency between `UI` and `Application`, not to mention the mysterious callbacks.

In reality, codebases are a lot bigger and more complex by nature, so the ways of creating a mess are almost unlimited. The principles for the

```
package mastercrupt;

public class Application {
    public void leet(String string, UI ui) {
        ui.setLeeted(Leeter.leet(string));
    }
    public static void main(String[] args) {
        UI ui = new UI();
        System.out.println(ui.leetMessage(args[0]));
    }
}
```

Listing 2.3: The Application class

```
package mastercrupt;

public class Leeter {
        public static String leet(String message) {
                return message.replace('e', '3');
        }
}
```

Listing 2.4: The Leeter class

Mikado Method can still be applied though, no matter how big or small the codebase is. The method serves as a guide and helps to identify the critical change path in order to be able to deliver. Lets see what we need to do with this particular piece of code in order to create a new deliverable for Stranger Eons Ltd.

### 2.1.2 Starting out

Before making any changes to the code, we make sure that everything works, we *Back out broken code (p.20)*. The code compiles, all the existing tests run and there are no checked out files nor uncommitted changes. We start from a clean state. We could also label/tag the current state in the Version Control System (VCS) with *"Before the new client"*, or something similar, to be able to return to it later if we need to.

Now, our goal is to create a system that can be delivered to Stranger Eons Ltd. We will use the term 'New deliverable for Stranger Eons Ltd', and we *Write down the goals (p.19)* as in *Fig. 2.2 (p.28)*.

***Mikado Goal:*** Mikado Goal: new deliverable for Stranger Eons Ltd


Figure 2.2: New deliverable for Stranger Eons Ltd

In the Java world, that means we will probably have a separate project source root from which we then create a JAR-file for Stranger Eons Ltd. We add this as a prerequisite to our business goal, the root of the Mikado Graph, as in *Fig. 2.3 (p.29)*.

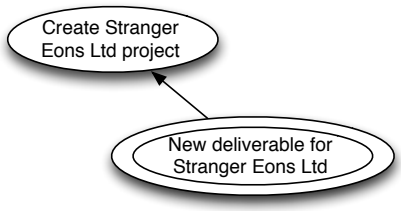


Figure 2.3: Create new project

Indeed, we analyze the situation to find out what we need to do, which is in contrast to the Naïve Approach. This is OK, because we *Seek things to try (p.19)*, where actual consequences of our changes tell us what steps we need to take next. Hence, we strive to make changes that will give us that feedback, as soon as possible.

***Step***: Create a project for Stranger Eons Ltd

Now we *Fix the leaves first (p.20)* and the Naive Approach will come in handy. In the naïve spirit, we just create the new project to see what happens. Luckily, we can do that without any problem.

In *Ch.1 Kick start your improvements (p.1)*, the rule says 'we check in to the main development branch if everything works as it should *and the changes make sense*'. In this case, the newly created project makes little sense to check in. In other cases, a created project might make a lot of sense.

So, we are back at the Mikado Goal. What next? Once again, we want to *Seek things to try (p.19)*. Driving development using Acceptance Tests or Customer Tests is something we prefer to do, and tests often give us real feedback about the next step. Therefore, we choose to create a test for Stranger Eons Ltd in the new project, a test which will look much like the one we have already for Gargantua Inc.

***Step***: Create test case

If we need to roll back the changes, we want to remember what we need to do so we continue to *Write down the goals (p.19)*. This step is also

```
...
    @Test
    public void testLeeting() throws Exception {
        UI ui = new UI();
        assertEquals("Leeted: 5ecret", ui.leetMessage("Secret"));
    }
...
```

Listing 2.5: The testcase for Stranger Eons Ltd.

added to the Mikado Graph, as in *Fig. 2.4 (p.30)*.

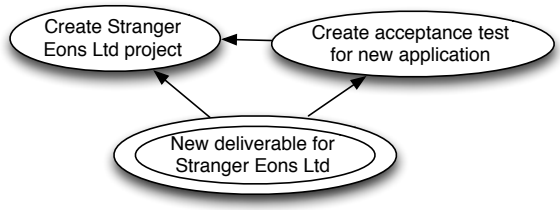


Figure 2.4: Create acceptance test case for new app

Still in the name of naïvety and since we have the project prerequisite in place, we create a new test whose only real difference from the one in the Gargantua Inc project is that this one expects a different return value, as in *Listing 2.5(p.30)*.

### 2.1.3 Dependency problems

But *now*, problems arise as we get a compilation error. This is actually a good thing, since it gives us information about what we need to change, and we got that information from just naïvely writing some code.

The UI class is in the `mastercrupt` project and we must avoid a dependency to that project, since it still contains code that can't be shared

between our two clients. Also, the UI class instantiates an application class, which in turn uses the Leeter for `mastercrupt`. This is best illustrated in the package diagram in *Fig. 2.5 (p.31)*.

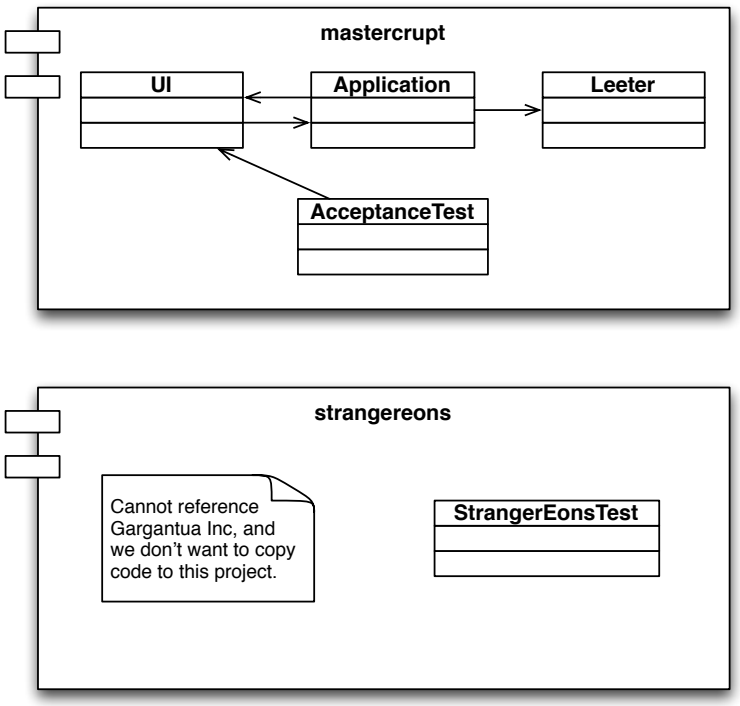


Figure 2.5: Problems with new project dependencies

In this simple case, this could probably have been spotted in advance by an attentive developer, but compilation of speculative code changes inside ones head is not feasible for a more complex system. In this case we made the change to see what happened, and left it to our compiler to tell us where it breaks.

The next natural step would've been running the tests but it isn't pos-

sible because the code doesn't even compile. If we would have been using a dynamic language, where it's impossible to lean on the compiler, the automated tests are even more important. They are the only way to consistently verify the effects of a change.

To resolve the compilation problems, one option is to duplicate the entire project. We won't do that because we think duplication is bad, as described in detail in *Don't Repeat Yourself - DRY (p.106)*. So, we need to change the chain of dependencies in order to allow us to add the test case to the project without any compilation errors.

> ***Why is it interesting to make changes that still compile?*** *IDEs can help out a great deal when it comes to refactorings that require changes to multiple parts of the codebase at the same time. For instance when the name of a method is changed, all places calling that method needs to be changed as well. A modern IDE can automatically do that for you. For a statically typed language, that support is highly dependent on compiling code in order to correctly find the places needed to change. As soon as the code doesn't compile, the automated refactoring support will have trouble finding the references to the altered code. The alternative is manual change which is really tedious, not to mention unnecessary and errorprone. Furthermore, it takes our focus off the real problem, which is doing the least amount of refactorings in order to be able to add more business value.*

### 2.1.4 A naïve resolution to the dependency problems

Now it's time to decide what to do about the dependency problems. `UI` has some common logic which we want to use in both projects. We choose to create a new project for the `UI` code and this project will be used as a common dependency, to share code between customers.

***Decision:*** Put `UI` code in separate project.

Now, the compiler errors have provided us with valuable information which we need to consider. We *Write down the goals (p.19)* and add this as a step to the Mikado Graph in *Fig. 2.6 (p.33)*.

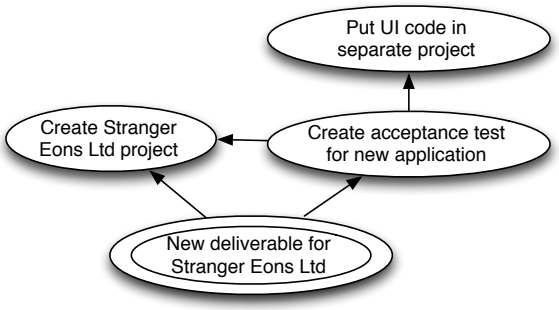


Figure 2.6: The decision to put `UI` code in a separate project

Time to fix the broken system.

### 2.1.5 Roll-back time

Now it's time for a non-intuitive step, but an important part of the method: *Back out broken code (p.20)*!

***Rollback:*** Roll back to the tag "Before new client".

The `strangereons` project has compilation problems and we don't want to do anything there, nor any place else. So, we roll back to the very beginning, which is the state tagged *"Before the new client"*. By doing so, the newly created project disappears and we have the code in its original state.

As a complement to rolling back, the changes that broke the code can be saved as a patch and stashed away until the prerequisites are implemented. When the prerequisites are met, an attempt to reapply the patch can be made. In this example, the changes are reproduced with little effort, so we won't use the stash-and-reapply approach.

### 2.1.6 Implementing the naïve resolution

We continue to *Fix the leaves first (p.20)*. To have the `UI` code in a separate project we need to have a project for it.

***Step:*** Create `UI` project.

We also need to move the code to the new `UI` project, to *Seek things to try (p.19)*.

***Step:*** Move `UI` code to new `UI` project.

Now our graph looks like this *Fig. 2.7 (p.34)*.

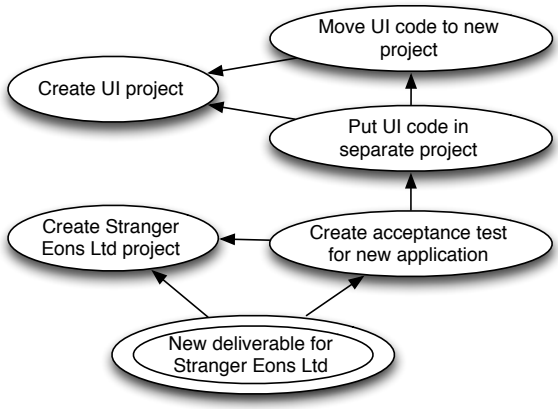


Figure 2.7: `UI` code in a sep project and create `UI` project

This path is one of many that might work for us. There is a large set of possible solutions we can end up with, and depending on our decisions along the way, we implicitly select one of them. The choices we make will affect how easy or hard our path will be. Sometimes we choose paths that aren't feasible, but then we are likely to realize that from an ever growing Mikado Graph. On those occasions, we take a step back

and reflect over the graph, and hopefully see where we took a wrong turn.

Adhering to good design principles often leads us right, and in *Ch.5 Guidance for creating the nodes in the Mikado Graph (p.103)*, some important principles on how to structure code are presented. Sometimes, we experiment to find the right abstractions. The Mikado Method helps us keep track of the needed changes, but technical skills, and domain and market knowledge, are also required to know the right direction to take.

### 2.1.7 Moving the `UI` code to a new project

According to Figure *Fig. 2.7 (p.34)* and our desire to *Fix the leaves first (p.20)*, we now need to create a new `UI` project and move the `UI` code there. That is easily said and done, but *ouch*: The code doesn't compile.

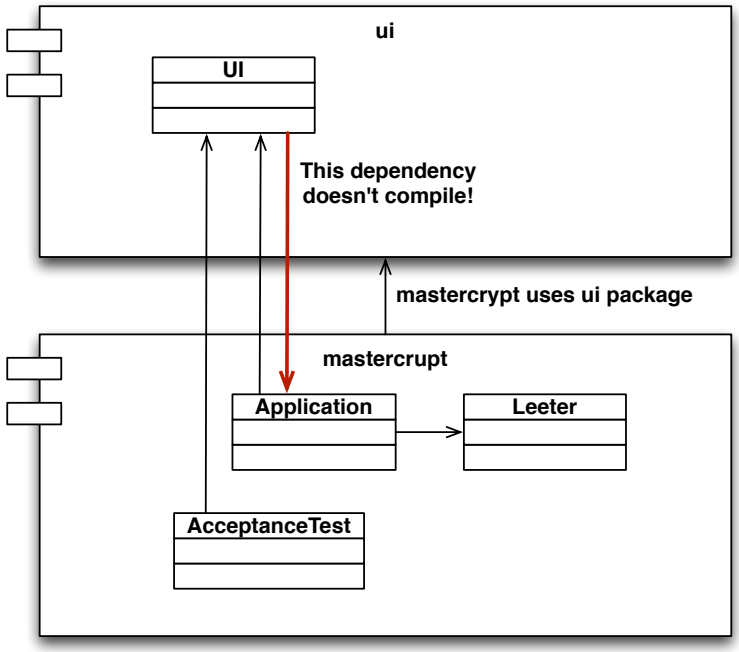


Figure 2.8: Problems with circular dependencies `UI-Application`

As we can see from *Fig. 2.8 (p.36)*, `UI` depends on `Application` and, unfortunately, `Application` depends on `UI`. It's a circular dependency, and we need to resolve that before we can move the code as we wish.

***Decision**:* Break dependency between `UI` and `Application`.

We often add decisions, like breaking dependencies, to the Mikado Graph even before we know exactly how to resolve them. Such items serve as a *decision node*, like in *Fig. 2.9 (p.37)*, and they help us defer commitment until the last responsible moment.

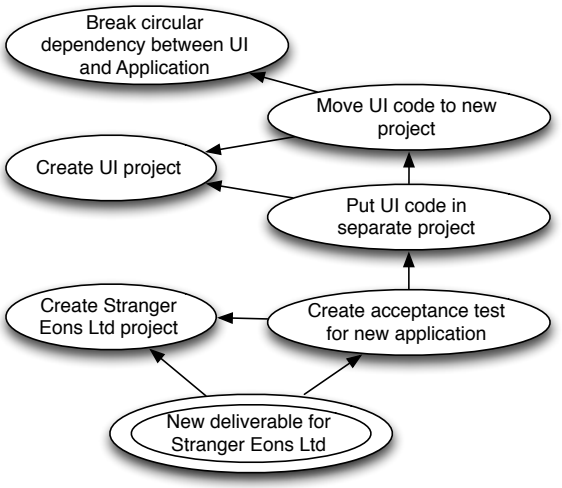


Figure 2.9: Break circular dependency `UI - Application`

After we've added the prerequisite, we roll back our changes again, since we always *Back out broken code (p.20)*.

***Roll back***: All the way

Codewise, we're back where we started, but we have built up a body of knowledge about the application and how we want to change it. We are again in the position to take care of a new leaf in the graph, to *Fix the leaves first (p.20)* without any errors in the code.

### 2.1.8 Resolving the circular dependency

A common way to break circular dependencies is to introduce an interface for one of the classes involved in order to change the direction of the dependency. By doing so, we adhere to the *Dependency Inversion Principle*. See *Dependency Inversion Principle - DIP (p.121)* for more details.

We choose to introduce an interface for the `Application` class, the `ApplicationInterface`.

***Step:*** Extract `ApplicationInterface`, including method `leet(...)`

The graph now looks like *Fig. 2.10 (p.38)*.

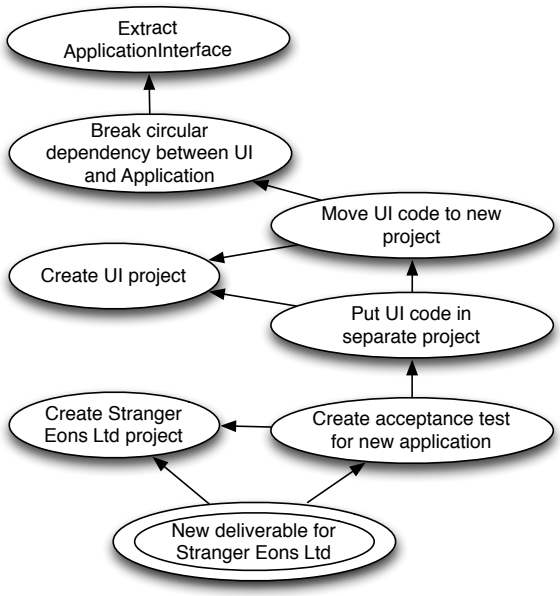


Figure 2.10: Application interface

We have a new leaf in the graph, so we try to perform that step without hesitation. This means creating the interface, let `Application` implement it, and replace all declarations of `Application` where we can use `ApplicationInterface` instead. By hand or with an automated IDE tool, this can actually be implemented without any errors. The affected code looks like *Listing 2.6(p.39)*.

```
public interface ApplicationInterface {
    public abstract void leet(String string, UI ui);
}
public class Application
        implements ApplicationInterface {
    @Override
    public void leet(String string, UI ui) {
        ui.setLeeted(Leeter.leet(string));
    }
    . . .
}
public class UI {
    ApplicationInterface application = new Application();
    ...
}
```

Listing 2.6: The extracted `ApplicationInterface` and its usages

This is good, but we still have the use of `new Application()` in *Listing 2.6(p.39)* that upholds the circular dependency.

### 2.1.9 Should we check in now?

We'd rather check in every small step than keep a batch of uncommitted changes. In reality, we might need to run a test suite locally before checking in. If that takes time, we might try to bite off a little more for every checkin. This again stresses the importance of having fast tests in order to shorten development cycle times.

That said, at the moment we are not sure if this path will actually break the circular dependency, thus we refrain from checking in. This is a judgment call which requires a bit of training to get the hang of it and at this particular time we don't think it makes enough sense to check in.

### 2.1.10 Adding dependency injection

We have only broken half of the circular dependency. There is still the problem of the `UI` class creating an instance of the `Application` class,

we saw in *Fig. 2.11 (p.40)*.

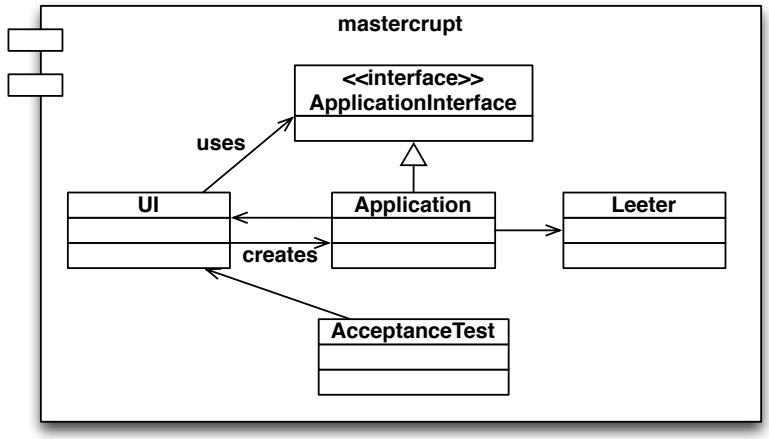


Figure 2.11: The `UI` still creates the `Application` instance

If we want to break the circular dependency chain completely, we need to move the instantiation of the `Application` class outside of the `UI` class, and inject it as an `ApplicationInterface` instance into the `UI` constructor.

***Step:*** Inject `ApplicationInterface` instance into the `UI` constructor.

This is often referred to as *dependency injection (DI)*, and is described more in detail in *Dependency Injection and Inversion of Control (p.124)*. This step obviously require us to have the `ApplicationInterface` in place and after we have added another node to the Mikado Graph, it looks like this: *Fig. 2.12 (p.41)*.

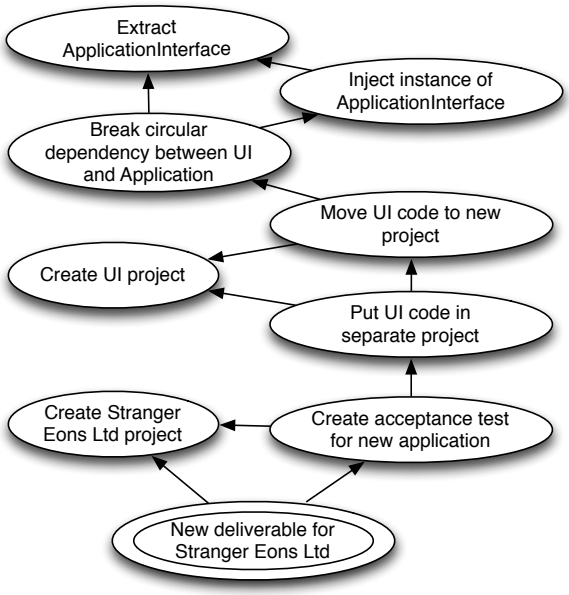


Figure 2.12: Injection `ApplicationInterface` instance in `UI`

The graph now reminds us of our next step in our refactoring and with the extracted `ApplicationInterface` in place it is time to decouple UI from Application. In order to loosen the tight relationship between them we turn our attention to the `UI` class and decide that a simple *Change method signature* will suffice.

Instead of creating an `Application` instance inside the `UI` class we inject the abstraction, an `ApplicationInterface`, through the constructor. This will effectively dissolve the coupling between the two implementations and we are a step closer to our goal. The code changes slightly and now looks like this: *Listing 2.7(p.42)*

In `Application` we change the main method so it looks like figure *Listing 2.8(p.42)*.

The test case is changed in the same way, as in *Listing 2.9(p.42)*.

```
public class UI {
    private ApplicationInterface application;
    private String leeted;

    public UI(ApplicationInterface application) {
        this.application = application;
    }
   . . .
}
```

Listing 2.7: UI class with ApplicationInterface injected in constructor

```
...
    public static void main(String[] args) {
        UI ui = new UI(new Application());
        System.out.println(ui.leetMessage(args[0]));
    }
...
```

Listing 2.8: Create the instance to inject in the main method

```
...
    @Test
    public void testLeeting() throws Exception {
        UI ui = new UI(new Application());
        assertEquals("Leeted: S3cr3t", ui.leetMessage("Secret"));
    }
...
```

Listing 2.9: Creation of instance in test case

Everything compiles and all the tests run! Now the class diagram has no circular dependencies. Note that the *realize arrow*, the triangular arrow, in the UML diagram in *Fig. 2.13 (p.43)* is also a dependency arrow.

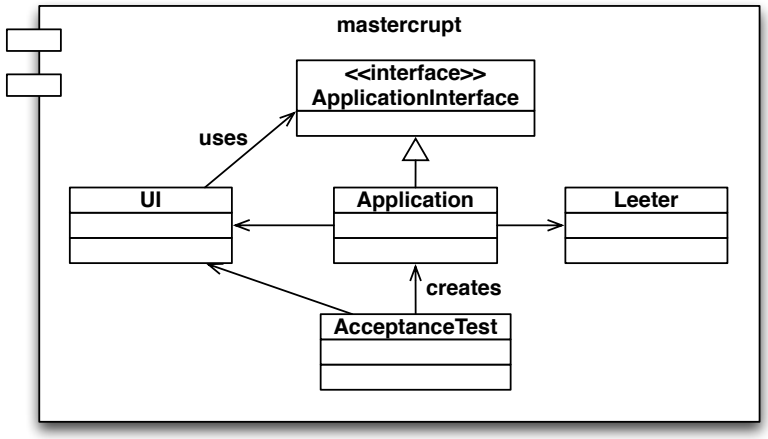


Figure 2.13: Circular dependency resolved

### 2.1.11 First checkin

We have broken the circular dependency in the code, and that *really* makes sense. It is time to check in! We use the label *Broke circular dependency between* `UI` *and* `Application`.

| ***Checkin**:* Broke circular dependency between `UI` and `Application` |
|---|

Now, we can tick off some nodes the graph, including the decision node for breaking the circular dependency since it is completed when all of its dependencies are completed. The graph looks like *Fig. 2.14 (p.44)*.

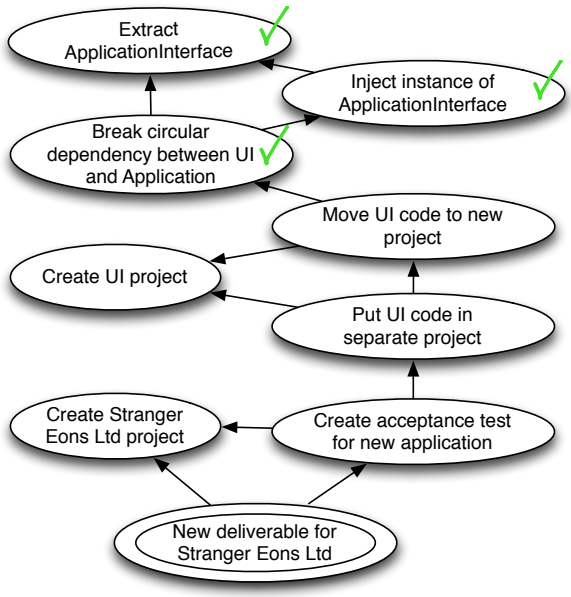


Figure 2.14: Circular dependency resolved

### 2.1.12 Moving the code to new projects

After dealing with some prerequisites, we now seem to be in a position where we, according to the graph, can create the new UI project.

***Step:*** Create new UI project

Now we are able to use the *Move class* refactoring which moves the UI class to the new UI project. The ApplicationInterface is now a part of the UI code, and has to follow along as well.

***Step:*** Move UI and ApplicationInterface to UI project

With an IDE that eagerly compiles code as we edit, it is extra important to add any projects that `mastercrupt` will depend on before we start to

move classes. If we don't, the project can't be compiled and we end up with a broken code base after the move.

We could add this step as a node to the Graph to remind us about the project dependencies, but in this case we don't, because a node like that would be too short lived.

Adding a lot of unecessary details and steps to the Graph only makes the Method feel like a cumbersome process and we would end up with a lot of overhead and ambigious graphs, which is exactly what we are trying to avoid.

Let this brief example serve as a reminder that sometimes obvious details or short live nodes can be left out of our graphs if that makes them more effective.

It is now possible to move `UI` and `ApplicationInterface` *without any compilation errors*, and the result is shown in *Fig. 2.15 (p.46)*.

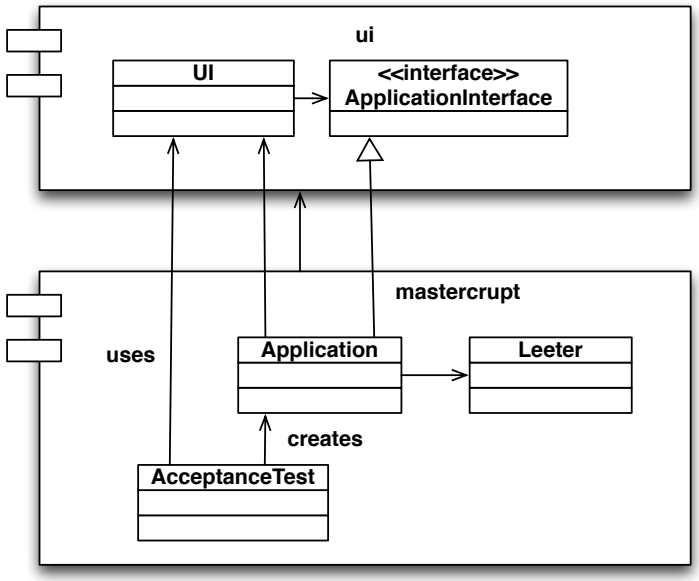


Figure 2.15: Classes in new UI project

The tests run, so we check in.

***Check-in**: UI-code in separate project*

Of course we also tick it off in the Mikado Graph in *Fig. 2.16 (p.47)*.

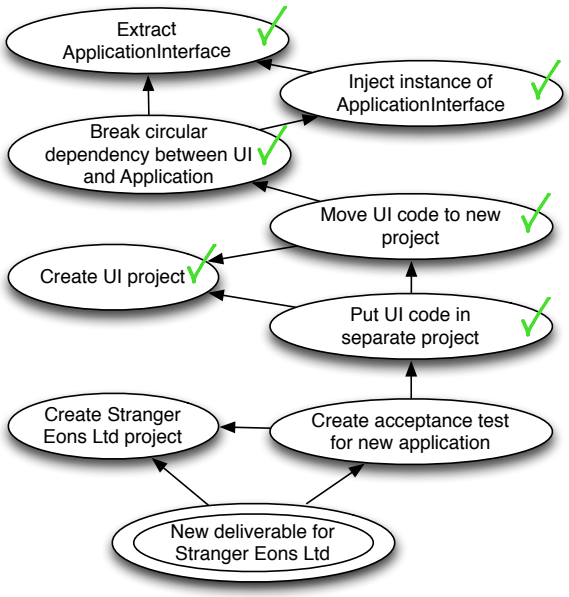


Figure 2.16: UI project established

### 2.1.13 Creating the new deliverable

If we look at the graph, we can see that the next leaf is almost where we started out, to create project for Stranger Eons Ltd. We also need to make it depend on the UI project, since we do want to actually use the UI project that we have just extracted from the codebase.

The test case looks like the one we created earlier, with the addition that the UI constructor takes an argument. The result is in *Listing 2.10(p.48)*.

At first, the new project doesn't compile since there is no local Application implementation of the ApplicationInterface. We implement the interface with an empty implementation, as in *Listing 2.11(p.48)*.

```
...
    @Test
    public void testLeeting() throws Exception {
        UI ui = new UI(new StrangerEonsApplication());
        assertEquals("Leeted: 5ecret", ui.leetMessage("Secret"));
    }
...
```

Listing 2.10: The testcase for Stranger Eons Ltd.

```
...
public class StrangerEonsApplication
        implements ApplicationInterface {
    @Override
    public void leet(String string, UI ui) {
    }
}
...
```

Listing 2.11: Make the Stranger Eons Application instance implement the ApplicationInterface

Now we can run the tests, but they fail since the implementation of `ApplicationInterface` is empty. Let's fill the implementation with something like that in *Listing 2.12(p.49)*. We also added the main method, like in the Gargantua application.

### 2.1.14 Done and delivering!

Now the tests pass! We're done and can deliver the system to Stranger Eons Ltd. The system now looks like *Fig. 2.17 (p.49)*

```
...
public class StrangerEonsApplication
        implements ApplicationInterface {
    @Override
    public void leet(String string, UI ui) {
        ui.setMessage(string.replace('S', '5'));
    }
    public static void main(String[] args) {
        UI ui = new UI(new StrangerEonsApplication());
        System.out.println(ui.leetMessage(args[0]));
    }
}
...
```

Listing 2.12: Make the test pass

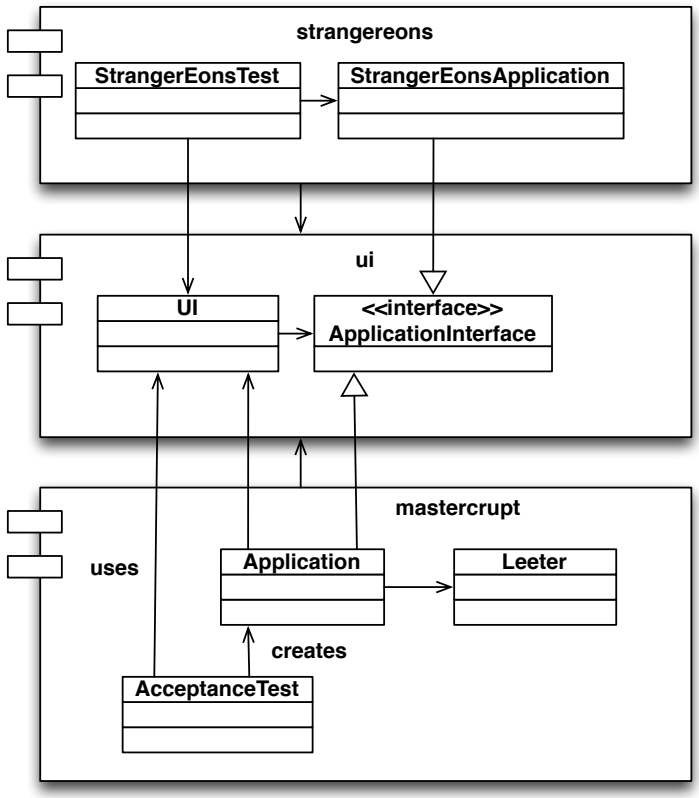




Figure 2.17: The final result. The system is ready for delivery!

We can tick off the last nodes of the graph in *Fig. 2.18 (p.50)*.

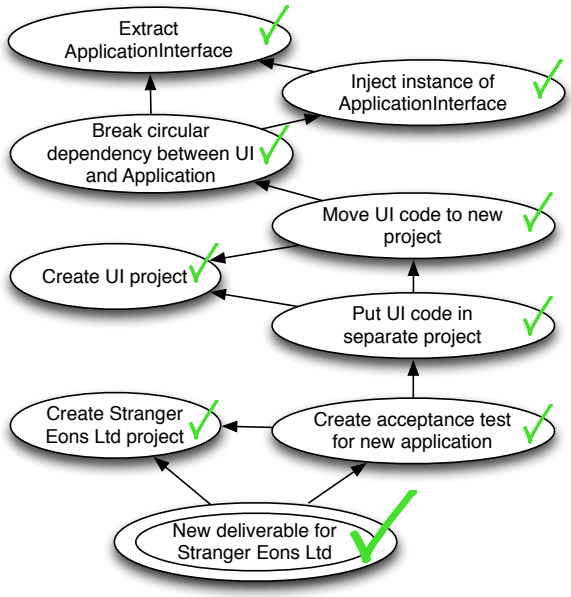


Figure 2.18: Done!!

## 2.2 Conclusion

We've managed to morph the code from one state that didn't allow us to do what we wanted, to one that actually does what we want. We did this by using the Mikado Method to *Write down the goals (p.19)*, *Seek things to try (p.19)*, *Back out broken code (p.20)* and *Fix the leaves first (p.20)*. We used the Naïve Approach combined with some simple analysis. Along the way we made use of some test-driven development, and atomic and composed refactorings. By focusing on doing small changes, step-by-step to isolated pieces of code, we kept the system in a working condition all the time. In addition to that, we also wrote

some brand new code to implement the new functionality.

Finally we ended up with a restructured system that is more suited to the needs of Pasta Software and their clients. The code might not be perfect, but it does the job.

Our attention was focused on actual problems which arose during the initial creation of the graph. After that, we quickly arrived at a feasible solution.

When we first started out, we didn't know the single most effective way to reach our goal, but this is very seldom required. The goal is to reach a *good enough solution* to a particular problem. There are often several ways to reach the same goal and perhaps better ones as well in this case. If we had the time, or the problem was small enough, we could reiterate the whole or parts of our refactoring and see what decisions make for what results.

> *For this particular problem, the authors have refactored the code more than a dozen times, and come up with several different, and working, solutions each time. The Mikado Method may produce different paths each and every time but don't worry, it always leads to the goal. It's an inherent feature of the Method which makes it very pragmatic and suitable for changing a system.*

If we find more things we want to change, the same method can be used over and over for different goals.

## 2.3 Using this example as a software Kata

The term *Kata* comes from Japanese martial arts and is the name of a type of exercise that aims at practicing a particular set of actions, over and over again. The example above can be used as such a Kata for practicing the basics of the Mikado Method, and the URL to the example can be seen in *Try This*. There is a short appendix on Katas in *App.A Software Katas (p.239)*.

## 2.4 Summary

Jake: Wow, we did it! We actually did it!
Melinda: Yes, we did! Great feeling, huh?
Jake: Very much. The Naïve approach stopped me from spending a lot of time analyzing, it just took me straight to the problems.
Melinda: Exactly.
Jake: It was also a good feeling to first capture the knowledge in the graph, and then start checking the nodes off as we worked our way back.
Melinda: Did you notice that we mostly ended up doing simple, atomic refactorings, except for implementing the logic for the new customer?
Jake: Yes. And you can apply the same principles for even larger systems, right?
Melinda: You follow exactly the same flow in a larger system. The main difference is that you might change hundreds, or thousands, of files at a time... of course with the help of automated tools.
Jake: That sounds cool, and scary.
Melinda: It can be, but more often it is actually rather comfortable.

## 2.5 Try this

- Get the code for the example from
  `https://github.com/mikadomethod/kata-java`
  or
  `https://github.com/mikadomethod/kata-dotnet`
  and do the steps yourself. Start with doing the exact mechanics as above, and draw the graph exactly the same way.
- Try to solve the same problem in your own way. What did you do differently?
- Try the method on some of your own code. How does it feel? Did you find a reasonable path?

# Chapter 3

# More about the Mikado Method

Jake: This is exciting!

Melinda: Good to hear! Now you got to see it in action as well.

Jake: That was great. Now I'd like to know some more about the graph...

Melinda: Sure.

Jake: ...and I'd also like to know when it breaks.

Melinda: I'll explain some of the principles and some relations to other thinking models to help you out.

Jake: Ok, how will that help me?

Melinda: You'll see better how it all fits together, and you'll be able to know better how to apply it and experiment with it.

Jake: Ok, so I can change it?

Melinda: Yes, if you need to. There are a few principles that I follow. You will probably be better off if you understand them.

Jake: Sounds great!

## 3.1 The details of the Mikado Graph

The Mikado Graph is a way to graphically represent the knowledge about which changes depend on which other changes. Lets look in more detail at the components of the graph.

### 3.1.1 The Mikado Goal

We call the goal the *Mikado Goal*. It should preferably be something that has a business focus, but it can also be a more technical goal. We want to *Write down the goals (p.19)*, so we draw the Mikado Goal on a piece of paper or a whiteboard, like in *Fig. 3.1 (p.54)*. We could use some software drawing tool, but our preference is for physical tools.


Figure 3.1: Start with the Mikado Goal

The double circles are there to tell more quickly where the original goal is when the graph gets bigger and more complex.

All the bubbles in the graph are goals that are treated the same, in that we want to achieve them. The most special thing about the Mikado Goal is that it expresses the starting point of the change.

### 3.1.2 The prerequisites

We've already mentioned that prerequisites often manifest themselves as compilation errors or test failures. We don't try to fix the errors immediately. That would be like trying to pick a stick from the bottom

of the pile in a Mikado Game. Instead, we just analyze the situation enough to decide on immediate actions that will resolve the errors. This could be something very intelligent, or something naïve, but we always *Seek things to try (p.19)*. The resolutions are drawn as dependent prerequisites in the Mikado Graph, like in *Fig. 3.2 (p.55)*. The prerequisites and the Mikado Goal are essentially treated the same way. We *Fix the leaves first (p.20)*, and we *Write down the goals (p.19)* of all the things that stop us.

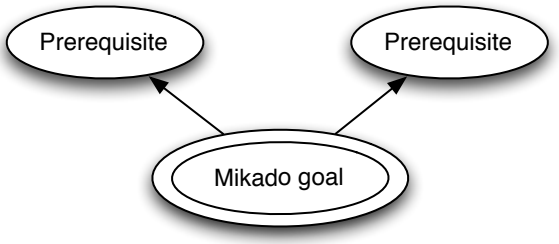


Figure 3.2: Immediate prerequisites

When we don't know what the consequences of our actions will be, we choose a naïve resolution to the problem to keep our momentum. When we realize there are things that are not immediately shown by the compiler or the tests, we write these down as prerequisites as well. This way, we can put all of our available tools and knowledge to use when we decide what to do. We also save a lot of time by being naïve and avoiding unnecessary analysis. We are naïve and practical when we can be, and smart and analytical when we must be.

This keeps us from ending up in *analysis paralysis* mode.

One rule of thumb is that; When we have been thinking for more than a couple of minutes, we try something naïve and practical again.

### 3.1.3 The transitive prerequisites

In the same way as we try to achieve the original Mikado Goal, we try the immediate prerequisites. At this point we often discover prerequisites to the prerequisites which are *transitive prerequisites*. We note those in the graph as well, like in figure *Fig. 3.3 (p.56)*. Then, as always, we *Back out broken code (p.20)* and *Fix the leaves first (p.20)* by repeating the process with the next node that has no prerequisites (yet).

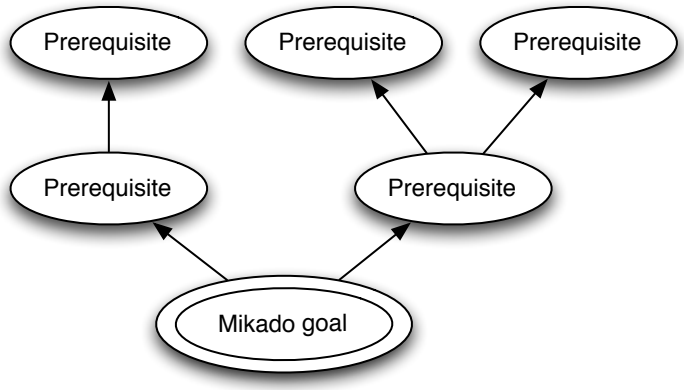


Figure 3.3: Next level of prerequisites

### 3.1.4 Prerequisites - steps or decisions?

In *Ch.2 An example in code (p.23)* we interchangeably called the prerequisites *steps* and *decisions*, but we don't draw them any different. The reason is that we do not want to create a new Unbeatable Modeling Language, with detailed rules about the notations used, and new versions of the notation every time we realize there are things not yet covered.

Instead, we try to limit the amount of notation details, and thus the risk of drawing the graph in the 'wrong' way. This also gives more flexibility

for the users to modify the graph notation to fit their needs. If there is a strong desire to make the decisions or any other node stand out, feel free to draw it in a different color or shape.

From a Mikado execution perspective, it doesn't matter much. We will have to implement all the prerequisite in due time.

### 3.1.5 The leaves

Repeating the process to *Fix the leaves first (p.20)* usually leads to a graph like in figure *Fig. 3.4 (p.57)*. By working our way through the code we are able to perform some change without getting any errors. This is a good indication that tells us we have reached the *leaves*. A leaf is usually a fairly simple action, like an atomic refactoring.

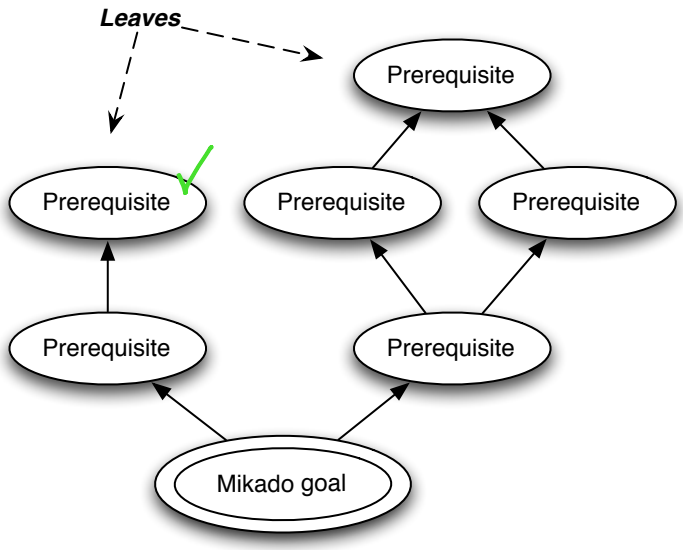


Figure 3.4: One tick mark

The leaves in the Mikado Graph are the only places where anything can actually get done. Otherwise, the system will break as a change is attempted. All other nodes are inhibited by their prerequisites. A leaf is a true leaf only when we can implement it, and all aspects of the system still work as they are supposed to. Some prerequisites may look like leaves at first, but if they have other prerequisites when we implement them, they are of course not leaves any more.

When all the prerequisites for a node has been removed, the node itself becomes a leaf, ready to be dealt with. Any leaf is as good as a starting point as another. Just pick one and see what happens.

### 3.1.6 The dependency arrows

As we *Write down the goals (p.19)*, we also add the dependency arrows between the nodes. The dependency arrows are pointing from the original goal towards their prerequisites, see *Fig. 3.5 (p.59)*. This might feel a bit awkward at first, since we often think of what needs to be done and point the arrows in that order. However, the arrows in the Mikado Graph represent dependencies so when we perform the changes, we travel in the opposite direction of those dependencies starting with the prerequisites first.

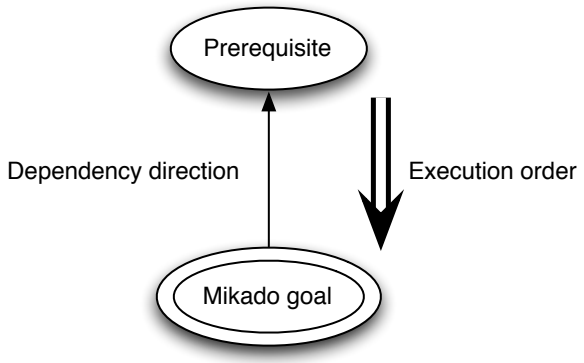


Figure 3.5: Dependency and execution directions

### 3.1.7 The ticks

Sometimes, we tick off the leaves one by one as we complete them, and sometimes we tick them off in a batch as we check in. If we tick them off one by one, we might face the risk of having to untick them if we discover that we missed something and must revert our changes. The ticks we usually use looks something like in *Fig. 3.4 (p.57)*.

We recommend using a distributed version control system or at least a personal, short lived, branch. Local copies and branches enalbles us to make as many commits as we want, since they don't affect the main code repository. We also encourage frequent commits within that branch, especially when some form of state that represents progress has been reached. That way it is easy to revert to a known state.

Ticking the leaves off is a good way of seeing progress and what we have accomplished. It is often a rewarding feeling to tick off leaf after leaf, knowing they are steps leading towards the goal.

Another way to tick off leaves is to just erase them as they are completed. The advantage is that we might need the space on the whiteboard, and the graph is not cluttered with finished work. The disad-

vantage is when analyzing what we did afterwards, or when we want to see all the things we have accomplished, we cannot do so.

The above are a few variations that come rather naturally. We just let the context decide how we do it. After ticking off all the leaves, the graph will look like in figure *Fig. 3.6 (p.60)*. This is what we are striving for!

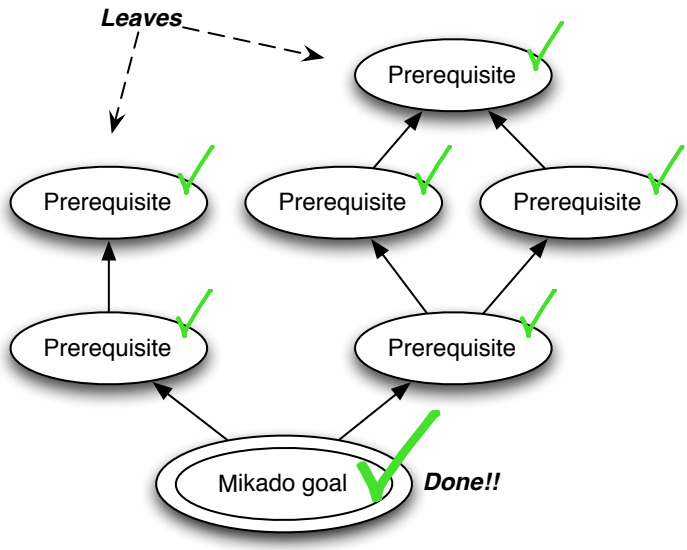


Figure 3.6: Done!

## 3.2 More details on building the graph

### 3.2.1 The naïve approach

When we *Seek things to try (p.19)*, and try to achieve the goal right away, we call it *The Naïve Approach*. We *naïvely* try to implement a change without analyzing too much in advance. By doing so, we don't

need to think about how different possible changes affect each other, something that could keep us in analysis paralysis mode forever. Instead, we just do it! As a start, we only need to know what we want to achieve, which is our core change. We don't want to analyze all the details or force ourselves to consider all possible complications that our change might impose on the system.

More often, we just make a naïve change. Then we see that the system is not built for the type of change we want to make, be it a mess or a well built system. We get errors. Those are not failures but rather exactly what we want! We have pulled information out of the system, that tells us what we can't do right away. We use that information to understand what *prerequisite changes* we must do first to the system.

The Naïve Approach is a way to create the Mikado Graph empirically. We let the feedback from our change guide us to the next step. This powerful technique is put to use in order to learn where the code pushes back on us. We use it to find places where we have circular dependencies, static calls, global variables, methods in the wrong class, or something else that makes the code hard to change. 'Oops, I did not realize I couldn't move this class here. How interesting!' Our objective is to push the code in the direction we want it. If it pushes back, we have found something that stops us. Great! We learn from it and add it to the graph. Then, we revert the code to a working state. We use these insights to define the change path and increase our knowledge about the system. As strange as this may sound it actually works very well in practice. We describe this in detail in *Ch.2 An example in code (p.23)*.

### 3.2.2 Building the graph using analysis

Even though the Naïve Approach is very powerful, there are cases in which we can't use it. There are also situations when we need to combine it with analysis. For instance, there might be nothing to change that can give us feedback, and we need to do a bit of analysis to figure out what we should do to start getting that feedback. We *Seek things to try (p.19)*.

In the example in *Ch.2 An example in code (p.23)*, we start off by doing some analysis, i.e figuring out what we need. Then we create the new

project and a test case for the new functionality. At that point, we can get some feedback, but in order to come up with those first steps you need some analysis. When we have the test case, we realize that we can neither use nor move the classes as desired. The feedback loop is closed, and we can start relying more on the Naïve approach again.

### 3.2.3 Undo is a dear friend

When working with computers, we have a great tool that is not available in the real world: **Undo**. We have made the first change to the system, the current Mikado Goal, and we have seen some places where the system breaks and we have written them down as prerequisites in the graph. The system is now upset; "the sticks are all over the place". This means we cannot compile the system, or run all the tests, so we are in a bad state at the moment. To get back into the last known good state, we *Back out broken code (p.20)*.

In a versioning system, Undo comes in the shape of *revert*, or *rollback*, which is essentially the same thing. It allows us to take risks and test things we would not, or could not, do in real life.

In his book "Working effectively with legacy code" [1], Michael Feathers talks about *scratch refactorings*. Scratch refactorings are changes we do to explore our options only to return the code to its initial state when we are done. The iterations over the Naïve approach and Undo to build the Mikado Graph is a systemization of scratch refactorings in order to learn more about the system.

### 3.2.4 Zoom in to focus, zoom out to reflect

The Mikado Graph helps us zoom in on a single task and perform that task really well. When we are done, we use the Mikado Graph to zoom out to get an overview of what we are doing, and why. If we learned anything new from the task we just completed, we add it to the graph or take care of the new knowledge in some other way.

### 3.2.5 Are we really throwing away the code?

This is so important that we think it's worth mentioning again: We always *Back out broken code (p.20)*, since we find it much more difficult to work with a broken codebase than to revert our changes and do them all over again. We have already made that mistake when we first faced the Hydra during the discovery of the Mikado Method, and it didn't end well. We want to *Fix the leaves first (p.20)* and get feedback from the compiler and the tests so they can guide us as we change the system. When the code isn't compiling, we get neither and we believe there is a way forward *without* breaking the code.

To some people, reverting the broken code feels like throwing work away and starting all over again, like it never happened. We believe this is a misconception of what developing systems is about. System development, and especially refactorings or restructurings, is almost exclusively about learning about the system, the domain, the language and technology in use. The actual performing of the changes accounts for just a fragment of the total development time and the great value of the Naïve Approach is what we find out and learn about the system, so we can see what actually stands in our way. The Mikado Graph then holds that information for us, so we can use it and decide exactly what we want to do at a later time. Hence, nothing much is really lost when we revert our actions. Contrary to what many believe, the stage is set and we're ready to execute some changes that will probably compile and work.

Some people want to take the broken code and *stash it* and reapply it later when the prerequisites are in place. This is accomplished by using the versioning system to make a patch file, or the equivalent mechanism for stashing away a version and reapply it later. This can work in an odd case or two but more often it is easier to redo the changes. Experiment and see what fits best in different situations.

### 3.2.6 How big should a step be?

This is what Joakim Ohlrogge told us about when he was learning the Mikado Method:

> A feeling that soon came over me when I started out was "Am I doing this right? How large is a typical Mikado step?". I decided that it should be comfortable. So, what is "comfortable"? I decided that it can't be too big, so I loose control over what I'm doing and can't remember what I've done since my last "saved (drawn)" step. And it shouldn't be so small that I feel that the process gets in my way rather than supporting me.

Getting in the way and supporting is highly dependent on the context. Sometimes, we might want to be a bit more detailed, for instance when we need to convey information to others. Other times we don't need that granularity because we already have a common language and the same idea of how things should be accomplished. At times like that we can choose to be a little less detailed in order to make progress faster.

### 3.2.7 Will I write better code?

The Mikado Method will not by magic means make anyone better at designing and writing software. Initially, it will just help putting existing skills to use in a more efficient and effective way. In the long run, our experience tells us we see the problems with the current design faster, thus giving us more learning opportunities. Harnessing these learning opportunities is up to the individual.

We have also seen a shift in developer approach, towards making more fine-grained changes and more frequent checkins even when not using the Mikado Method. We believe this is an improvement that leads to better code.

In *Ch.5 Guidance for creating the nodes in the Mikado Graph (p.103)* we try to give a glimpse of software design principles that has helped us become better programmers, and better at restructuring software.

## 3.3 The principles of the Mikado Method

The Mikado Method is relatively easy to explain, and the body of knowledge that is needed to use it is fairly limited. While this makes it easy to get going, it is also easy to assume that any change or adjustment made is an improvement and it will still work.

The Mikado Method rests on a handful of principles, just like many other methods. The more the principles are honored, the better the results will be.

We want to explain more than the bare mechanics, as a way to give more insight and provide the same foundation we have. Because when that foundation is in place, more astute use of the Method and even improvements in the method itself can take place.

### 3.3.1 Goal orientation

When we *Write down the goals (p.19)* and do things that are connected to the Mikado goal, often a direct or indirect business goal, the most important task is tackled first when we *Fix the leaves first (p.20)*. That probably makes sense but that focus is easy to lose when pressure is added or distraction comes from a boss, customer or someone else.

The clear, goal-oriented approach also doubles as good base for dialogue with business people who does not always understand how technical implementation details are tied to business changes. With the goal and the graph, people who aren't responsible for the technical implementation can see how that work is directly connected to business needs. This can help disarm a more than familiar situation where developers are accused of delving into technology for technology's sake. It also avoids the reverse situation where business people are accused of not understanding the essential nature of technical improvements.

Defining a very specific and distinct goal makes the code easier to work with, not just because it can be discussed and related to, but also because it provides a clear focus and something tangible to work towards. At the same time prerequisites are more easily identified and achieved along the way and optional solutions to the same problems often present themselves in a totally different light when there is a plan.

The bigger decisions are best made when the whole picture is visible, i.e, in a zoomed out mode, whereas the technical decisions are best made as the code is written. The graph is the plan and serves as a high level strategy for reaching the goal and the thinking and the nodes are more of a tactical orientation. We can zoom out for strategy and zoom in for tactical matters.

### 3.3.2 Transitivity

When all the nodes in the Mikado Graph are created from true prerequisites of the Mikado Goal, and we *Fix the leaves first (p.20)*, the work will flow naturally in the right direction. We don't worry when different ways present themselves. There is always more than one way that will lead to the goal. Most ways are just variations and some are equally good.

The code in *Ch.2 An example in code (p.23)*, has been solved in at least four different ways. In spite of taking different paths, leading to different solutions, we have always fulfilled the business goal and that is really a very small example. For a normal system there is an almost infinite number of paths to achieving the goal.

The Mikado Graph cannot be replaced with a simple list of things to do, since that will cause the transitive information that makes up the graph and its dependencies to be lost.

### 3.3.3 Pragmatism

The method is pragmatic as it doesn't look for the optimal solution, just one that is sufficient for the goal at hand. The prerequisites allow fine-tuning the of solution, but the method itself does its best to maintain focus and avoid *gold-plating*.

Many times, engineers and developers want to be able to say "This is the best way to do it!" The Mikado Method takes a different and more humble slant, saying "This is one way to do it."

We try to remember that "*perfect* is the worst enemy of *good enough*."

### 3.3.4 Simplicity

The naïve, exploratory approach of revealing what is in the way will point in the right general direction when we *Seek things to try (p.19)*. It shows where the next problem is in the simplest way we know. The graph is also the simplest possible thing we can come up with that helps us remember the things we need to do in order to reach our goal.

### 3.3.5 Visibility

Visual aids and information radiators such as whiteboards, flipcharts or plain paper can provide an overall picture in any modern project. This also works when we *Write down the goals (p.19)* when using the Mikado Method and it guides and reminds us of where we're going and what we are currently working on.

### 3.3.6 Iterative, incremental and evolutionary

> Q: How do you eat an Elephant?
> A: One bite at a time.

The Mikado Method is incremental in the sense that it tells how to create a refactoring map one step at a time. This also implies that the Graph will continue to change as long as prerequisites and transient prerequisites are found on the journey to the goal.

While the effects from the actual change effort comes as small alterations and improvement increments, the Method itself operates iteratively in the process dimension. This means that the system can evolve incrementally from work in an iterative fashion which opens up the possibility to select a good time for making critical changes or spread the work over time.

It also helps and encourages a one piece at a time execution flow where it is possible to stop at any time, like at the end of an iteration, to make a release.

As opposed to a Big Bang change, the iterative nature of the Method and its incremental results produces an evolutionary effect rather than

a revolutionary one. This is normally preferred as it is easier to cope with. No large merge work, no stray branches that people wonder if they are still needed and most importantly, code can be delivered and deployed at all times.

If the work is spread out over longer periods of time it is likely that change will happen outside of the scope of the change that was first addressed. If this happens, and it usually does if the change is a big one, the Graph needs to be revisited and updated. Sometimes a whole branch must be replaced and sometimes a few new transient dependencies is sufficient.

Regardless of the impact of the change, the practical nature of the Method encourages assessment of the new opportunities and makes the implications of the disappearance of others easy to handle.

### 3.3.7 Statefulness

Figure 3.7: Simple things to the rescue

There is a man walking in the country side. It's a beautiful summer day and the sun is really doing a good job of reminding him of how thirsty he is. In the distance up on a small hill there's an old well, the kind where there is a crank and a bucket on a chain. Determined by his thirst the man starts to climb the small hill and as he walks up to the well and looks into it he feels the cold refreshing air from the well on his face.

He decides to haul some water using the old bucket and the crank which despite it's old age looks like it is in a working condition. He folds his jacket and puts it on a small rock just beside the well, then tosses the bucket down into the well and he hears a distant splash.

The bucket is heavy and the well is a bit deeper than he initially thought. The mans arms start aching and he feels the need to take a short break.

The kind soul that built this well knew that hauling water is hard work and he put a ratchet on the clog, so the bucket won't fall back in the water if the crank is let go. The man pauses, catches his breath, and is ready to go. After another short pause the reward arrives and the man cups his hands and puts them in the refreshing water.

Many big refactorings are like hauling water from a deep well, it's really hard work and pausing is needed. We have done restructurings that have taken days, weeks, and even months in some cases. We know that large restructurings benefit from pausing and reflecting.

Without a ratchet, this is not possible. The Mikado Graph serves as *our* ratchet which is why we *Write down the goals (p.19)*. It allows us to take a pause and leave our work for a while. When we're ready to pick up the work again we can do so, exactly where we left off. The graph keeps track of how far we've come and it saves a lot of work so that we don't have to backtrack and examine the codebase to know where we are. Just like a ratchet, the graph provides the support needed to catch a breath which in the end saves a lot of time and energy.

The graph is also a very fast way to refresh our memory. We sometimes compare the graph to a *saved file* from a computer game, which can be loaded into the working memory. Just by looking at the graph, the priorities, thoughts and goals are restored. The graph can lie untouched for a long time but after only a short glance, work can be instantly picked up where it was left off. Compare this to reading a long list of

TODOs hidden in the code or a huge technical debt backlog. Not as compelling right?

### 3.3.8 Reflectivity

The Mikado Graph help us when reflecting over previous refactorings, in discussions and learning. Even in the case where a refactoring could have been performed with a few small steps the Mikado Graph can be useful. We can use it to reflect over the path taken to see if something could have been done differently, and what that might have been. Often just taking a short coffee break can put the Mikado Graph in a different light and make it possible to see it as if it ware for the first time.

The Mikado Graph can also help when practicing and rehearsing refactorings, over and over, until it can be smoothly executed. This is especially useful when working with a change that is hard to do in many small steps over a longer period of time, as described in the story in *When we cannot check in small changes (p.99)*.

## 3.4 Relation to other thinking models

No part of the Mikado Method is really new. We have already mentioned Michael Feathers' scratch refactorings as one critical part, and reverting changes in a versioning system is as old as versioning systems. Writing graphs probably have hundreds of years of history, and refactorings were popularized by Martin Fowler many Internet years ago.

As often when we find something that works, we also find that it rests on principles, practices and ideas from other fields. These are some of those in a more condensed format.

### 3.4.1 Theory of Constraints

The Theory of Constraints (ToC) was introduced by Eliyahu M. Goldratt in "The Goal". One part of the ToC is the *Thinking Processes*, where

one tool is the *Prerequisite tree*. The Mikado Graph represents such a prerequisite tree for a given Mikado Goal.

The prerequisite tree has a goal called the *Injection*. To reach that goal, *obstacles* need to be overcome. For each obstacle, there is an *intermediate objective* that overcomes each obstacle. The objectives can in turn have other obstacles that need to be overcome, in a transient manner.

The Mikado Graph contains the injection, the Mikado Goal, and the objectives. Which are the prerequisite nodes.

### 3.4.2 Empirical control systems

An empirical system can be defined as a system that generates a different outcome at different times, even though the inputs are exactly the same. A common example of this is when driving a car. Even though the exact same pedals are pushed and the same steering wheel is turned, a car can end up in different places. This is due to the fact that empirical processes are random. They are not random in the sense that the results can be anything, but random in the sense that there is a correlation between input and output. The exact outcomes will vary. When steering a car, there is a correlation between the steering wheel and the way the car turns. When you steer right the car turns right, but the exact degree of change is unknown.

There is a whole body of science on how to control such systems called Control Theory. The common key is control systems they require feedback and corrective action in order to be controlled.

When the Mikado Graph is built using the Naïve approach, the compiler and the tests provide the feedback, and the Mikado Graph is then extended with the corrective action needed. In Control Theory, the essence of fast feedback is important in order to minimize the errors in the process. In the Mikado Method, it is important that small experiments are performed to get feedback instead of doing long sessions of analysis.

One could argue that the codebase is defined and not stochastic, but from the eyes of a developer, it is usually not.

### 3.4.3 Scientific method

The scientific method is

1) building a (new) hypothesis

2) creating experiments to verify or break the hypothesis

3) performing experiments

4) analyzing results -> 1)

The Mikado Graph can be viewed as the hypothesis. The experiment is to actually perform the next leaf, which can be the Mikado Goal itself. As long as the actions of the leaves can be performed, the hypothesis holds. When there are compiler or test errors, the hypothesis breaks and it needs to be updated with new knowledge. When the Mikado Goal can be implemented, the hypothesis is held all the way.

### 3.4.4 Pull systems

In Lean, one of the central concepts is that value should be pulled from the system. In practice, this means that if someone orders a car, the sales person pulls a car from the assembly line, which in turn pulls the engine, the chassis, the drive line and the tires from their respective suppliers, and so on, all the way to the nuts and bolts. As the nuts and bolts are delivered, the engine and the chassis are assembled, and in turn the assembly line can put the car together and deliver it to the customer. This system has proven be very competitive in manufacturing and several other disciplines.

The opposite, a push system, is essentially creating something without establishing the need for it. It could be a manager telling the employees in a factory to start manufacturing a certain number of cars and hope that sales can sell them. This approach introduces a greater risk into the system because relying on forecasts, predictions and guesses is inheritly more risky. The manager could for instance get the number of cars right but the color of the interior wrong, or vice versa. As the number of properties that can vary goes up, the bigger the risk becomes with a push system.

The Mikado Method can be viewed as pulling the needed changes from the codebase. The compiler and test failures resulting from the Naïve approach pull the next changes, and do so on until the leaves are reached. Which are the 'nuts and bolts' of the Mikado Method.

The opposite of pull systems is the 'push' version. Predictions of the changes that are needed are used, without actually verifying with something like the naïve approach. That often results in changing the things that are *easy* to change rather than the things that *need* to be changed.

Sometimes this means that the Mikado Method won't allow a developer to change that part he hates with his gut. However, more often developers have a good gut feeling of what is wrong with the system, and the Mikado Method helps them to prove that.

In this chapter we focused on the underlying values behind the Method which we believe has to be understood in order to adapt or change the approach to a suit specific need or circumstances. When the values are internalized, it will become easier to see when the Method can help or when it's just going to be in the way.

## 3.5 Summary

| | |
|---|---|
| Jake: | Whooa, there is quite a lot of depth to the Mikado Method! |
| Melinda: | Yes, it is related to some successful thinking models, like pull systems, the scientific method, and the theory of constraints. |
| Jake: | And although it seems a bit *ad hoc* at first, there are actually some principles that seem essential. |
| Melinda: | Exactly. If you disregard them, you weaken the whole system. |

## 3.6 Try this

- Have you used any of the other thinking models described in this chapter? How did it go?

- What values are most aligned with your beliefs?
- What values would you have a problem with in you current work environment? What could you do about those impediments?

# Chapter 4

# Patterns when using The Mikado Method

| | |
|---|---|
| Jake: | Now I'm starting to get the hang of this. |
| Melinda: | Good. The rules of the Mikado Method are fairly easy to grasp. |
| Jake: | Yes, indeed. But I guess there is some more to it? |
| Melinda: | Yes and no. |
| Jake: | Huh? |
| Melinda: | The method itself has just a few simple rules, and those rules help you handle just about any situation. |
| Jake: | You mean the graph is never the same? |
| Melinda: | Exactly. But there are a few situations I have found myself in more than others. |
| Jake: | Ah, like patterns? |
| Melinda: | Yes, like patterns. |

## 4.1 Graph patterns

When using the Mikado Method and drawing graphs, there are some patterns we have seen that recur from time to time, and that might be

useful to experiment with in certain circumstances.

This is not an exhaustive set in any way, but rather here to show some ways of being creative about problem solving. We urge anyone to come up with their own patterns. The only thing to remember is not to get too analytical and scientific about it. The graph should always be easy to draw and understand.

### 4.1.1 Having multiple nodes depending on one other node

Every now and then we have a bundle of nodes that depend on the same other node. Instead of drawing tens of arrows, cluttering the drawing, we gather them in a grouping node and draw one arrow.

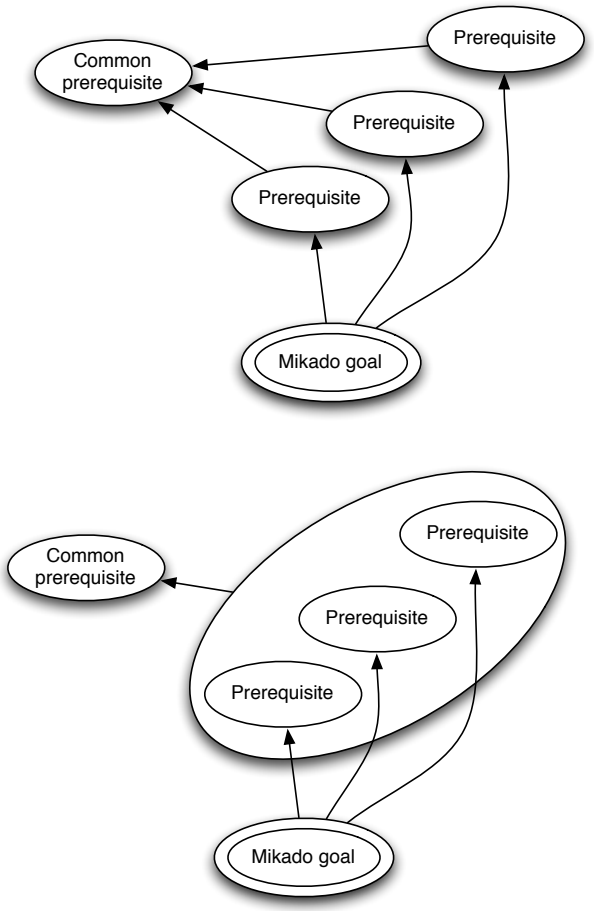


Figure 4.1: Having several nodes depending on one other node

The two graphs in figure *Fig. 4.1 (p.77)* are equivalent.

### 4.1.2 Prerequisites that apply to all nodes

Sometimes we want to perform something before we even start to create the Mikado Graph. Those prerequisites usually show themselves in that all nodes we draw point to that prerequisite. In those cases, we don't waste lines on the board and try to draw them in the same Mikado Graph. Rather, we create a separate Mikado Graph, or just make a list of those changes. Examples of such changes are general code clean-up, like removing unnecessary or unused code, commented out code, or applying a common code formatting template on all of our code.

### 4.1.3 Doing the same change for similar structures

When we draw the Mikado Graph and find that we have the same dependencies, or steps, to perform for several goals, we can simplify things a bit. Instead of repeating them, one after the other, we draw them in a separate bubble like in *Fig. 4.2 (p.79)*.

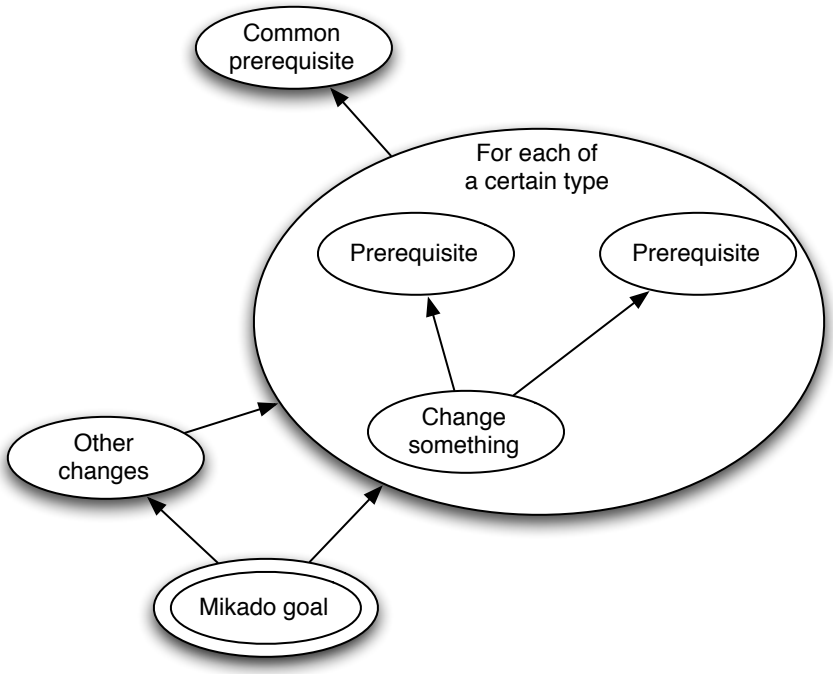


Figure 4.2: Doing the same change for similar structures

This way we don't waste paper and energy, and the intent becomes much clearer to the next viewer, which might be, in the future, us.

### 4.1.4 Dealing with several different goals at the same time

We prefer to work with one goal at a time, but if there are several different goals that needs to be dealt with simultaneously, those graphs ought to live side by side. Our experience is that problems with a system often have common root causes, and fixing one issue can solve

several problems at once. A Mikado Graph with several different roots can help us find those hot spots, like in *Fig. 4.3 (p.80)*.

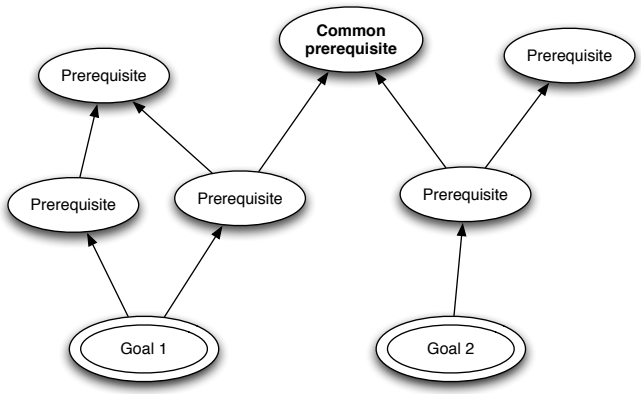


Figure 4.3: Several goals with common prerequisites

### 4.1.5 Splitting the Mikado Graph

Sometimes we run out of paper or whiteboard, or we just want to break out a part of the graph. In those cases, we in figure *Fig. 4.4 (p.81)*. The broken out part should not have any arrows back to the original drawing.

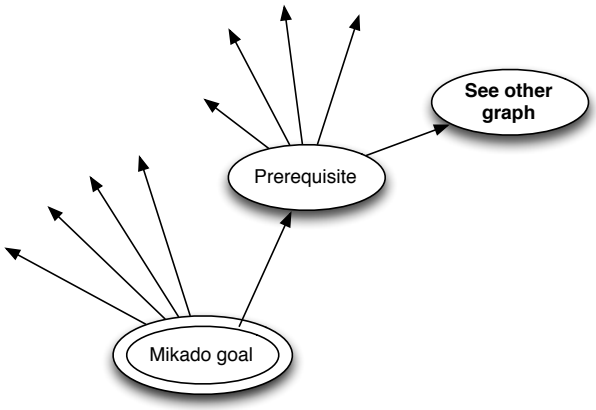


Figure 4.4: Splitting a Mikado Graph

### 4.1.6 Exploring options

When we have a problem that can be solved in more than one way, we probably want to explore our options. Then we split the dependency arrow, creating one for each option, like in *Fig. 4.5 (p.82)*. After that we travel along all branches until we have sufficient information and know reasonably well what the consequences are for choosing each option.

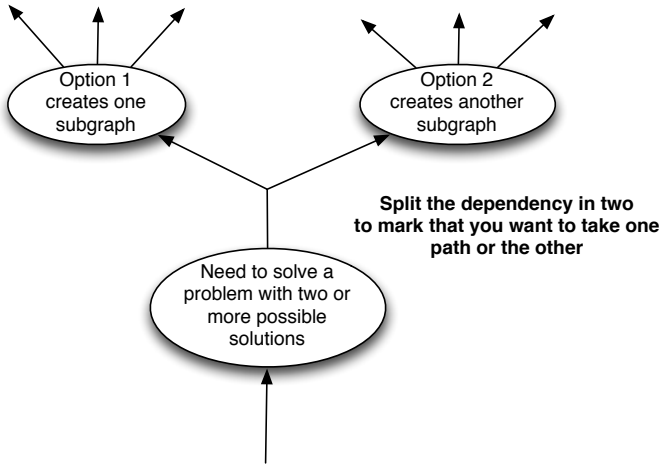


Figure 4.5: Exploring two different options for solving a problem

Exploring options gives us more information so we more easily can resolve design discussions where we disagree on the solution to choose. Typically when we cannot overview the consequences properly.

If we have other things we need to do anyway, we do them first and wait on the parts where we are not sure which path to take. Working with the system and the Mikado Method will help us build more knowledge about the system. When we have taken care of the other paths, we might have gained enough knowledge about the options to know which to choose. Sometimes it even happens that the problems we wanted solve with the options have already been dissolved.

## 4.2 Code patterns

When restructuring we often behave in the same way when we face similar problems. The following sections contains some generic, and some a few more specific, ways to restructure code.

```
public class Employee {
  private String name;
  public String greet() {
    return "Hello␣"+name;
  }
}
```

Listing 4.1: Initial Employee

```
public class Employee {
  private String name;
  private String getName() {
    return name;
  }
  public String greet() {
    return "Hello␣"+getName();
  }
}
```

Listing 4.2: Name field encapsulated in a method

### 4.2.1 Update all references to a field or method

When changing field or method implementations, there is sometimes the problem of updating all references to that field. An example is when we have a name string-field that is accessed directly in a class, but we want to change that to move the name field into a Person object, still contained in a field in the current class.

The process is

- Encapsulate field.
- Change implementation within the encapsulation method.
- Inline the contents of the encapsulation method.

In code, it would look like in *Listing 4.1(p.83)*.

First, we *Introduce indirection* on member `name`, resulting in *Listing 4.2(p.83)*.

```
public class Employee {
  private Person person;
  private String getName() {
    return person.getName();
  }
  public String greet() {
    return "Hello␣"+getName();
  }
}
```

Listing 4.3: Get the name from a Person delegate object

```
public class Employee {
  private Person person;
  public String greet() {
    return "Hello␣"+person.getName();
  }
}
```

Listing 4.4: Inlined getName() method

In *Listing 4.3(p.84)*, we change the one usage of `name` to use an introduced `Person` object instead.

After that, we do *Inline method* on the `Employee.getName()` method in *Listing 4.4(p.84)*.

Granted, for this simple code, it really doesn't matter, but in some cases a member or property can be used in a lot of places.

The same technique can be used when changing calls to a method. We *Introduce indirection* in all places the method is used by wrapping the originally called method in a new method. Then we change the implementation in the new wrapping method, and then we *Inline method* as above to introduce the change in all places.

### 4.2.2 Frozen partition

When we have external users of some parts of the code, and we want to be backwards compatible to those users, we can freeze that part of the code.

When making extensive changes to a codebase over time, it is easy to accidentally change the code for people that are downstream of us and the product of our work. Automated tools like refactorings and regular expressions are powerful tools, but sometimes they can be applied on a bit to much of the code, changing parts that shouldn't change.

Sometimes, it is just enough to be aware of what cannot change. To make this easier, we can move the part of the code into a particular project or directory, to faster see if the frozen parts are changed.

Another strategy is to write reflective API tests, tests that are not affected by refactorings, that verify the signatures, classes and interfaces that must not change. A variation of this is to place the API tests in a project that is not affected by the automated refactorings. This will also provide excellent documentation for anyone using the API on what parts that will be more stable.

### 4.2.3 Mimic replacement

Every now and then we need to shield off, or replace, an existing framework, but we have a problem when that framework is used almost everywhere the codebase. This makes it very difficult to make detailed changes in all the places where the framework is used. Instead of changing the users of the framework to fit the replacement, we change the replacement to fit the users.

In this situation we can develop a *mimicking version* of the framework we want to replace. This mimicking version will not have all the functionality of the framework, but only the functionality that is actually used in our application is implemented.

When the mimic is implemented, a switch is made from the old framework classes to the new mimic classes in one of two ways. Either, we let the mimic have exactly the same names and package names, or namespaces, as that of the old mimicked framework. Or, we let it have the same names except for the package. When we switch, we change the import or usage statements, just linking in the new framework instead of the old. By doing so, a framework can be replaced without changing any of the code that uses the framework. The mimicking version must

of course also do the work that the framework did, possibly by relaying to a new framework.

A variant of this is when we are pleased with the existing framework, but need an interception point to be able to alter behavior for testing, or to introduce alternative implementations. The mimic version can have the option of just forwarding to the old framework, almost in a one-to-one mapping, or to having mocked or stubbed responses.

This technique cannot always be used, since it might be impossible, or infeasible, to create the mimic structure.

### 4.2.4 Replace configuration with code

Many software developers, and others too, believe that applications should be extensively "configurable" in or after deployment. The reason put forward is often that "it is better to have a flexible application". The real reason is more likely that no one knows how the software will be used, or no one dares make the decision. This usually leads to complex code and complex, extensive, configuration files. These configuration files often stand in the way of navigating, understanding, and changing a codebase. They also add complexity in that all relevant configurations should be tested, and irrelevant configurations should be signalled as wrong by the application. This is tedious work that adds little value, since most of the possible configurations are usually left unused.

There are two types of configuration parameters. One is configuration for parameters that don't change after the code is built. The other is for parameters that must be editable when deploying or starting the application, or even while the application is running.

The first category of configuration is better built into the application as configuration packages, code that provides the parameters as constants or even as objects for the application. This means that we build a distributable for each variation we have.

If we have a lot of this type of configuration, we might make up time by generating the code from the configuration files and throw the configuration files away. If we have too many variations, this is just a sign that we cannot handle the complexity we have created, and the solution needs simplification.

Configuration parameters that must be variable in and around runtime are the parameters that actually need configuration files. When our environment forces us to use configuration files, we make sure the intersection with them is a limited as possible. This will greatly simplify any future and immediate refactoring work.

## 4.3 SCATTERED CODE PATTERNS

Packages, or projects, and their dependencies create restrictions on what code can reference what other code. While this is a thing that can be used to our advantage, in the face of change they may also pose a problem; we cannot move or access code where we want to.

A common scenario when dealing with legacy code is when the application at hand is split into packages, but little care has been put into what code goes in what package. Code that should be packaged together is scattered across the packages in the codebase. The packaging could break all the packaging principles described in *Principles of packages (p.125)*.

The problems usually manifest themselves in a couple of different ways. We might have *difficulties to find the code we are looking for* because it is not given by the package names what goes into the packages.

It also shows in *dependency problems with libraries or other packages when moving things around*, typically when we are using the naïve approach. If the logic is spread across the codebase, there is usually a nasty web of dependencies to keep everything together.

Another symptom is when *all packages have references to all and the same libraries*.

A typical Mikado scenario is when we need to extract a part from such a codebase to create a library, to use in several places. We will describe some common ways to move code around and break out parts into new packages.

### 4.3.1 Merge, reorganize, split

When dealing with scattered code it is often beneficial to merge the packages, to temporarily avoid blocking dependencies, and thereby get more slack to implement the changes we want. We often merge the packages, reorganize, and then extract the new library, as illustrated in *Fig. 4.6 (p.89)*.

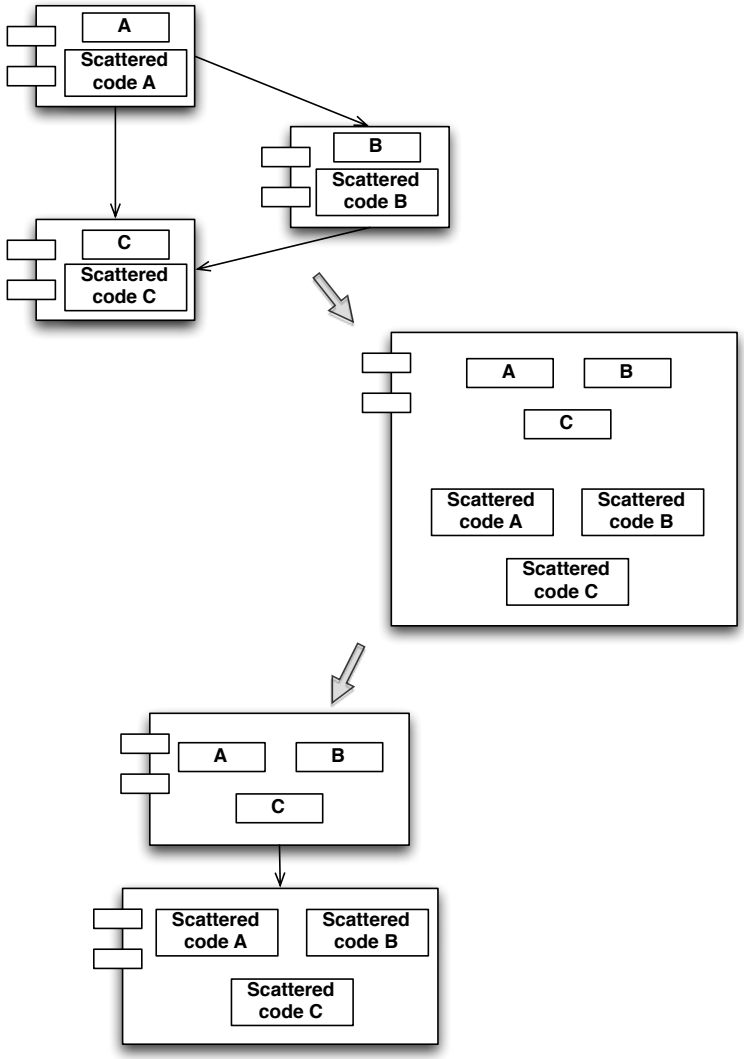


Figure 4.6: Merging code to split out in another way

After we have merged the packages, we can group the scattered code and naïvely move it to the new package. The compiler, or our tests, will then tell us if we missed some dependency that we need to take care of first. Typically, we get problems with dependencies that reference parts of A,B or C, and we cannot reference back to the A-B-C package because that would create a cyclical dependency.

The usual solution is to use the *Dependency Inversion Principle - DIP (p.121)* to insert an interface, like we did in the example in *Resolving the circular dependency (p.37)* in *Ch.2 An example in code (p.23)*. The created interface is packaged with the scattered code and moved into the new package.

After such a change, there is more information available on how to structure the software. If we need to further extract functionality from the A-B-C packages, we can do that in a similar fashion. If the packages don't have to be split for any reason, we don't split them. As stated in *Principles of packages (p.125)*, we don't split packages when we don't have to.

The Mikado Graph can look in many different ways from this move, but a typical graph might look like *Fig. 4.7 (p.91)*.

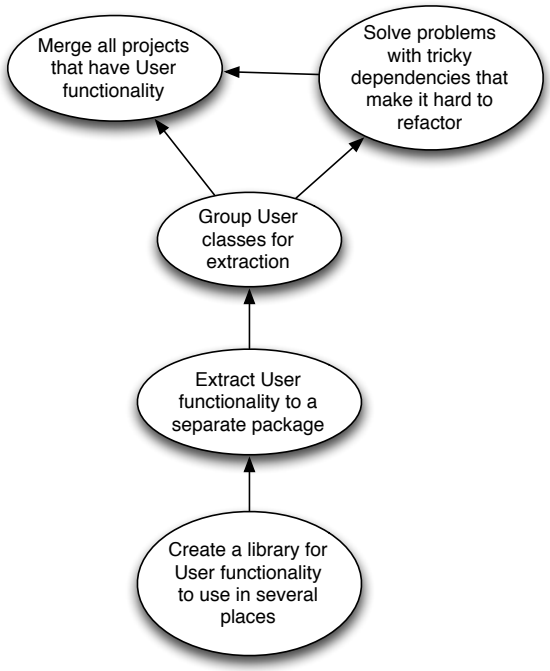


Figure 4.7: Extracting a library with user functionality

In most cases, it is fairly straightforward to merge two packages. There is rarely more to it than to merge them and let them have the union of their respective libraries, and let all packages that depend on any of them depend on the new package.

*But isn't this just a way to hide the dependency problems?* Well, yes, in a way. This is often not the permanent solution, but more a stepping stone to something better. But then again, if we can make this move and avoid having these dependency problems as a critical path blocker, we could just leave them there until they enter our critical path again.

One way to see this merge-move is as a common trick in mathematics; to transform a problem into another dimension, to solve the problem in

the dimension where solving it is easy, and to then transform the result to the original dimension. Merging the projects is a little bit like transforming the problem to another dimension, and then we transform it back to a better solution by extraction the packages we need.

Sometimes, the packages cannot be merged. Then we have to use another tactic to extract the new package.

### 4.3.2 Move code gradually to a new package

If the projects cannot reasonably be merged, we can move the scattered code directly to a new package, like in *Fig. 4.8 (p.93)*.

In the first step in the figure, we have already created a new empty package and added that package as a dependency for the other packages. In the second step we move the scattered code from package C, the leaf in the package dependency graph, to the new package. By starting in the leaf C, and then take B and then A as shown in the last step, we avoid creating some cyclical dependencies that could occur if for instance *Scattered code A* has dependencies to *Scattered code B* or *Scattered code C*.

By naïvely moving the code to the new package, we might find some dependency problems that we have to take care of. As always, we add that information to the Mikado Graph.

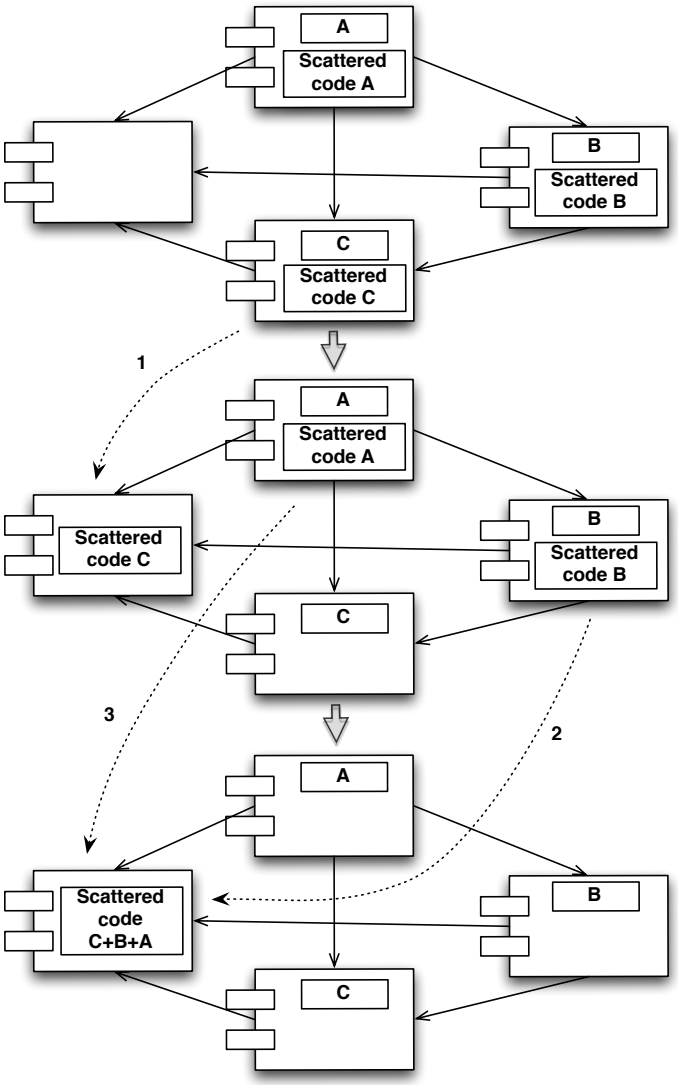


Figure 4.8: Move code gradually to a new package

These are rather simple package dependencies, to understand the principles. In reality, these dependencies can be much more extensive. Analyzing the dependencies to find the leaves can be helpful, but tedious. We often use a combination of analysis and the naïve approach when trying to figure out what to do.

### 4.3.3 Move code in the dependency direction

In some cases, it is possible to just move scattered code in the dependency direction, and then extract the new package, like described in *Fig. 4.9 (p.95)*. The first step is the original situation, and for each step we move the code along the dependency direction, until it all is in one package, and we can move it to a separate package altogether.

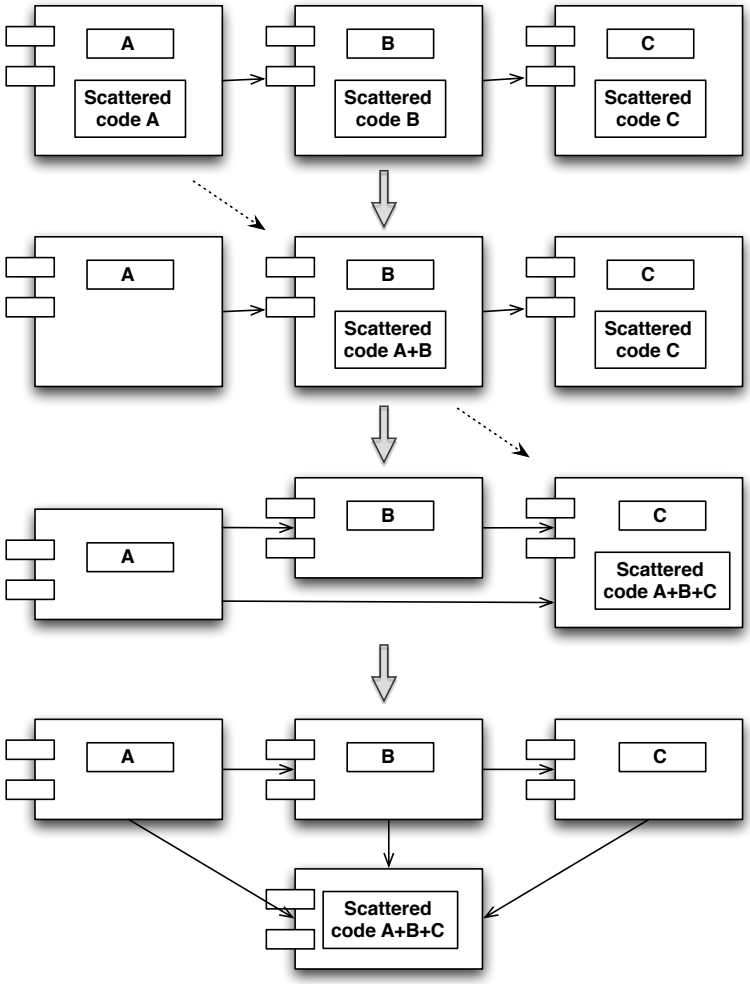


Figure 4.9: Move code in the dependency direction

When we move the scattered code in the dependency direction, the code will continue to compile as long as all dependencies and transient

dependencies are still in place. Any dependency problems we get from moving code are dealt with in the same way as in *Merge, reorganize, split (p.88)*, typically using dependency inversion.

This approach, to move the code in the dependency direction, is not always possible, since it cannot be done when the dependencies are messy. When possible, it is a rather simple operation. We might have to change the dependencies of any code that refers to the A or B package when the code is moved out of there, to point at the new packages where the code resides.

### 4.3.4 Creating an API

Breaking out the functionality as in the previous examples might have made our solution comply with the cohesion principles, or enabled us to do what we want, but there might still be some work to do. Maybe the code extracted contains too many classes, or too much information, for the consumer of that library. Maybe parts of the implementation shouldn't even, or couldn't even, be provided. Maybe the implementation of the package changes a lot, but the used parts, the application programming interface (API), of that package is rather stable. Maybe there are to be several implementations of that API.

In these cases, it might be a good idea to create an interface package for the API, and an implementation package to keep the implementation of the interfaces. This works a bit like the dependency inversion principle, but for packages. The consumers of the package program against the interface package, and in runtime the implementation package is provided. By doing so, the packages particularly adhere to the Stable Dependencies Principle *SDP (p.131)* and the Stable Abstractions Principle *SAP (p.133)*.

As often, we can use the naïve approach to try and move out the implementation part to see where it breaks, and use that information in the Mikado Graph. We usually separate, or extract, the necessary interfaces and classes. When they are the only ones directly used, we can move the implementation part to a separate package. A typical Mikado Graph might look like in *Fig. 4.10 (p.97)*.

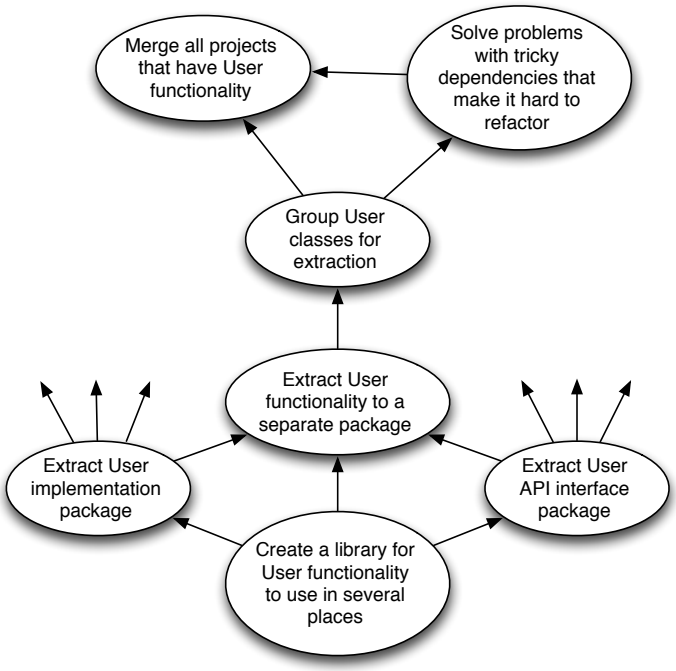


Figure 4.10: Extracting a library with user functionality, including an interface and an implementation package

### 4.3.5 Temporary shelter

As described above, one type of problematic code is when we have related code scattered all over the codebase. The above examples are more related to when classes are scattered, but sometimes the scattered code is on a more fine-grained level, more like methods here and there or even blocks of code within methods. This could mean duplication, but not necessarily so, and not only so.

Instead of continuing down the same path, we try to find a couple of core areas of the scattered code. Examples of this can be User,

or Payment, something that fits the domain at hand. We choose an existing class or create a new one for these core areas. These classes will be *Temporary Shelters* for the functionality. Then we try to move all User-related functionality to the Temporary Shelter. This might be ugly, but it is just a stepping stone. In figure *Fig. 4.11 (p.98)* there is a sketch of what this might look like in a Mikado Graph.

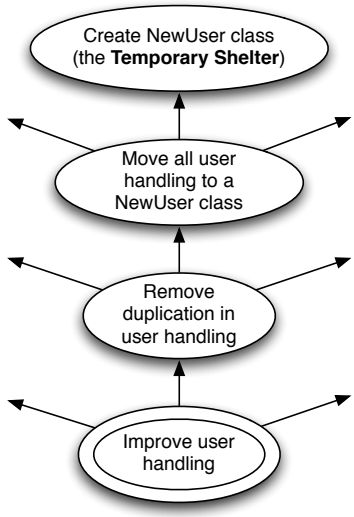


Figure 4.11: Move functionality to a Temporary Shelter before performing a change

After awhile, we are likely to see patterns of similar code in those rather ugly Temporary Shelters. We then try to remove the duplication by extracting those common sections. One way is to create a base class, and add variations in subclasses or in separate functions. Sometimes it is better to break out separate concerns to new classes, and delegate to those classes.

Hopefully, the previously scattered blocks of code end up in a comprehensive and cohesive structure.

## 4.4 When we cannot check in small changes

Even though it is desirable, sometimes it just is not feasible to reach a goal taking small steps that can be committed to a versioning system continuously. It could be that other developers are working on the code, and our changes will result in merge conflicts that are very tedious and disruptive to resolve. It could also be that making half a change would introduce a pending state where there are two ways of doing things, which might be confusing at best, and error prone at worst.

We have used the Mikado Method in such a situation, to practice our refactoring over and over again until we have felt confident that we can perform the change in the time-slot available. This includes heavy use of automated refactorings and refactoring scripts to speed up the process and relieve the drudgery of doing the same refactorings over and over again.

> Once upon, Melinda performed a large, single-step, refactoring when she and a colleague were to replace old XML with pure code. The XML glued together business logic and caused some serious problems for the developers. The XML had been a big concern for about five years, but no one had dared touch it.
>
> The reason for the change was to simplify code navigation and enable further refactorings, which were made almost impossible due to the XML that was like a wet blanket over the entire codebase. There were only a few tests in place, partly because the framework made it hard to test at a good level and partly because of historical neglect, and there was no time to cover the application in tests. In addition, the change had to be made in a single step, or the development in the project would grind to a halt during the change. This was just not an option.
>
> The two started with a Mikado Graph for the change. They made some changes, saw what broke, extended the graph and reverted. After almost every revert they updated to the latest version of the code.
>
> Some of the prerequisites in the Mikado were running scripts that

generated code from the existing XMLs. Some of the prerequisites were regular expression replacements in the IDE. Some of the prerequisites were standard refactorings that the IDE provides.

Some of the prerequisites were leaves that could be checked in and ticked off in the Mikado Graph, but most of the changes actually required replacing all the XML with code, and this could not be checked in until it all worked, and all XML-files were replaced.

To perform such a change without serious merge conflicts, it is necessary to have everyone check in their code, but in this case the development could not be halted. The solution was that the refactoring should be performed one evening, and everyone had to check in the code before 5 o'clock that afternoon.

In the days before, they had practiced and refined the Mikado Graph to be able to make the change as swiftly as possible. They had probably rehearsed parts of the graph 20 or 30 times when it was time to actually perform the change.

At 5 o'clock, the refactorings began, this time for real. Scripts were run, code was generated, and refactorings were performed across the codebase. At 7 o'clock, they realized that they had, after all, missed a special case, and had to revert the code and start over. After a couple of hours, and no further mishaps, they had performed the refactorings, and all the XML code was gone. They verified the solution by running the few scenario test cases that existed, and then they checked in. At about 1 o'clock in the morning they could leave the building.

In spite of the massive change, six months later there were just a couple of minor problems reported from the refactoring. This would have been hard to achieve in such a short time in any other way.

## 4.5 Summary

Jake: OK, are these are all the patterns there are, or...?

Melinda: Oh, no, these are probably just a small sample of all the patterns that are out there.

Jake: Then I shouldn't limit myself to the patterns in this chapter?

Melinda: That is correct. You will probably find your own patterns after a while.

Jake: Sweet. I also liked the idea of practicing your restructurings a couple of times before actually doing them.

Melinda: That is a good pattern when you can't afford having the code in a pending state at all.

Jake: It is like letting waves of changes flow through the code, learning a new piece every time.

Melinda: Repetitive learning is often a good thing.

Jake: Especially when you're a bit thick in the head, like me.

Melinda: Aren't we all a bit like that from time to time?

Jake: True. At those occasions, it is good to have a safety net like the Mikado Method!

## 4.6 Try this

- If you have ever tried the Mikado Method, did you make any changes to the graph to suit your specific needs? What changes? Why?
- What were the three most interesting patterns? Why?
- Can you think of any occasion when any of the patterns in the chapter would have been useful?
- Have you got any patterns of your own?

# Chapter 5

# Guidance for creating the nodes in the Mikado Graph

| | |
|---|---|
| Jake: | Hey, Melinda. I've made a few restructurings now, but sometimes I seem to wander off in the wrong direction depending on the choices I make. |
| Melinda: | I see ... are you familiar with any software *design principles*? |
| Jake: | Errrr, avoid duplication, is that a principle? |
| Melinda: | Yes, that is one, and there are several others. |
| Jake: | Ok, but what difference do they make? |
| Melinda: | They help me a lot in my refactorings, since they provide me with a force field that usually take me in a good direction. |
| Jake: | A force field?! Ha, just what I need! Do you have a spaceship to go with that? |
| Melinda: | You have to travel the design space in your mind, my friend. But I'll be happy to join you on the journey. |

## 5.1 GOOD DESIGN

In software development, there are so many situation we can end up in that it is impossible to describe them all. A subset of those situations are working solutions. A subset of the working solutions is what could be called "good design," which is illustrated in *Fig. 5.1 (p.104)*.

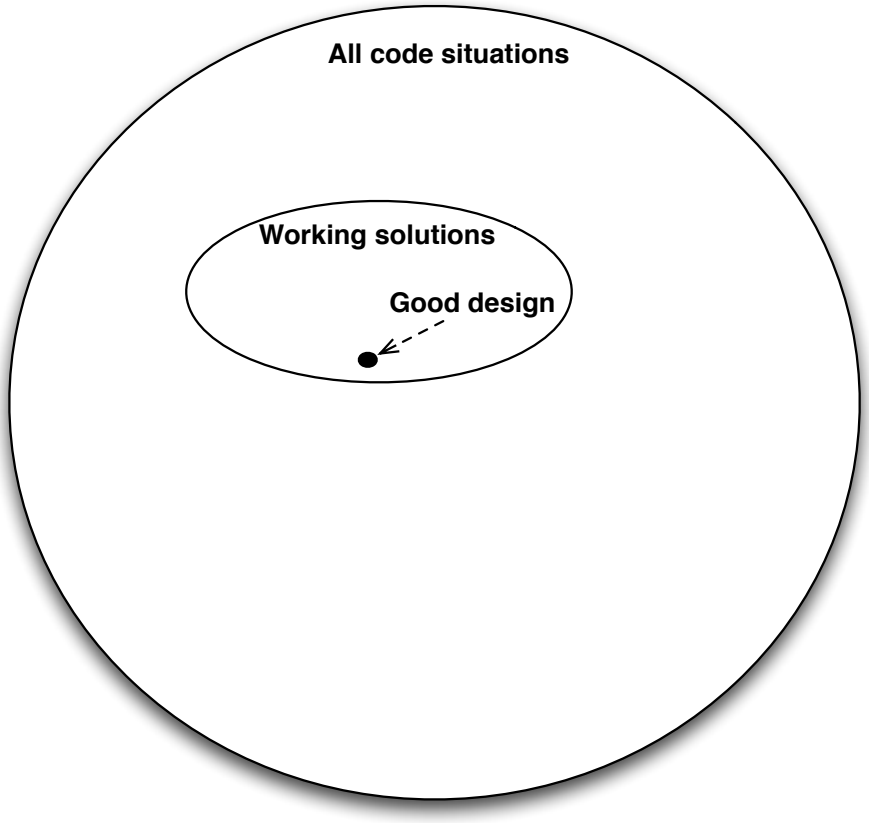


Figure 5.1: The subset of good design

Interestingly enough, code that is attributed with the description of having a good design in one case might be over-engineering in another. Good design is nothing absolute, but rather very context dependent. The sense for what constitutes good design normally takes years to develop, through countless hours of discussing code and architectures with other people of different backgrounds, and especially getting critiques for our own code, as well as giving critiques to others to see what their response is.

## 5.2 Design principles - the forces of nature for software development

Design principles can be seen as the forces of nature to create well-designed code. In nature, there are forces like gravity, electric forces, magnetic forces, and more. We can fight against them, like working against gravity by lifting a heavy object off the floor, or up a hill, but we have to consume energy in order to do so. Sometimes we have to consume energy just to balance a heavy object in an elevated position, like holding a barbell over our head. Just like gravity, the design principles can be used to pull our code in the direction that requires less energy to create and maintain. We just have to know how they work in order to use them.

Sometimes, the design principles will pull in different directions. In those cases we have to balance them for our specific solution. This is a bit like having a maglev train that uses magnetism to oppose the gravity force, in order to avoid friction forces that are present in a conventional railway. By finding the right balance, great advantages can be achieved.

## 5.3 The landscape, the path and the goal

To illustrate, but with a great deal of simplification, the design space can be seen as a landscape with hills and valleys, where the design principles are pulling downwards.

When we are writing or improving a system, we want to have a goal, a

place where we want to go. This is our Mikado Goal. The goal is like a specific place in the landscape where we want to end up. Around that place there are both high places that requires more energy, and valleys that require less energy, to get to and to stay in.

The Mikado Graph is the path from where we happen to be at the moment, to the desired area. The decisions we take along the way will affect how difficult our path will be to travel, and how good a place we will end up at within that area.

When we create the Mikado Graph, it is like traveling in such a landscape, but backwards. We start in the goal, and backtrack to where we are, to find the first couple steps we can take. Sometimes, we don't know what the landscape looks like, nor what will be a good path to travel. The Mikado Method helps us find a path, and adhering to design principles will help us find good paths and footsteps, to a better designed solution.

The following sections will give brief overview of some of the more important design principles. For a thorough description of these principles, please see the referenced books.

## 5.4 Don't Repeat Yourself - DRY

The principle was first stated in Dave Thomas and Andy Hunts book "The pragmatic programmer" [2] as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.".

The obvious interpretation of this rule is that code should not be copied and pasted into a system. When we copy-paste code, we create an implicit and very hard to find dependency between the two pieces of code and the reason for which they exist. When the reason for their existence changes, or if there is a bug in the duplicated code, we must change the code in both places. This is an error-prone operation, since it is easy to miss a duplication somewhere in the code. It can also be a very tedious operation, since we must do an exhaustive search in the codebase to rule out the risk of having duplicated instances. This is obviously much more work than just changing code in a single, non-duplicated, place.

Melinda remembers:

I was in a team which developed a web based survey and reporting tool. There was a lot of email going out from this system: reminders, invitations and passwords that needed to be reset etc. So we wanted the email addresses to be correct obviously. I was pretty junior at that time and this was one of my first big software projects. I had just been introduced to regular expressions, which can be a challenge at first. For this particular problem I thought it would be a perfect match though.

So, I implemented a regular expression that verified an email address. Actually, I must admit I browsed the internet and found what I thought was a working piece of code. But of course there was a flaw in it. One of it's problem was it couldn't handle IP-addresses. The specification for email addresses allowed them in the form name@127.0.0.1, for instance, which I didn't know.

This became apparent after someone, probably a computer geek like myself, tried to add an address in this particular form. A bug report was written and I fixed the verification of email addresses ... at least I thought so.

One week later a very similar bug was reported. *The system can't handle emails which are in the form xyz@127.0.0.1*. I thought to myself that *this must be a joke*, or *someone is running an old version*. I just had to verify this and started doing some testing, and to my surprise, there was a problem.

After locating the code that handled sending emails I found the same buggy verification of email adresses I had previously fixed. Only it was in another module.

Copy-pasted duplication is fairly simple to remove by just extracting a method for the duplication. If there are smaller variations for the implementation, we can solve those by providing a parameter to the method, or create two or more methods that share the same base.

On the not-so-obvious side, the same knowledge can be captured in completely differently looking implementations, but it can still be con-

sidered duplication. The latter, not-so-obvious duplication, is often harder to find and remove. Several pieces of code that perform the same process flow, but with different business logic, is a perfect example of that.

A typical Mikado scenario for the DRY principle is when we need to change the logic of something, like *user handling*, and that logic is scattered and/or duplicated all over the codebase. This is a common situation, and is discussed a bit more in *Temporary shelter (p.97)*.

## 5.5 Low coupling, high cohesion

Good, maintainable and stable software often exhibits *low coupling and high cohesion*, as described by Larry LeRoy Constantine [11].

*Low coupling* means that there should be as few couplings, or dependencies, between different parts, like classes and packages, as possible. The connections should also be of the kind that ties the parts together in the most stable manner possible.

*High-cohesion* means that the contents of the parts should have as much in common as possible. This also stabilizes the system, in that changes are contained and cause as little ripple effects through the system as possible.

These two recommendations have to be balanced, and there is usually no perfect system.

## 5.6 The S.O.L.I.D. principles of class design

In the book "Agile Software Development-Principles,Patterns and Practices" [12], Robert C. Martin describes a set of principles that together should guide a software development effort.

One set of principles are the class design principles. The idea of these principles is to create classes that are as stable as possible in the face of changes. This means that the principles guide us when it comes to

what functionality to group, what interfaces to put on that functionality, and what dependencies between classes to encourage.

S.O.L.I.D. is an acronym-of-acronyms, and stands for **S**RP, **O**CP, **L**SP, **I**SP and **D**IP. The latter acronyms will be explained in the following sections.

### 5.6.1 Single Responsibility Principle - SRP

*"A class should have one, and only one, reason to change."*

This principle states that we should not give a class several responsibilities. So, what is a responsibility? In some cases it is perfectly clear. An example would be a `User`-class that, except for its name, also knows about authentication, presentation and storage. These are four different responsibilities, and should be in four different classes.

In other cases it can be more diffuse. Should user validation logic be in the `User` class or in a `UserValidation` class? It depends. If the user logic only changes with the validation logic and vice versa, then they should be in the same class. If, for instance, the validation logic changes in different contexts, then it should be in a separate class, or in separate classes.

The idea is to find a good home for functionality instead of just cramming it into the class or module we just happen to be in at the moment.

So, why is the SRP a good idea? One good practical reason for the SRP is that it will be *easier to read the code* on a high level and actually understand what each part is doing. Another good reason is that *code that is not changed is unlikely to suddenly break*. Having several responsibilities, i.e reasons to change, in the same module will require that module to change every time any of the responsibilities require a change, thus increasing the risk of accidental errors and instabilities. A third good reason is that it also *decrease the risk of creating bad dependencies* between the implementation details in the different responsibilities, making it hard to refactor that module when needed.

One extreme way of breaking this rule is by having a *God-class* [13] that contains all important logic and does everything. At the other end of the rope are systems where our classes are split to the degree that

responsibilities that belong together are separated in different classes. This is not good either, and will create unnecessary complexity and more classes than needed in the system. As so many other times in software development, we have to strike a balance, avoiding these extreme cases.

### *SRP in the Mikado context*

Often when we have used the Mikado Method, there is a piece of functionality that we want to use in several places, but it is entangled in other logic at its current location.

An example would be Reuse the entangled login functionality. To do that we probably have to Break out the login responsibility to a new class, resulting in a Mikado Graph like in *Fig. 5.2 (p.111)*.

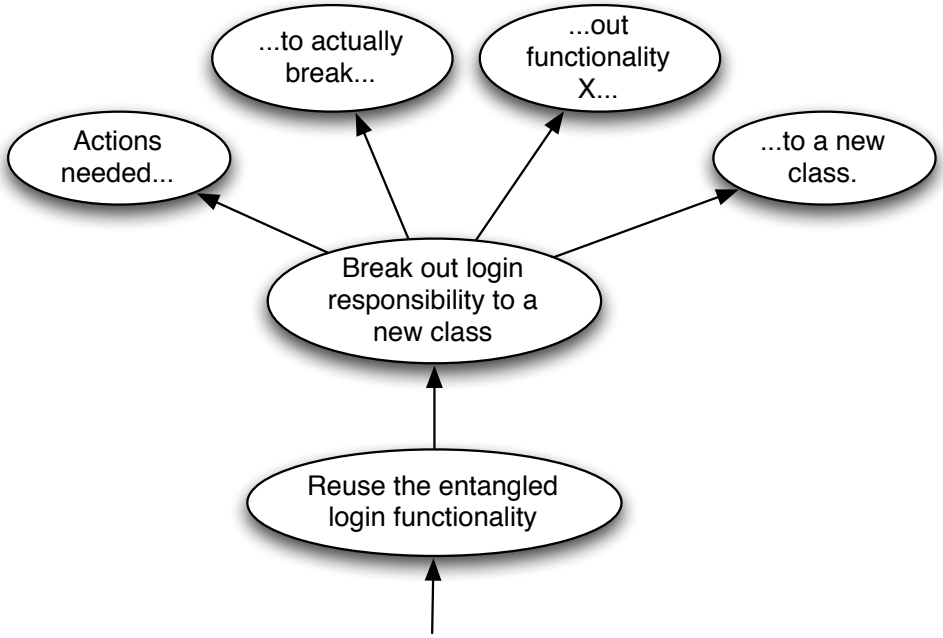


Figure 5.2: SRP in a Mikado diagram

### 5.6.2 Open-Closed Principle - OCP

*"A class' behavior should be extendable without having to modify it."*

In its purest form, OCP tells us that when adding a feature to a codebase, we should not have to alter any existing code, just add new code.

Among other things, this principle effectively rules out programming variations using conditionals, such as if-statements. Misused if-statements is usually a big hurdle when improving large and unwieldy codebases. When making changes to a large codebase, we always strive for removal of variations coded with conditionals.

```
. . .
public static final int FIVE_CARD_POKER = 0;
public static final int INDIAN_POKER = 1;
. . .
public class CardGameEngine {
    . . .
    public void deal(Game game, User user, Dealer dealer) {
        if(game.getType() == FIVE_CARD_POKER) {
                user.setCards(dealer.deal(5));
          } else if(game.getType() == INDIAN_POKER) {
                user.setCards(dealer.deal(1));
          } else {
                // else what?
          }
    }
}
```

Listing 5.1: An OCP-breaking card game engine

This is a small example where a flag is used to signal what game is played in a card game engine, shown in *Listing 5.1(p.112)*.

When the engine deals the cards, it checks against a type flag on the `Game` class to decide what to do, in this case it deals the correct number of cards. This means that when adding a game we must change the class that holds the type flag definitions, together with all if statements in the code that are similar to the one above. This means breaking the OCP. Even worse, when there are several rounds of dealing in a single game, even more flags must be added.

Further, there is no guarantee that there is logic for all possible game types, which might result in a runtime exception, the "else what?" part.

Another version of this code would be the card game engine in *Listing 5.2(p.113)* that doesn't break the OCP.

The logic is now encapsulated in the abstraction of the different `Game` implementations. The `CardGameEngine` knows only about how to connect the Game, the User and the Dealer. It knows nothing about the games.

Adding another game now would only involve creating a new implementation of the `Game` interface. The deal logic would not have to change in any of the existing classes.

```
interface Game {
      void deal(User user, Dealer dealer);
}
public class FiveCardPoker implements Game {
    . . .
    public void deal(User user, Dealer dealer) {
        return user.setCards(dealer.nextCards(5));
    }
}
public class IndianPoker implements Game {
    . . .
    public void deal(User user, Dealer dealer) {
        return user.setCards(dealer.nextCards(1));
    }
}
. . .
public class CardGameEngine {
    . . .
    public void deal(Game game, User user, Dealer dealer) {
        game.deal(user, dealer);
    }
    . . .
}
```

Listing 5.2: OCP compliant card game engine and the extracted functionality

Responsibilities for each of the different games are moved into their own respective classes. That way we will be honoring the SRP as well.

Conditionals are a common OCP breaker, and the above is a common way of resolving that problem. It is called *Replace conditional with strategy* in Martin Fowlers book *Refactoring* [4].

### *OCP in the Mikado context*

Say that we wanted to add a new game, *Whist*, in the CardGameEngine. One way would be to alter all the if-statements across the codebase. As discussed, this is generally error-prone, and it makes the situation even worse by the increased amount of complexity.

The other option is to first make the CardGameEngine OCP-compliant, and then add the new game. In order to do that we need to move the logic in the if-statements into classes, like we did in *Listing 5.2(p.113)*, and then add the new Whist game. This could be seen as doing a *Refactoring in advance (p.146)* step, where we first refactor the code and then implement the new logic.

A Mikado Graph of this would typically look like in *Fig. 5.3 (p.115)*.

Why don't we chose to first add the new game and then refactor? We could do it that way, but if we start adding the logic in the old solution, we would first have the problems of adding it correctly in all the places, which would take a lot of time. Second, we would have an even worse problem when we are to extract all that if-statement logic to its own classes, and thereby make the solution OCP-compliant.

When we start with preparing the code with a refactoring, our experience is that we finish faster, on average. In addition, we are not even tempted to leave the code after adding all the new if-statements, skipping the refactoring step. As discussed in *Red-green-REFACTOR (p.195)*, it is very important to actually do the refactoring step to avoid building up complexity.

At the occasions when we cannot see any good generalization, we might have to wait until after the new functionality, a game in this case, is implemented, as discussed in *Waiting for the right abstraction (p.205)*. These are usually cases where there is not too much complexity built

up anyway, and another if-statement won't make the transformation any harder.

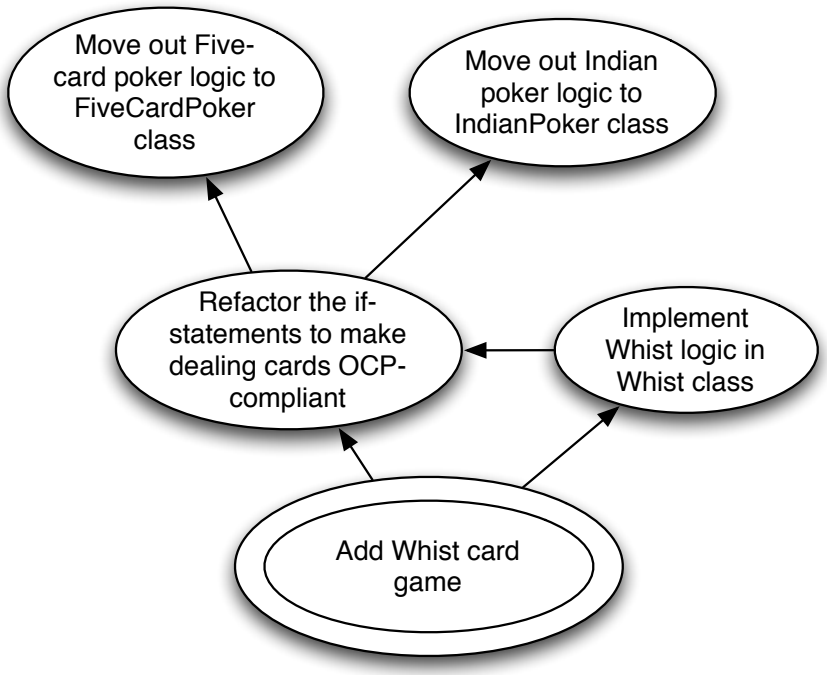


Figure 5.3: A Mikado Graph for adding the game of Whist to the CardGameEngine

### 5.6.3 Liskov Substitution Principle - LSP

*"Derived classes must be substitutable for their base classes."*

A sub-class to an abstraction should be possible to use anywhere the abstraction is used. A classic example to illustrate this is the Square-Rectangle problem. Lets take a class `Rectangle` as in *Listing 5.3(p.116)*.

```
public class Rectangle {
        public Rectangle(int height, int width) { . . . }
        public void setHeight(int height) { . . . }
        public int getHeight() { . . . }
        public void setWidth(int width) { . . . }
        public int getWidth() { . . . }
}
```

Listing 5.3: Rectangle implementation

```
public class Square extends Rectangle {
        public Square(int side) {
                super(side, side);
        }
        public void setHeight(int height) {
                super.setHeight(height);
                super.setWidth(height);
        }
        public void setWidth(int width) {
                super.setWidth(width);
                super.setHeight(width);
        }
}
```

Listing 5.4: A Square that inherits the Rectangle

Then we find a use for a `Square` object, and want to add that. A square is a special case of a rectangle, so subclassing `Rectangle` might seem like a good idea, as in *Listing 5.4(p.116)*.

The problem here is that the contract of the `Rectangle` says we can change the sides independently. If a method takes a `Rectangle` and changes its height, it would expect that the width is unaltered. That is part of the contract of the `Rectangle`. With the implementation of square above, the *derived class is not substitutable for the base class*, hence it breaks the Liskov Substitution Principle.

There are at least three ways of breaking the LSP. One is that the abstraction used is wrong, which could be said for the Rectangle-Square above. Another is that a subclass overrides a method in a way that is inconsistent with the protocol of the abstraction, often in complex inheritance hierarchies. A third possibility is that another method takes

a superclass for an argument, and then performs totally different actions depending on the implementing subclass.

### *LSP in the Mikado context*

Finding code that breaks the LSP is not that common, but it happens. We haven't found any particular Mikado patterns for LSP problems, so we have to look at the particular constraints of the situation and build our graph from that, just like we would normally do.

## 5.6.4 Interface Segregation Principle - ISP

*"Interfaces should be fine-grained and client specific."*

We don't want to expose more functionality than is needed for a client. When we have several clients with different needs, we create one interface for each type of client. If a client needs more at some later point in time, we extend the interface at that time. All clients should be on a *need to know basis*.

When this rule is broken, classes expose a fat interface, thus creating implicit dependencies between the unused members of the fat-interface-class and its users. This adds to the complexity of the codebase, decreases stability and also makes the system more difficult to refactor. Instead, we try to find seams in the application where the connecting interfaces can be kept to a minimum. This will minimize the risk of having inappropriate relations between different parts of the system.

*Fig. 5.4 (p.118)* is an example of an interface that has become too large.

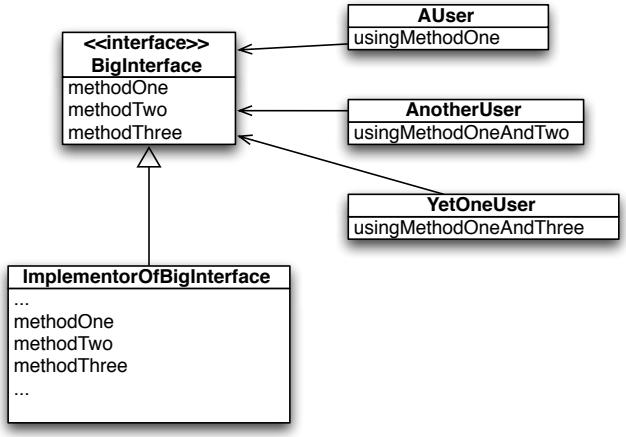


Figure 5.4: An unnecessarily large interface

An interface can be broken apart in different ways, for instance with a common base interface, as in *Fig. 5.5 (p.119)*.

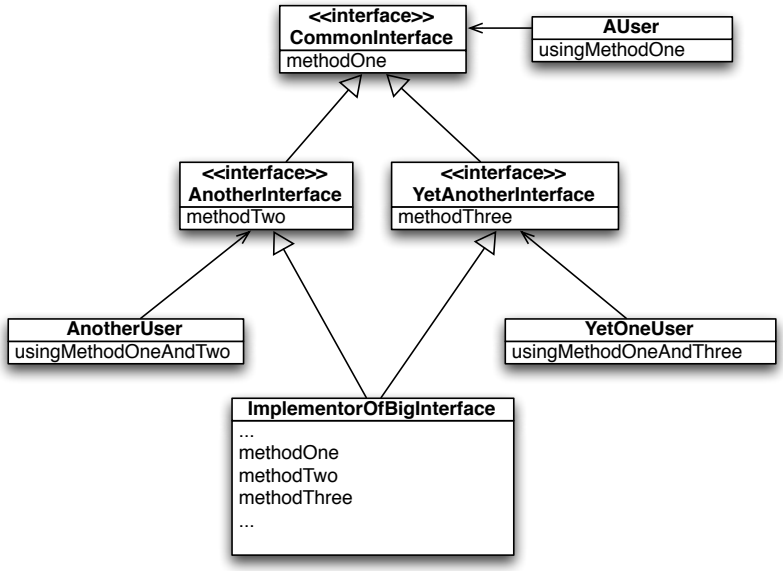


Figure 5.5: The large interface broken apart with a common base interface

Another option is to have a flat structure of interfaces if there is no common component in the original interface, as in *Fig. 5.6 (p.120)*. This is also useful when we don't want to have that for one reason or another.

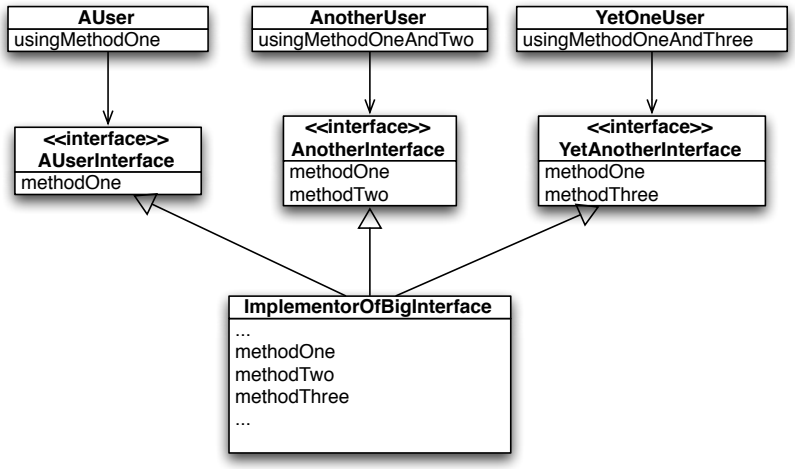


Figure 5.6: The large interface broken apart in a flat structure

### *ISP in the Mikado context*

Problems with too extensive interfaces are often discovered when we try to split up functionality, perhaps moving parts of it to new packages.

We can always experiment by naïvely removing one method of the big interface at a time to see where it breaks, and thereby build knowledge about the application. That knowledge will let us know what interfaces we need to create, and we often use that information to build the Mikado Graph.

The automated refactorings of some IDEs can help us extract new super-interfaces from an existing interface, and then use the most specific interface possible at each place where the old reference is used. We need to know how to divide the methods among the interfaces, but we can get significant help with the boilerplate coding. Before doing these types of refactorings, we make sure we know the possibilities of our IDE to avoid manual labor.

We keep on extracting different client specific superinterfaces until we can do what we set out to do, possibly until the original big interface is empty.

### 5.6.5 Dependency Inversion Principle - DIP

*"Depend on abstractions, not on concretions."*

Abstractions, like interfaces or abstract/virtual classes, are in general more stable. Partly so because they often contain no implementation, and partly so because they represent a concept that is at a level above implementation.

Dependency chains from concretion to concretion often make it difficult to refactor code. Say we want to move a class to a new package, but it is involved in a dependency chain that prevents us from moving it without breaking the code. By inserting an abstraction, the dependency chain can be broken, and the code can be moved. The abstraction reverses the dependency, and splits the chain. In *Fig. 5.7 (p.121)* there is a dependency A to B.

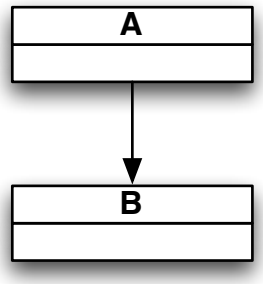


Figure 5.7: Direct dependency between A and B

By extracting an interface `InterfaceB` with the functionality that `A` needs, the chain is broken, like in *Fig. 5.8 (p.122)*

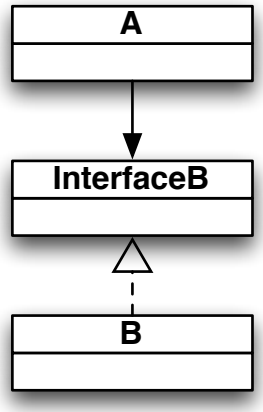


Figure 5.8: Dependency chain broken by inserting an interface

### *DIP in the Mikado context*

The DIP is one of the more commonly applied principles when the Mikado Method is used. Earlier in *Ch.2 An example in code (p.23)*, we illustrated this with the help of a UML-diagram *Fig. 2.1 (p.25)*. An interface was extracted and used to break the circular dependency, which is a common way to adhere to the DIP.

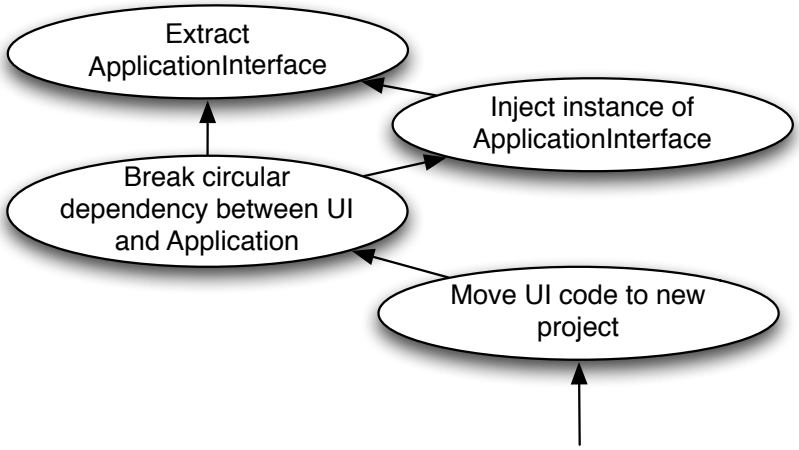


Figure 5.9: DIP in action from the example

Another situation that frequently appears is the need to split code and put it in a separate package, possibly because some form of reuse is desired. Just putting code in a different package won't solve the problem, it will persist until we have changed the direction of the dependencies.

By introducing an abstraction in the target package which defines an API that is used by the newly separated code we are able to take the first steps. The remaining code then implements this abstraction that is specified by the breakout code. In *Fig. 5.10 (p.124)*, this is the case and we have put the functionality in class A into a separate package.

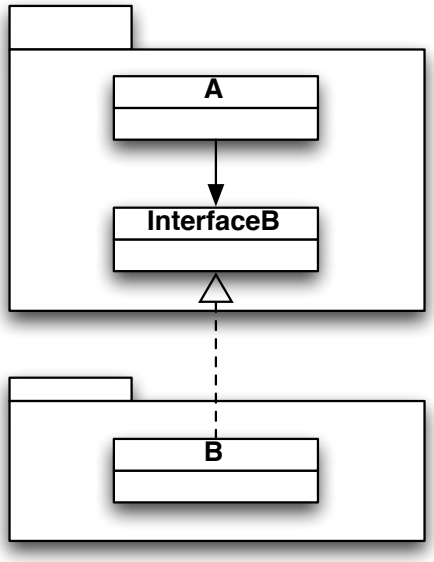


Figure 5.10: When using DIP to break out code to a new package

### *Dependency Injection and Inversion of Control*

By using the DIP, we have introduced a *seam* [1] in the application, a place where we can create variability in the applications behavior without changing existing code. This provides flexibility often needed when refactoring code, but flexibility comes with a cost.

Troublesome dependencies in object-oriented programming are often related to instantiation of objects, since that ties the instantiating class to the instantiated by a call to its constructor. This is a rather strong dependency which makes it difficult or impossible to use another class instead of the one instantiated.

By using the DIP on such a dependency, to instead make the instantiating class depend on an interface, we are posed with another problem.

Abstract classes, or interfaces, cannot be instantiated, hence the dependency has to be created outside of the instantiating class, and the dependency then has to be *injected* into the instantiating class. This is often referred to as *dependency injection (DI)*.

Taking this to the extreme, one realizes that eventually *something* must instantiate classes. This decision is often deferred to *Inversion of Control (IoC) containers*, which are frameworks that specialize in wiring together objects in an application. Sometimes, heavy usage of such IoC containers becomes a burden, making it hard or impossible to follow the flow and logic in an application.

In short, the problem with dependency chains can be solved with DIP and dependency injection. The problems of dependency injection can be solved with IoC containers. The problems with IoC containers can be solved with using less dependency injection.

The conclusion is that we only want to use DIP and dependency injection where we absolutely must have that variability. The Mikado Method usually show us where we need that variability. When we have the variability, but we don't need it, we refactor that part to use direct instantiation in order to reduce complexity.

## 5.7 Principles of packages

The name *package* is a bit ambiguous in software development, so lets define the fuzziness a bit.

*Package* often means the UML notion of a package, which is a logical group into which classes are grouped. In Java, this corresponds to the *package declaration*, which in Java also corresponds to the folder on the filesystem into which the class-files and other resources reside. In other environments, such as Microsoft and Ruby, the equivalent is the *namespace*.

*Package* also translates to UML's *deployable or executable components*, which are anything from a single class, to an arbitrary number of UML packages, packaged for deployment. In a Java/JVM-environment, for instance, a component is typically a *java archive, a JAR-file* or some other archive. In a Microsoft environment, a component is typically

an assembly, most likely a *dynamic link library (DLL)*. A component can in turn be a packaging of other components, or contain several UML packages. Components are often represented as projects in a development environment.

The principles are much more important when dealing with UML components, or projects, than when dealing with UML packages. However, they are still relevant to UML packages, since we sometimes break out a package to become a new component. In general, the principles can be relaxed a bit in packages inside of a component, and especially if the packages are not facing the risk of becoming components themselves.

We will use the notion *package*, by which we in general mean a UML component.

Packages have their own rules for organization that strive for low-coupling and high cohesion. The most cohesive and least coupled package imaginable is when we have one single package with all our code. Then we have full cohesion and zero coupling, which is perfect, the ideal situation.

We only break up an application into several components, or projects, when we must. When we create packages, we use the following principles as good guidelines for how to structure our components contents and dependencies.

In general work, and when using the Mikado Method, understanding these principles helps us make good decisions fast on when to split or merge components.

### 5.7.1 Cohesion principles

The cohesion principles give us forces to put on our code. They will in general tell us what code to put together into a package.

Most of the time they will pull in the same direction. When they don't, or it feels difficult to find a home for a class, it is likely a sign that our classes don't adhere to the class design principles in the previous sections, especially the SRP and the DIP.

### *The Release Reuse Equivalency Principle - REP*

*"The granule of release is the granule of reuse."*

This principle simply states that when we want to reuse something, we should put it in a package and release it. When we reuse a package, we always reuse the entire package.

The main releasable is the application we are deploying. Usually when developing code, we can put all our code into one package, probably in the application we develop. This is our release granule, the deployable. When we don't have anyone reusing any part of our code, we should not create separate packages.

We always try to keep our application in as few packages as possible, preferably a single one like in *Fig. 5.11 (p.127)*. This way, we get no coupling and full cohesion. Another reason for this is that splitting up our code into packages communicates that they are not always used together. If it is true that the packages are not always used together, then all is well. If this is false, then this is noise and miscommunication that will cost time and money throughout the life of a software development effort.

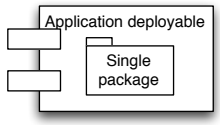


Figure 5.11: Single deployable application with a single package

In addition to understandability problems, the more packages we have, the more difficult it will be to refactor the codebase. Packages and package dependencies limit the room for moving around parts of the code, like classes, since the dependencies don't always travel over package boundaries without trouble. This is what we saw in the *Ch.2 An example in code (p.23)*, illustrated in *Fig. 2.8 (p.36)*.

On the other hand, if someone is interested in using a part of our code

for another application or deployable, then we extract and release that part of the application to a separate package with the shared code. This looks like *Fig. 5.12 (p.128)*. This will be for both of us to reuse. Out of respect for everyone using the shared package, we define how this separate application should be released.

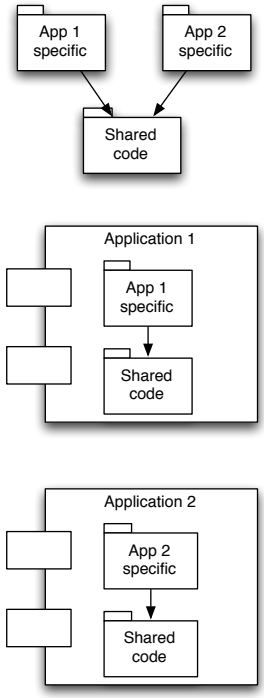


Figure 5.12: Two applications with a shared package

Creating such a package and defining a release process, and keeping too it, is tedious work. We don't want to put ourself to it if we don't have a really good reason. There is *always* an overhead of having several packages.

**We keep everything in a single package when we can, and only**

***split it when we must***.

In other words, we *split a package just in time for its reuse*, and not *just in case* someone might be interested in it.

### *The Common Reuse Principle - CRP*

*"Classes that are used together are packaged together."*

If a set of classes are used together in several different places, those classes should also be packaged together.

If the classes to reuse would reside in several packages, any user of that functionality must set up a dependency to each of the packages, thus increasing the number of inter-package dependencies, making the system more complex.

If we put more classes than necessary into such a package, we create implicit dependencies also to the unused classes. If we reuse one of the classes in the package, dependency-wise we reuse all of them. Therefore, we want to keep those package as slim as possible.

### *The Common Closure Principle - CCP*

*"Classes that change together are packaged together."*

If a set of classes change for the same reasons, they should be packaged together. They should be closed to the same kinds of changes. If the same set of classes are split into two packages, both packages need to change when making a change, which creates unnecessary overhead in a release process and in development complexity.

## 5.7.2 Coupling principles

The coupling principles are about how packages are connected. As for cohesion principles, they put forces on where to put classes, but more from a dependency point-of-view.

### *The Acyclic Dependencies Principle - ADP*

*"The dependency graph of packages must have no cycles."*

Cyclical dependencies have the potential of creating ripple effects that *come back to haunt us* in that they re-enter the code in the package we are changing. The cycles can occur in several steps, like the evil dependencies of *Fig. 5.13 (p.131)*, making them difficult to spot by just looking at the code. We find this very confusing when developing software, and we have a hard time wrapping our heads around such dependencies. Cyclical dependencies usually prevent us from building the system when we are using a statically typed language.

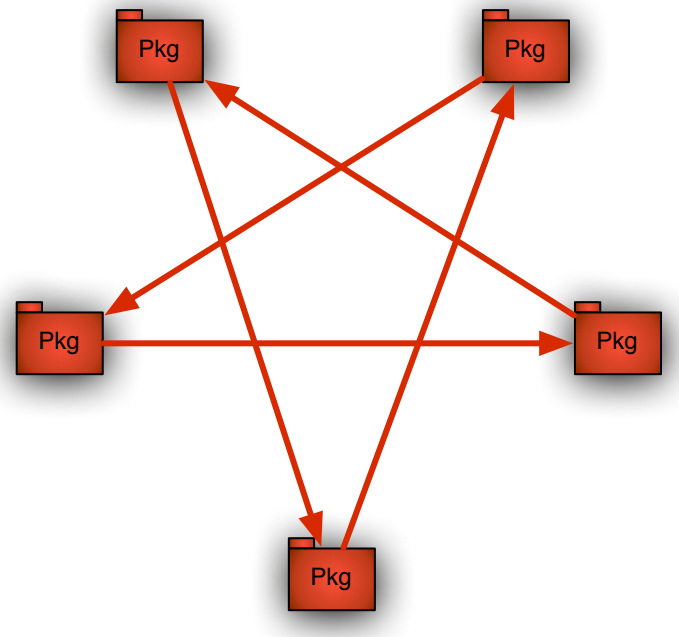


Figure 5.13: Cyclicas dependencies are evil

For parts of a codebase that aren't strictly checked for cycles, they can still exist. One common area is dynamically typed languages that only require everything to be in place when the system runs. While still being able to run the system, we are not protected from the ripple effects of cyclical dependencies. In statically typed languages, reflection and dynamic runtime features can have the same effect.

### *The Stable Dependencies Principle - SDP*

*"Depend in the direction of stability."*

The more incoming dependencies a packages has, the greater is the value of it being stable. If not, from every change in that package there will be ripple effects to all of its dependencies and transient dependencies, like in *Fig. 5.14 (p.132)*. A simple rule of thumb is to avoid dependencies to packages that we know will change a lot.

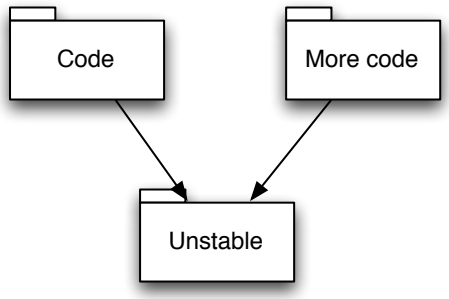


Figure 5.14: Bad situation with unstable dependencies

A more preferable configuration is the one in *Fig. 5.15 (p.133)*.

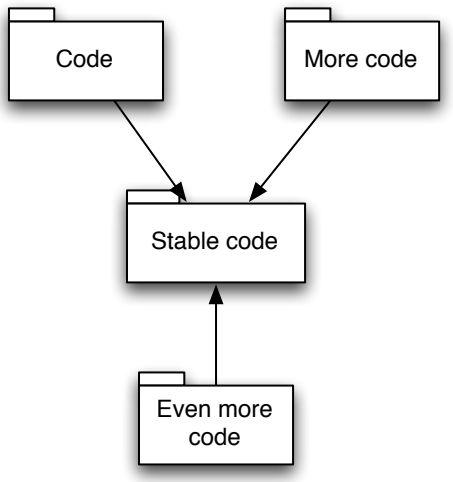


Figure 5.15: A better situation with stable dependencies

### *The Stable Abstractions Principle - SAP*

*"Abstractness increases with stability."*

Abstract classes and interfaces have fewer moving parts, and thereby fewer reasons to change. They are inherently more stable. In the figure *Fig. 5.15 (p.133)* in the previous section, the *Stable code* package is likely to be a package with abstractions.

All packages cannot be stable since a system has to change somewhere in order to be developed. The key is to move the more stable interfaces and abstract classes to their own packages, and their volatile implementors to implementation packages that depend on the API-packages.

### 5.7.3 Packaging principles in the Mikado context

The packaging principles are important to follow when doing large restructurings. They come into play when we need to make decisions on how to structure an application, what path to take, and sometimes to give a name to the problems that we see in the code. As well as for the class principles, the packaging principles can help communication when writing the nodes in the Mikado Graph.

In *Ch.4 Patterns when using The Mikado Method (p.75)*, there are a few patterns that deal with packages and how to move code around, and break parts out to new packages. They are found under *Scattered code patterns (p.87)*.

## 5.8 Side-effect free programming

In functional programming, the concept of *side-effect free code* [23] is essential to create a result that is maintainable and possible to parallelize and distribute on a network of computational nodes.

The concept of side-effect free programming also has great benefits in object-oriented software. There are two important ways to avoid side-effects: making immutable objects and using pure functions.

### 5.8.1 Immutability

Mutable state makes any application harder to understand and more prone to bugs. This comes from the difficulty to understand what and who is reading and writing what data when. The problem is multiplied in a multi-threading environment, where even other threads can change the mutable state. Multi-threaded access problems are "solved" by locking, or synchronizing, certain execution paths for multi-threaded access. This "solution" introduces its own problems with deadlocks and single-threaded execution that degrades performance.

We had better make our code immutable wherever we can, and there will simply be fewer places in the code that mutate state and have

```
...
public String leet(String unleeted) {
    return unleeted.replace('S', '5');
}
...
```

Listing 5.5: Return first

potential bugs. Whenever we can, we create classes that only have immutable properties. This means that any fields in the class should be set in the constructor, and then never changed.

### 5.8.2 Pure functions

Another important part is to have *pure functions*. This means that any operation on data should act as a mathematical function in that the returned value is completely defined by the function and the parameters passed to it. It should not depend on any other state, nor should it change any other state. For the pure functions this, means that any returned data is created instead of changing the state of existing objects.

Code that has no side effects is *extremely easy to refactor and change* since we can move things around more easily.

In functional programming, all methods, or functions, must return a value, hence every function has a return statement, implicit or explicit. A good practice to start creating side-effect free functions is *return first*. This means that any function body should start with the return statement, like in return first.

It is a very simple concept, and there is nothing magical about it. When we follow this practice, it is extremely hard to update or create any mutable state in our code.

This kind of thinking may take some time to get used to, but it can make significant improvements to programs. It does requires common functional programming constructs for working with collections, constructs such as map and filter.

### 5.8.3 On the horizon

Looking towards the horizon of software development, this will be the only way to develop well performing systems in the future. Since processors cannot scale in terms of serial execution to a reasonable cost anymore, they are scaled horizontally, meaning that a processor will consist of more and more kernels that are not as fast as the present processors, but the compound potential computing power will be increasingly higher [22]. That potential power can only be achieved if a program can be executed in parallel and *without* inter-process locking, such as synchronizations of data. Since immutable state doesn't change, there is no need for synchronization, meaning that programs using immutable data structures will run faster on future multicore processors.

### 5.8.4 Side-effect free programming in the Mikado context

When working with the Mikado Method, a majority of the time is spent figuring out how to navigate the dependencies of an application in order to make a change.

Mutable state creates dependencies that are not directly seen in a dependency graph, but rather temporal dependencies that lie the order of execution, in run-time. In a multithreaded environment, even the best test-suite might not find an error. The step in the naïve approach where we find errors is therefore much more difficult to perform than for functional errors or compilation problems.

When we come across such code, we change strategy towards trying to make the methods, or functions, pure and the data objects immutable. This will probably require us to take the refactoring in the Mikado Graph on a small side-path, but it is often worth the effort.

What we have discovered is that pure functions and immutable objects are easier to move around in refactorings and restructurings. Hence, the more side-effect free code we have, the smoother our current and future refactorings will be.

## 5.9 Dependency management

As we saw from the last couple of sections, software is all about dependency management. The Mikado Method adds no magic to handling dependencies. It merely provides us with a structure and a process to utilize our capacity to the max, so that we can solve our problems as best as we can.

## 5.10 Summary

| | |
|---|---|
| Jake: | I need to start thinking about my dependencies! |
| Melinda: | Yes, realizing that was an epiphany in my software development career. |
| Jake: | And I'm going to try to adhere to the design principles. |
| Melinda: | That's good. Remember that they sometimes seem to pull in slightly different directions, so you have to balance them. |
| Jake: | Knowing which direction is the best requires experience... |
| Melinda: | Correct! Which comes from practice. Starting here and now is a good time and place. |
| Jake: | It usually is! |

## 5.11 Try this

- Take the principles, one at a time to see if you can find a couple of examples in your code that break them.
- Also, try to find code that adheres to the principles. Was it easier or more difficult finding code that broke the principles?
- Find some dead code in your codebase and remove it. How did it feel?

# Chapter 6

# Application restructuring techniques

| | |
|---|---|
| Jake: | What other ways of restructuring code are there? |
| Melinda: | A bunch that are quite popular. |
| Jake: | Are they equally good? |
| Melinda: | They all have their benefits. |
| Jake: | So why do I need the Mikado Method? |
| Melinda: | The existing techniques often solve only parts of the problem. The Mikado Method is a sort of 'container' for them. |
| Jake: | To keep me from getting lost? |
| Melinda: | Exactly. And to help identifying where to start and when to stop. |
| Jake: | And to help me communicate. |
| Melinda: | Yes, the Mikado Method complements the existing techniques. |

## 6.1 The state of the code restructuring art

The Mikado Method has not sprung out of thin air. It builds upon existing work in the computer programming field. It does so in regards to good software design and how to change code and how not to change code.

Lets take a look at the ways of working that have preceded and inspired the Mikado Method. As we go through the different techniques and list their benefits, we also show how the Mikado Method fits into each context and possibly completes or makes them more powerful.

## 6.2 Rewriting from scratch

*Rewriting* is typically an effort where an old application is rewritten from scratch, and that old application is then supposed to be replaced by the newly written one at the flick of a switch. The very idea of a rewrite is commonly considered because it is believed that a tipping point has been reached and the estimated cost for the actual rewrite will be lower than the cost of shoehorning in the next dozen features.

### 6.2.1 The pitfalls of rewrites

Many organizations have felt the pain that comes with poorly structured code. They think it's too expensive to go on and keep telling themselves that *this time we'll get it right*. They might even argue for a makeover with comments like *if we just knew then what we know now*. In fact people are likely so busy finding arguments for a rewrite that they completely forget how they got where they are. Let's take a closer look at a few things we think everyone should be aware of before even speaking the word "rewrite":

**Same team, same mess** If the same team gets a shot at the same problem it is likely that the same mess is created, or a slight variation thereof. To get to the bottom of this a systemic change is needed. The

environent i.e the way the work works and the teams practices need to profoundly change, or the result is going to be the same again.

For further illustration, any decision that is different from that of the previous implementation, the treaded path is again into the unknown, and the results are equally unknown. Therefore, it is wiser for a team to learn how to improve an existing codebase, rather than re-create a mess, in a slightly different fashion.

**New team, new mess** People join and leave projects and organizations, and after a while the people with the original knowledge of the system aren't there any more. If the domain is so complex that the system was turned into a mess because of that, it is likely that a new team creates a new [different] mess.

**Same organization, same mess** Sometimes the reason for the current problems lies outside of the software development team. Managers and leaders pushing too hard for new features is probably the most common case. Maybe this outcome isn't so odd after all since the traditional engineering leadership doesn't count for much when it comes to persuading managers about the structure of work. This is a trap though, because if we don't address dysfunctional workflows and change the way the work works, we will end up where we are, even after a rewrite or two. The only difference is that we will have spent more resources and time, and probably gained a few gray hairs.

**It's OK to make a mess** If a team can throw away a created mess and start all over, it is like saying "It's OK to use paid working hours to create a mess. Let's make a new one!"

**The devil is in the details** Underestimating the complexity of a rewrite is easy to do because it is very hard to know about all the details and subtleties in an existing codebase before rewriting an application. It is equally hard to tell beforehand if the dark corners of conditionals are there for a reason, if they went in there because of some misunderstanding, or if they have become obsolete. Those reasons must be investigated and carried on into, or discarded from, the new application. Users will be disappointed if they are depending on and expect certain functionality that just isn't there.

**Big bang by nature** A rule of thumb for a change process is that it should frequently produce tangible value for the stakeholders. A

rewrite does not do that, it more often results in a *big bang* of value as the new system replaces the old one. Or at least, that is what one is hoping for.

As opposed to big-bang value-adding we strongly feel that software development is about constantly restructuring existing systems to fit newly and continuously aggregated knowledge. If a team claims they are so good that they won't create a mess, then they should be able to change the existing application into something better. But, there will always be *"if we knew then what we know now"*. The trick is not to start over into a new unknown, but rather to improve what is already there, with the knowledge gained.

All of these reasons combined make us believe that rewriting is an anti-pattern. By rewriting, the prior knowledge and hard work that went into the existing application is lost. Moreover we believe that the risk profile of a rewrite is a lot higher, for at least three reasons:

First, the existing application will be neglected, meaning existing customers will likely suffer.

Second, the complexity of carrying on all the existing functionality to the new application is daunting, and often results in future users being neglected.

Third, the return of the money invested in the new application will not come before the rewrite is completed. In addition, the money that is spent on the replacement application could instead have been spent on the current one, and thereby adding value. This too has to be taken into account when looking at the financial aspect of a rewrite.

The complexity, neglect of customers and a rather doubtful economic advantage makes us think that it is hard to get ones moneys worth out of rewrites, thus making them a high risk approach.

### 6.2.2 The rise and fall of Netscape

The infamous rewrite of Netscape's popular web browser is now a modern classic, and should serve all developers as a reminder of how not to approach software restructuring. Netscape had almost the entire browser market, but they thought they had reached ways end with

their existing implementation, and went for a rewrite. All resources were put into the rewrite, a rewrite that to no surprise took longer than expected. All the little details, all the conditionals in the old application were there for a reason. During the rewrite, the existing browser didn't get the love it required. An upgrade was released, but it was too little, too late. Soon enough users started to switch to Microsoft's Internet Explorer which had become a lot more appealing. After a while, the new version of Netscape was released, but it was buggy and lacked some important features. Soon thereafter, Netscape went out of business [29].

### 6.2.3 The upsides of a rewrite

The big upside of the rewrite approach is that we can start out with a new architecture and design for the application. We might also be able to disregard the old code and save us the trouble of reading lots of code we don't want to handle. In addition, rewrites are sometimes perceived as a more interesting line of work, and can be used as a way to attract or keep key personnel. These arguments should of course be weighed against the downsides of a rewrite and never be used on their own.

### 6.2.4 Sometimes a rewrite is the only solution

There *are* rare cases when rewrites are the only viable option. For instance, when a piece of hardware or software platform has reached its end-of-life. No longer is there a way to get the components or support needed to keep the system alive at a reasonable cost. At moments like this, a rewrite is probably preferable over deserting the whole business case.

If this kind of situation appears, we recommend the *Strangler application* approach [10] to make the transition smooth(er).

## 6.3 Strangler application

The term *Strangler application* was coined by Martin Fowler. He got the inspiration to the name from a specific vine that grows around a tree to reach the sun by the treetops. As time goes by, the vine covers more and more of the tree, until it's host eventually dies. By now the vine has grown so strong that the support of the original tree isn't needed. This vine is called a *strangler vine*.

In the same manner, an existing application can be replaced by a *strangler application* that initially just proxies the calls to the old application. As time goes by, functionality can be moved from the old to a new structure in small increments, instead of everything at once, in a big bang. In addition to enabling small increments, it also gives a point of variability in the application where testing and measurements can be made to ensure quality.

This approach is not only applicable for total rewrites, it can also be used to replace parts of applications.

It also allows an incremental approach, both in terms of moving functionality as well as implementing new features along the way. The risk profile of such an approach is lower, since it is possible to add functionality to the old system, as well as the new. In addition, if development is halted, there is little or no money invested in unfinished functionality.

The difficulty is knowing more in detail what to do, initially when creating the proxies, and then later on when moving functionality piece by piece. For this, the Mikado Method is a great complement to the Strangler Application approach. Creating the proxies or moving specific functionality are both well suited as goals in a Mikado Graph.

## 6.4 Refactorings

> ***Refactoring*** *(noun): A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*

> ***Refactor*** *(verb): To restructure software by applying a series of refactorings without changing its observable behavior.*

Refactorings were made popular in the book *Refactoring* by Martin Fowler et al [4], but were probably mentioned in print first by Leo Brodie in *Thinking Forth* in 1984 [32].

The word *refactoring* comes from the mathematical term *factoring*, where it means splitting an expression into its components. A very simple example of factoring is the number 12, that we can factor into 2*6, 2*2*3, and 3*4. In refactoring, components are also merged, like in the *Inline method*.

Most of the refactorings that Fowler describes in his book are *atomic refactorings*, changes we cannot split further while preserving functionality. Some examples of atomic refactorings are the *Move method*, or *Introduce parameter*.

Fowler also describes some *composed refactorings*, such as *Replace conditional with strategy* and *Replace inheritance with delegation*. *Refactorings compose*, means that we can combine several atomic, or composite refactorings into sequences to create new composite refactorings. As Fowler describes, combining these refactorings can make a significant difference.

Most of these atomic refactorings, and many of the composed ones, are automated in modern IDEs. The automation substantially reduces the risk of making errors while doing them.

### 6.4.1 Big refactorings

In his book *Refactoring to Patterns* [5] Joshua Kerievsky describes refactorings with the purpose to transform code to and from some of the most common design patterns from *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma et al [6].

Fowler briefly mentions big refactorings in his book, where a series of smaller refactorings can be applied to a code base one after the other to constitute a more extensive change.

When we pick a Mikado Goal which has the characteristic of changing the internal structure rather than adding new functionality, the entire graph becomes a huge composed refactoring. It might be stretching the refactoring definition a bit, but technically, it is correct.

More often, the nodes in the graph are refactorings by Fowler's definition, and some larger sub-graph of stringed together refactorings could be argued to be refactorings as well. Other parts, and the whole, are more often restructurings, changes where the observable functionality is altered.

We believe that it is getting the desired effect of changes to code that counts. Drawing the exact line between refactorings and restructurings is not our first priority when changing a codebase to meet our business goals.

### 6.4.2 Refactoring in advance

Refactorings, by the definition, can also be used to prepare a system for change. This means that we string together a series of refactorings first, in order to make the actual change to our system as smooth as possible.

Let's say we have different behaviors in a codebase that are chosen by comparison with constants, and these constants are used in conditionals all over the codebase. If we want to add more behavior, we could edit all the conditionals and hope we don't need to do it again. Another way to approach this would be to start with a refactoring called *Replace conditional with strategy*. After that, we can get away with only the addition of a new strategy, an implementation of the abstraction, for the desired functionality. It also means that the change is isolated to one place.

Sometimes, refactoring in advance is called *prefactoring*. This has some ambiguity with the meaning that Ken Pugh uses in his book [30], where prefactoring rather means using the *a priori* knowledge we have about good design and refactoring, and apply it in advance to avoid refactoring. While this can be good at times, there is a fine balance to strike. There is a risk that a system gets over-designed, and overly complex from such an approach.

### 6.4.3 Refactoring for a comprehensible codebase

Code is read a lot more than it is edited and as we realize this, it becomes obvious that the code must be easy to read, as well as express its intention well. Code that doesn't read well costs extra money each time it is read and edited, which makes it more expensive to change.

Sometimes, we are able to create nice and shiny code from the start. But more often, code grows slowly, like a garden or a plant, and we have to change it, trim it, and clean it as we go. This is where refactorings really shine. They enable removing comments by expressing new things learned from reading the code *in* the code, such as by renaming variables, methods and modules. They also help reduce complexity, which is often done by splitting large methods into smaller ones with the help of *Extract method*.

In "Clean Code" [7], Robert C Martin et al gives plenty of advice on how to make code clean and readable. In "Implementation patterns" [8], Kent Beck explains how to write code that communicates its intent well.

### 6.4.4 Refactorings from the Mikado perspective

All the refactoring techniques mentioned in this section are essential knowledge for a software developer. They are of immense help when restructuring an application as they provide easy-to-use building blocks of change. They provide the technique for incrementally changing an application where every step is a small and low risk change to a codebase.

By referring to existing knowledge, such as cataloged refactorings, in a Mikado Graph, information is conveyed about what needs to be done. It also communicates how and where to use a particular refactoring so seasoned developers don't have to rediscover what needs to be done and more junior developers can learn in what context different refactorings are best suited.

The Mikado Method benefits heavily from the body of refactoring knowledge and we believe that the method adds the missing piece to the restructuring puzzle by finding the future path of necessary steps and

illuminating it. The method also helps communicate that path within a team or an organization by visualizing it. It also addresses the risk of spending too much time with the wrong problem by having a clear goal to work towards at all times.

## 6.5 WORKING EFFECTIVELY WITH LEGACY CODE

Michael Feathers' book *Working effectively with legacy code* [1] has an extensive set of detailed practices for safely getting software under test and ready for change.

One of the big challanges with altering legacy code is knowing how to start and where to put that first cut. Feathers presents several different ways of introducing what he calls *seams* in an application. Seams are places in code where behavior can be altered without changing the code there. They are used to introduce tests to make an application more flexible to behavioral change.

Feathers also introduces *effect sketches* in which he draws relevant dependencies for affected code in a graph, as a way to explore how dependencies stretch through the codebase. By drawing these sketches before and after a change it becomes clearer how the changes affect the dependencies in an application.

Another important practice are *Scratch refactorings*, refactorings that are made to explore options, only to return the changed code to its original state afterwards. In that respect the Mikado Method can indeed be viewed as a brainchild of Feathers' work.

*Scratch refactorings* and his *Lean on the compiler* technique are powerful empirical tools the Mikado Method makes heavy use of, in addition to analysis. The effect sketches also touch the aspect of how to manage larger changes. The Mikado Method adds the focus that a goal provides and a robust way to find the *changes* needed to get there.

## 6.6 Design patterns and design principles

All software developers will sooner or later come in to contact with *Design Patterns* in their career, either by implementing some themselves, or working with them as a part of a programming language or an API. The concept of Design patterns and the pattern language originally came from Christopher Alexander et. al. and is described in their book about designing towns and buildings, *A Pattern Language* [31]. In [6], Gamma et. al. took these ideas and applied them to software development and in the same fashion uses patterns to describe design solutions to recurring situations and problems.

*Design principles* can be viewed as the "abstractions" of design patterns, or any well structured code. They are the high-level description of what constitutes code that is change-friendly. In *Agile Software Development* [12], Robert C. Martin present good practices for developing software, especially design principles and how to structure packages. Later we take a deeper dive and drill into the details of how design principles relate to the Mikado Method in *Ch.5 Guidance for creating the nodes in the Mikado Graph (p.103)*.

If refactorings and Feathers' advice tell us *how to walk*, the design principles are the *general direction*. In itself the Mikado Method says little about software design, and equally little about deciding what path to take, but it will help create and keep track of the paths of change that lead to the goal. While software design principles improve the chances of making the correct choices that lead to a sustainable system faster, the Mikado Method provides the big picture. It effectively shows a starting point and the following steps which are needed to transform a software system effectively into a new desired shape.

## 6.7 The Mikado Method

When we were struggling with our beast, we first tried all sorts of refactorings. We had our code under test, we tried to follow the design principles, but all of that didn't take us anywhere near where we wanted

to be. We lacked a plan and we tried to fix every flaw that we found. We were missing the *Big Picture*. We needed to rise above the swamp of code we had and get a helicopter perspective. We already had most of the pieces, but it took a failure and some insight to get the last pieces in and put them together into something that better met our needs.

The Mikado Method ties together a lot of existing practices, including avoiding the general anti-pattern of a rewrite into a cohesive whole. It is a method usable for attacking almost any change in an efficient and effective manner. The Naïve Approach solves the problem of not arriving at the required sequence of refactorings and changes needed, and the Mikado Graph provides a graphical language to visualize those sequences in a change-centric way. The leaf-first approach enables us to work with the smallest increments and always have our system in a deliverable state. The method is agnostic when it comes to the details of change. It is merely a change container, which makes it easy to extend or adapt to suit most of our special needs.

## 6.8 Summary

| | |
|---|---|
| Jake: | Ok, I definitely won't do the rewrite again! |
| Melinda: | Well, you might have to. But then the Strangler application is probably the best way to do it. |
| Jake: | But I'd rather use the Mikado Method, and then refactor and use Feathers' legacy techniques inside it. |
| Melinda: | You can use the Mikado Method with the Strangler application approach, or with refactorings and the legacy techniques. |
| Jake: | Sweet. And with design patterns and principles? |
| Melinda: | Yes! The Mikado Method is rather agnostic when it comes to the actions performed. |
| Jake: | I might even come up with some new techniques on my own? |
| Melinda: | You should. Not everything can be pre-packaged, because every situation is unique. |

## 6.9 Try this

- Which of the techniques in this chapter are you familiar with? Do you have any favorites?
- Which of the techniques are you not familiar with? Make a list.
- Select one technique you know that you will aim at getting better at, and one unfamiliar technique you will learn.

# Chapter 7

# Organizing our work

| | |
|---|---|
| Jake: | The Mikado Graph works well for me when I pick up my work after a break. |
| Melinda: | Yes. On top of that it is a good way to communicate. |
| Jake: | You mean with my co workers, like we did? |
| Melinda: | Yes, or with a whole team, or the future you. |
| Jake: | Ah, you mean after a longer break? |
| Melinda: | Yes. |
| Jake: | I see. But how detailed should I be with the Mikado Graph? |
| Melinda: | It depends. |
| Jake: | And when should I start using the Mikado Method? |
| Melinda: | There are a few different ways to begin... |

## 7.1 Different scales and scopes of code improvements

Software is destined to change and hopefully to improve. Depending on the scale and scope of the changes and improvements that are desired, the value of using the Mikado Method varies. The following sections will provide some insight into different levels of code improvements,

and when the Mikado Method comes in handy.

### 7.1.1 Small-scale improvements

By small-scale improvements, we mean things that are on the scale of a method or a class, and where the implications of making changes mostly stay within a handful of lines of code. This includes *bad naming*, minor and obvious *duplications* and *unnecessary code*, to name a few. These are the types of flaws that are often remedied in seconds in a modern IDE, by using a pre-packaged refactoring from a menu.

Unless a dynamic language or a dynamic feature like reflection is used, the compiler can probably tell if something went wrong. Regardless, it is wise to have some sort of safety net while performing changes, like a suite of tests. This suite should be so fast that it can be run after every change. If it takes longer than a couple of minutes, chances are they become more of a straightjacket than a safety net. When dynamic languages or features are used, the need for that safety net will be a lot more obvious.

Small scale improvements make a lot of sense to do continuously as the code is read, or before implementing new features. When readability is improved, benefit comes almost instantly. As a bonus, others who follow will benefit too. The book *Clean Code* [7] has plenty of advice for dealing with these situations. The behavior we are describing is sometimes referred to as "The Boy Scout Rule": Always leave the camping ground cleaner than you found it. *Always leave the codebase in a better shape than you found it.*

Even if the change is ever so small, like renaming variables and methods, we still advocate that it is backed up or checked in into a versioning system as soon as possible. This way, nothing is lost and all those small improvements can be kept if a rollback of something bigger is unavoidable.

As cleanups are more or less *ad hoc*, the Mikado Method doesn't add much value at this low level. To us, small-scale refactorings have a lot to do with clarity, and the fact that problems tend to surface as the surroundings get less opaque. It is almost like seeing something for

the first time and under a different light, thanks to all these refactorings. When we slowly remove dirt and mud from our code, the need for larger repairs becomes more apparent. Our discovery of medium-scale improvements often starts with smaller improvements. One of the reasons we engage a lot in small-scale refactorings is to be able to see the bigger problems.

### 7.1.2 Medium-scale improvements

Medium scale improvements typically include a handful of classes. Some examples are *replace a wild-grown conditional with polymorphism*, or *replace an implicit tree with composite*, or split up a single class that has too many responsibilities into several classes. Other examples include fixing awkward usage of abstractions, or fixing a part of a design that is disintegrating under changing requirements.

In addition to discoveries made when doing small-scale refactorings, these changes also come from an itchy feeling every time a set of classes is read. Change come when we think to ourselves *there must be a better way of doing this*, and there often is.

Changes like these usually take a couple of minutes to a couple of hours to make for a single developer or a pair, depending on the size and the number of classes or responsibilities involved.

This is where the Mikado Method starts becoming useful and adds value. To start with, this is a convenient size problem for a beginner when trying out and learning how to use the method. It it also useful when interruptions are common and there is a desire to get back on the train of thought quicker.

When changes create ripple effects through the codebase, or every fix that is applied just opens up a new can of worms, these are strong indicators that a *large-scale improvement* is needed.

If there is teamwork going on, now is a good time to start communicating improvements within that team, since it will soon start to have more significant impact on the code and the work system. Learning about the Mikado Method might also come in handy at this point.

### 7.1.3 Large-scale improvements

A large-scale improvement is when several parts of a system needs to change. Structural improvements include breaking up heavily entangled code and mazes of dependencies, extracting APIs from what were not meant to be API, or just dealing with the ripple effects of a larger change.

Often, there is an external goal and a bigger vision connected with these kinds of changes, and normally they pose a lot of work. To make matters worse, it is hard to estimate the complexity and size of changes like these.

When large portions of the code are changed the whole team will have to be engaged, and everyone needs to know the goal to be able to help out and understand. The team also has to communicate outside of its immediate vicinity, especially if the change is based on an external goal.

**This is the perfect situation for the Mikado Method.** As the code undergoes these changes work is probably spread out over several days, weeks or months, and can benefit a lot from being carried out by several developers on a team. Putting the Mikado Graph on a whiteboard during this period of change is a great way to communicate that goal, the progress and its current state. External stakeholders, managers and other people with interest in the progress are also more likely to get involved if they can see what is happening.

## 7.2 SELECTING THE GOAL

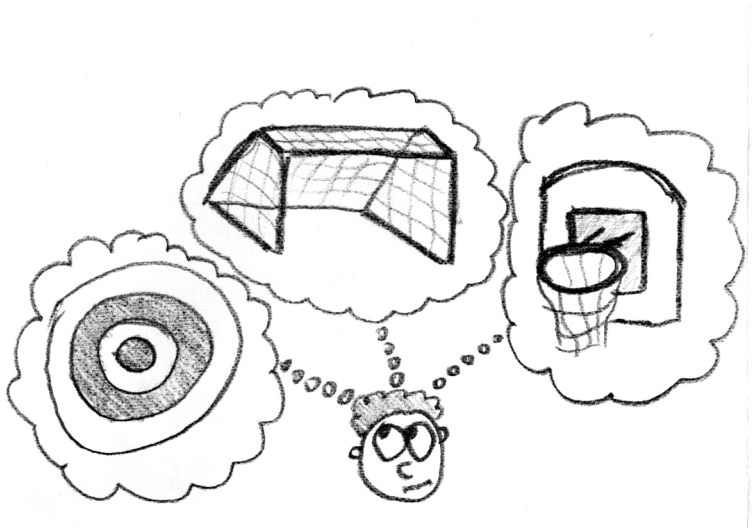

Figure 7.1: Selecting a goal

A very important aspect of the Mikado Method is to select the goal, or goals, to work with. Sometimes, we come up with a goal relatively easily, and sometimes it is a bit more difficult. There are mainly two different types of goals to work with: new features and structural improvements.

### 7.2.1 New or augmented features

A *new or augmented feature to implement* can be used as a goal, perhaps in the shape of a use case, user story, theme or epic [20]. In the example in *Ch.2 An example in code (p.23)*, the new deliverable was the goal. When we implemented the goal and problems appeared, the Mikado Graph was grown with new prerequisites.

Large features, like themes or epics, can be broken down to subcomponents as a first step, and then each of the subcomponents become goals respectively. This way of working keeps the information about the features and the refactorings in one place.

Sometimes, the goal needs to be broken down one step further before we have something that we can actually try and get feedback on with the naïve approach, just as we had to do in the example in *Ch.2 An example in code (p.23)*.

For new features or changes, the goal can be formulated as they normally are in a story or requirement. This ties them back to the business value of the work so the non-technical people can read the graph and get an idea of what's going on.

### 7.2.2 Improve the structure

Code that is often read and edited, or is unfit for its purpose, may need improvement too. Unlike a feature driven change, an effort to *improve the structure* leaves the external functionality of the system unchanged.

There are numerous potential candidates for structural improvements. It could be a custom-made framework, or an overhaul of the entire persistence architecture. To take on such a task when adding a single user story or feature might be too much, as changes of that caliber can require some additional planning and communication before they are started.

We recommend that the goal is formulated as something that has a tangible business value to the organization in terms of increased return on investment, positive cost-benefit, or a reduced risk. That way, we get a lot less friction in the discussions with the people that pay for the system. Normally these types of improvements are often requested by developers, and if they can also estimate the quantitative improvements in hours or money, it is even more likely that they will end up being approved.

Some examples of goals are *Replace hardcoded implementation with a pluggable architecture to add new services faster at lower risk* or *Replace XML configuration with code to simplify code navigation*.

## 7.3 Working as a single developer

When we are working as single developers on a project or product, we have some advantages and some disadvantages. The big advantage is that communication within the team is easier. There are also hardly any disruptive disagreements or political problems within 'the development team' either.

On the other hand, no one will be there to critique the work, or help us reflect upon what we are doing. That is up to us to provide, so we write the graph on a piece of paper that we keep nearby. During and after our work, we can always take a step back to look at the graph. By doing so we get a chance to reflect and possibly modify our strategy and tactics.

Writing clearly in the nodes of the graph and being precise might not feel important at first, but we have noticed that coming back to a messy graph even after a short break can add significantly to the already cumbersome work of restructuring. This is why we try to keep the graph in the same condition as we would if it was a shared one.

## 7.4 Working in a pair

Pair-programming is a practice propagated by development methodologies such as Extreme Programming [25]. Normally when doing pair programming, one person, the 'driver', is in charge of the mouse and the keyboard and writes the code needed. The other person, the 'co-driver', helps the driver by keeping track of what to do next, proofreads the code, and is constantly alert and ready to take over driving when it is time to switch. Switching driver/co-driver should occur fairly often, to avoid getting stuck in a rut.

Since the co-driver doesn't need to focus on the nitty gritty details of code and has both hands free to draw, the co-driver is best suited to update the graph. Solving problems as a pair has other advantages too, the difference between tactics and strategy, which becomes very striking when the Mikado Method is used.

While the driver focuses on going from ideas to actual written code,

the co-driver can stay at a higher level of abstraction, keeping the big picture in mind. The abstraction level of future design is more aligned with the whole graph, and the tactical decisions of programming are more with the single nodes of the graph. The driver focuses on 'the how' and the co-driver updates the graph and decides on 'the what and when'.

When the roles are switched back and forth during pairing some extra effort is needed to make the Mikado Graph readable for the both the driver and the co-driver. The graph needs be explicit enough for any one in the pair to be able to continue the work where the other left off.

## 7.5 Working in a team

We can also use the Mikado Method in a team, to coordinate ongoing re-designs and refactorings. If we do, we want the Mikado Graph to be put in a place where everyone can change it and see it, like on a whiteboard in our team room.

Putting the graph on a whiteboard sends signals to the team. There is a *visualized goal* and there is a path to get there, it is *collectively owned*, and anyone is allowed to change it and collaborate around it. This gives everyone the opportunity to contribute by expanding, changing, or questioning the graph and a more tangible incentive to reach the goal.

When we have a visual representation of the goal in the form of a graph, it is easier to spot if a decision is in direct conflict with that goal or if it takes the code in the right direction. A collectively owned Mikado Graph can also double as a visual task board and brings the whole team closer to a mutual understanding of what they are doing. To increase its effectiveness, it's important that all team members can quickly understand what the essence of a node is, which makes an early agreement on the granularity of node content all the more important. Let the team decide upon how explicit and detailed the information in the nodes of the graph should be and stick to that level of detail during the lifetime of the graph.

Large-scale restructurings or refactorings, the kind where the Mikado Method is especially helpful, should start by communicating within the

team and then spread this information to the rest of the organization. Since this kind of work often affects larger areas of the codebase, there may be conflicts with the work other people are doing. Which is why large, unannounced, single-person refactoring raids scale poorly.

A few more aspects of this are discussed in *Ch.8 Setting the stage for improvements (p.169)*.

To work as a team with the restructurings, we need to decide on how to split the team's focus between adding new functionality and embarking on structural improvements.

### 7.5.1 Working with the main focus on new or improved features

One way of working is with the main focus on implementing new features, and restructuring parts only if they pose an immediate problem. This way, it is easier to get acceptance for structural improvements, since they are tied to a clear business value.

We can use the features as goals, grow the graph if need be, take on as few goals as possible at a time, and try to finish the goals as soon as possible. This way of working reduces the risk of spending time on things that don't get done. This is sometimes referred to as *Limited Work-In-Progress*, and is an important part of the Lean paradigm [28].

When the structural improvements needed are too big to be contained within the implementation of a single feature, this approach has to be somewhat modified.

### 7.5.2 Working with the main focus on structural improvements

When the aim is to improve the internal structure of a system, the graph is created with that improvement as the Mikado Goal. Everyone on the team should, in a coordinated way, build the graph and prune the leaves. Focus should be on reaching the goal as quickly as possible.

This situation is rarely desirable since features are not continuously delivered and its hard to motivate further investments in time and money, even though it may be crucial for the product. When the frequency of delivered functionality from developers to stakeholders drops, so does the level of trust, hence we don't want to get stuck in this situation. In spite of this, it can be a useful approach to concentrate focus to make an investment for the future, to quickly move past an obstacle to make it easier to get future features out the door.

When we are working with a system on a daily basis, we get a general feel for how well the system responds to change. This knowledge is *internalized*, and we consciously or unconsciously shy away from working on certain parts of the system. This is important information, since it probably affects both spirit and productivity in the team. These parts of the system are good candidates to consider for a focused improvement effort.

### 7.5.3 Combining structural improvements with implementing new features

Most of the time we need continuous structural improvements along with the development of new features. The distribution between those aspects varies heavily, however, and when there is a substantial need for structural work it's a good idea to distribute the effort somehow. One way is to split the team to let some work with new features, and let the rest focus solely on structural improvements. Another way is to spend a certain amount of time on features and the rest on structural improvements.

In order to keep the balance between structural and feature goals, we must set up for collaboration and communication to keep everyone aware of what the rest of the team is doing. One way to achieve this is to rotate between structural improvement and implementing new features. At the very least, we need to get together on a daily basis to get an idea of the common results and plans.

It is also possible to give a signal when a feature-worker enters code that is touching goals and prerequisites of the structural improvements. This sends a message to the structure-workers to take care

of that part as soon as possible. The user story might reveal work and details that will change the graph. This is also a good opportunity to have a design discussion, and perhaps update the graph. There are, of course, a multitude of ways in which you can combine structural and feature work and we have to find what works best in each context. If we inspect and adapt regularly, we can normally find the conditions that work best for us.

Splitting the focus on feature and structural work is especially suitable when we have the need for refactorings that are more of a long term kind, or very complex by nature. It is not unlikely that the system needs to be delivered several times before the improvements are done. One of the Mikado Method's better traits is that it enables us to do this work in the main branch, as the system is always in a deliverable state.

### 7.5.4 Distributed teams

Sometimes, we are not situated in a single room. At those occasions, try to find a tool that will allow everyone to view and modify the Mikado Graph, preferably in real time, and where several users can simultaneously update the graph. In addition, a teleconferencing system, or the like, for the verbal communications is also good to have in place. In the end, the needs of the team will have to decide what solution to choose. A good rule of thumb is to start with something that is as simple as possible, and then update when there is a need to do so.

As always with distributed teams, more time and effort would have to be put into communication, in this case around the creation and evolution of the Mikado Graph.

## 7.6 How much Mikado should I use

There are several ways to start out with the Mikado Method. The following different tactics are some that we have tried, but they are not in any way the only ones possible.

### 7.6.1 The Dogmatic tactic

Before a new way of working and thinking is internalized we need rigid rules to follow. One way to encourage reflection and to increase the amount of training we get is to apply this new way of working and thinking as much as we can.

By using the Mikado Method for every new restructuring, small or large, we maximize our exposure to the ideas and mechanics of the Method in order to learn faster.

Worst case, a piece of paper is wasted by drawing a ring around a goal after we have realizing that it could have been achieved right away. Best case, there is always a graph prepared when trouble appears.

### 7.6.2 The Pomodoro tactic

We start out as we usually do with an improvement or addition in mind but also add a time constraint. If nothing is achieved during the first Pomodoro, we take a 5 minute break and try to figure out why. This could be a good moment to start drawing a Mikado Graph of the dependencies, for instance by reverting the changes and starting fresh.

Worst case, 25 minutes is lost, which really is not a lot. Best case, no graph had to be drawn.

Overall, the Pomodoro Technique is an excellent companion for the Mikado Method. The focus-reflection cycle of the Pomodoro Technique amplifies the *zoom in-zoom out* cycle of the Mikado Method perfectly. In the 25 minutes of focused work, a leaf is a suitable chunk to work on, and if needed, the graph can be expanded with new findings. The 5 minutes of reflection will help zoom out to look at the big picture, giving a great opportunity to snap out of any blockings. The reflection might lead to extend the leaf with new prereuisites, revert the current state, or realize there is a better way alltogether.

### 7.6.3 The 3rd level tactic

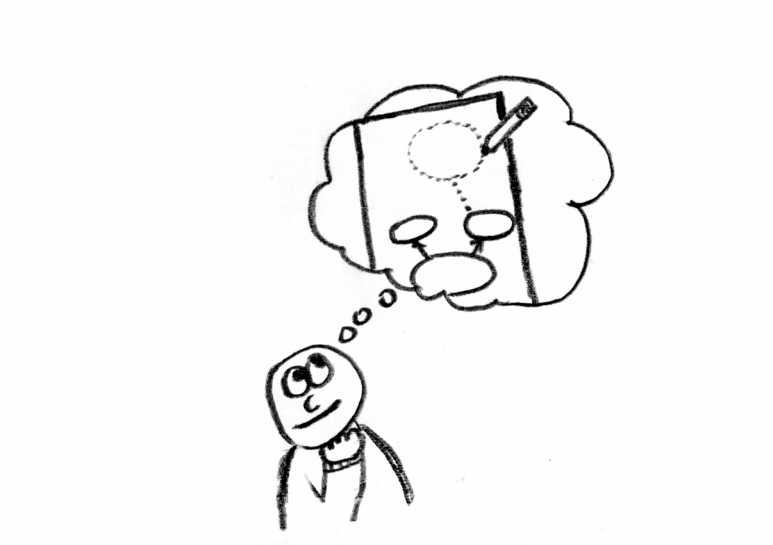

Figure 7.2: 3rd level tactic

We keep the dependencies in our head as long as there is only a goal and one level of prerequisites. As soon as it gets more complicated than that, we draw the graph and keep adding to it.

This tactic could also be called *the accidental tactic*, since often we don't know we are about to get in trouble when we are working. All of a sudden, we are in the middle of a mess. This is a good time to pull out a pen and some paper and start drawing a graph. Then we revert our changes, starting over again in a more structured way with the Mikado Method.

### 7.6.4 Which tactic should I choose?

When learning a new thing, we often try it out in all cases to see where it fits, and where it doesn't. We find the limits by stretching the method to its boundaries a few times. This means that it is a good idea to use the Mikado Method for all tasks in the beginning, like the dogmatic tactic. After a while, we know better when to use it, and can choose to use it on a case by case basis.

## 7.7 Small steps

When we start out, we move with baby steps. We find easy tasks that will benefit all the software developers, including ourselves. We do basic and low-risk clean-ups, like removing dead code and unnecessary comments while getting to know the structure of the codebase. We try to get a good feeling in our stomachs, and in others' stomachs, almost like a warm-up at the start of a physical training session.

In the short run, these small changes might not look like much. But if a few small changes for the better are done every day, the whole will soon be a lot better, maybe even within an iteration. As we learn about the codebase and the most pressing problems, we can take on bigger tasks and changes.

We think taking on bigger tasks is about taking more small steps, and faster progress comes from higher frequency and equally small steps. The Mikado Method offers help by helping us find these small steps. It is the ability to find and take these small steps often enough that is the core of solid and robust software development.

## 7.8 Summary

| | |
|---|---|
| Jake: | I should think about what goal I have? |
| Melinda: | Yes, you should. Sometimes features are better, and sometimes structural improvements are better. |
| Jake: | And how do I know? |

Melinda: If you work a while with the code, it often comes to you. Implement features and make small improvements to see what is stopping you.
Jake: I like delivering features, but I guess that some improvements are to big to do along with a feature?
Melinda: Some are, but that is a good mindset.
Jake: And when I know what is bugging me, then I just draw the graph?
Melinda: You draw it when you have to. Right away or when you can't remember what you need to do.
Jake: And if I work in a team?
Melinda: If you work in a team, doing it together is a good thing to get it going. You want collective ownership.
Jake: And if I'm on a distributed teams?
Melinda: If you work on a distributed team, you need to pay extra attention to the communication. Any software that you can use for drawing and editing ought to be sufficient for the graph.

## 7.9 Try this

- How would you use the Mikado Method? In a team, as a pair, or as a single developer? What tactic would you use?
- What do you need to get started? Make a list of your prerequisites.
- Make a list of the problems you see in your codebase. Mark them as small, medium, or large scale problem? Would the Mikado Method be a good fit to help you solve your problems?

# Chapter 8

# Setting the stage for improvements

Jake: I'm starting to feel pretty confident with the technical aspects of restructuring code!

Melinda: You are! Now we can start talking about how to prepare for a restructuring effort.

Jake: I knew there was more to it...

Melinda: Yes. Now we want to make sure that you spend your time wisely, working with the essential parts of the problem.

Jake: Sounds fair enough. But isn't that what you always do?

Melinda: In some organizations, yes. In others, there are some things you should get in place before doing any improvements.

Jake: Shouldn't you find that out by finding the prerequisites in the graph?

Melinda: It is fairly easy to find the technical stuff that way. When there are people involved, in organizations, it is more difficult. There are some things that are good to do up front.

Jake: It is hard to run a compiler, or automated tests, on people...
Melinda: That is one way of putting it ...

## 8.1 How to verify our work

### 8.1.1 Feedback cycles

When dealing with code improvements, we are much better off if we can verify that our changes did not break anything else. Programming benefits a lot from closing feedback loops and shortening the feedback cycle. To illustrate, we use the example of driving a car.

When we are driving a car, there are several feedback cycles at different scales. By turning the steering wheel, we get feedback from how the car changes its direction by its position on the road changes. We get feedback on how we are approaching our destination from road signs.

In the car, there is the visually observable feedback, *if the wheel turns clock-wise, then we go right*. In addition to feedback that is observable by anyone in the car, when we are working the steering wheel and the pedals, we get a deeper level of understanding for the details of the car system.

Included in this understanding is the variability of the system, like *when we turn or brake, how much does the friction between the tire and the road vary*. These attributes of the system cannot be specified exactly or known beforehand as the road might be wet or icy which affects the driving experience substantially. These characteristics are *stochastic*, meaning that they randomly vary, but some values are statistically more likely than others. In order for us to steer such a system, feedback and constant corrections based on that feedback are required.

The deep understanding of such a system is often called *tacit knowledge*, and we can only attain it by practice. To become an expert driver, we need thousands of hours of driving practice, and we need to push the car to its limits, and beyond, in order to learn the behavior and boundaries of handling the car.

For us to avoid risking the life and limb of ourselves and others, we want to practice in a place where it is *safe to fail*, preferably in a restricted area where we can be sure not to suddenly crash into something unexpected.

The faster we get the feedback, the faster we dare to go, to the point where processing and responding to the feedback becomes the bottleneck. Therefore, we don't want to be constantly overwhelmed with detailed information. To know the exact rotation of the wheels is not that relevant when controlling a car, and the oil pressure is only of interest when it is out of the ordinary.

We also try to limit unwanted variability, or noise, in the feedback loop. Dirty windows, play in the steering, and slipping tires are some examples of noise that makes it harder to control a car. The similarities between software development and driving a car are plentiful.

We want to be able to try things, to sit by the steering wheel, and get feedback from our own actions. The development equivalent would be to be able to run the system, and preferably run automated tests for the product we are developing and see how that goes.

We want to have an environment where we can try things and fail safely, in order to learn how the system behaves at its boundaries. Just like no two cars are the same, no two systems are the same. In order to perform, we need to learn. The more things we can try without disturbing or depending on others, the better. This is similar to the ability for everyone on a team to be able to run the automated tests on their local workstation, and on a build machine.

Just as we don't want dirty windows on our car or play in the steering we want to limit the amount of noise in our system development feedback loop. We want a failing test to fail because of the things we've altered, not because that the Sun, the Moon and Saturn are aligned, or that the pointer on the screen was located at the wrong pixel. We don't want to measure everything measurable, but only the most important indicators that tells us how we are doing.

In software development, there are several ways to create feedback cycles. The compiler, automated tests and manual tests are some ways we'd like to describe a bit more detailed.

### 8.1.2 Automation

One of the most important things that can be done about the build process is automating it, and we mean fully automating it. All the way from checking in code to the completion of a newly produced CD, installer or whatever the final artifact is. Automation enables one of the cornerstones of reliable feedback – repeatability.

A typical build process setup is *pull all code from a versioning system, compile, run micro tests, put together artifact, deploy, run macro tests, create documentation, create distributable artifact.*

In fact, try to automate as much as possible as soon as possible. If we truly need the results of the automation, it is time well spent. But spending time on automating things we don't need is utter waste and should be avoided.

The book *Continuous Delivery* [36] by Jez Humble and David Farely offers detailed advice on how to automate building, testing and deployment.

### 8.1.3 Automated tests

Our best friend when we are up against a bigger refactoring or restructuring is *automated tests*. When we say automated tests we mean pieces of code that execute a part of the system and assert that the expected functionality is there, without human intervention. Such tests can be written on a macro level, and on a micro level.

#### *Macro level tests*

Automated tests that work the system from the outside, like a user would, are extra valuable when systems need to change. Those tests are sometimes called macro, system, functional, or acceptance tests. They typically launch a GUI, click buttons, enter text, select checkboxes and more, just like a user would. By doing so it exercises the system in an end-to-end fashion. It's very common that all of these

actions in turn store a state in a persistent manner, usually by using some sort of database. As the suite of tests is run, it verifies the *business value* of the system as well as its wiring and plumbing. We don't want the functionality to change as internal changes are made, so these tests become our safety net that catch us if we make mistakes. The naïve approach partly relies on this safety net and it also does a good job and tells us what prerequisites should be considered.

The downside of such tests is the end-to-end characteristic which means they often take a long time to execute. In return they are able to provide loads of high quality feedback, but that feedback arrives slower than the feedback from micro level tests. Improving the execution speed and tightening the feedback cycle is possible to a certain degree, but dramatic improvements come at the price of the amount of and quality of feedback they provide. To make major improvements in execution time less code has to be exercised, which means bypassing important parts of an end-to-end test. So in order to get even faster feedback we need a complement, namely micro-level tests.

### *Micro-level tests*

Micro level tests are tests that don't use any external resource, such as a file, socket or a screen. They test smaller chunks of logic and should execute entirely in memory, which means that hundreds, thousands, or even more tests can run within a second. This makes it possible to verify a lot of the logic in the code in a very short time. The downside is that they normally don't verify that a user can actually perform a business value function in the end product.

These tests might come with another type of problem if written in a bad way. If care is not taken, they can be on the same level of detail as the refactoring or restructuring changes. Hence, for every piece of code that needs to change, there might be several tests that have to change as well. In a way, *bad micro tests could hold the code hostage*.

If a code hostage situation appears, there are a couple of things that can be done. One is to start writing macro tests, on a higher level, if they don't already exist. Then we can just delete the troublesome micro tests.

Better yet is to try and find a level for micro tests where they verify modules with a behavior that is more invariant to the way the application is structured. This often implies a small cluster of classes, performing some cohesive service to the application. Testing such a cluster gives more stable micro tests and does not affect the feedback cycle time.

### 8.1.4 Compiler support

For statically typed languages, a compiler gives even faster feedback than the tests if something breaks, since compilation precedes running tests. There are also so called *eager compilers* in many development environments that check code as it is typed, and that feedback is almost instantaneous.

Leveraging the support from the static type system is a great opportunity for fast feedback. The early signs that something broke during a change are very useful when creating a Mikado Graph.

Most IDEs for statically typed languages provide great refactoring automation for many atomic and composed refactorings as well, but only as long as the code compiles. Not making use of that was one of the big mistakes we made when we tried to slay the Hydra. When we couldn't compile our code we had to revert to refactoring using search and replace which is *much* slower and more error prone than the automated refactorings of the IDE.

Even IDEs for dynamically typed languages provide refactoring support, but when dynamic language features are used, refactorings might occasionally fail. If a dynamically typed language is used, we once again want to emphasize the importance of automated tests, both micro and macro level tests.

### 8.1.5 Manual testing

In some settings there are teams of testers that manually click through or in another way stimulate the system under test and verify that it acts as expected. We recommend automating the repetitive tasks of testing as much as possible. If this is done properly, the automated

tests enables faster turn-around cycles, and they also contributes to the confidence in the system by acting as *executable specifications*.

Manual testing is still needed, but the time invested should be used to explore the system and go beyond the scripted tests rather than finding errors on a daily basis. Repetitive tasks are perfect for machines. Thinking outside of the box is something humans do a lot better.

Deploying and then letting the users report problems is also a way of getting feedback. We don't recommend this as a strategy to find errors, but releasing and getting user feedback is essential in order to build the best product possible.

### 8.1.6 Pair programming

Our favorite way to get fast feedback is pair programming, as mentioned in *Working in a pair (p.159)*. The co-driver will give us immediate feedback on the code we are writing. We can even get feedback on our ideas in a design discussion, before we write any code. The feedback can be at many levels, from details about the programming language and the shortcuts in the development environment, to philosophy about paradigms and principles of software development.

We always try to work actively to practice pair programming on projects, because we know that will benefit both the project *and* the pairing developers in their development.

### 8.1.7 Dynamically typed languages

Dynamically typed languages require executing tests for verification of changes, since there is no possibility to lean on a compiler when changing the code.

If no tests are in place before making a change, a good idea is to make *Cover area X with tests* a prerequisite to the change in the Mikado Graph when working with dynamically typed languages. These types of tests are often referred to as *characterization tests* [1], test written to define the actual functionality of the code.

### 8.1.8 Balancing speed and confidence

Difficult and complex work is easier if it is done in small steps. If a change can be verified quickly, confidence goes up and so does the speed of development. In order to quickly get where we want to go, the verification has to run fast. In a statically typed language the compiler helps, and on top of that, a really fast test suite can add to our confidence.

There are, however, factors limiting our speed and how fast we can go, for instance the tests. Micro-tests are supposed to be lightning fast, and macro tests are normally slower, sometimes much slower. Even when we have a lot of tests you sometimes want to move slowly forward ar first. As the confidence in the tests grows or is reducded, the pace of the changes are matched accordinlgy. Sooner or later a gut feeling for the test suite and the code base is developed and that makes decisions about the pace of changes come very naturally.

It helps a lot to know how much testing is needed for a particular change, and when it is time to move forward. We have found that increasing the size of the chunks that are altered until the system bites back, and then decreasing them, is a good way of getting to know the system and its quirks. While this balance is highly individual and differs between codebases, working in pairs and with a safety net in the form of automatic tests always boosts confidence.

We start by taking a look at the present situation to see what impact a mistake or traveling too fast may lead to. Some systems are mission critical, whereas some are not. Being safe takes time and energy, while being less safe can cause much damage, and we need to find the balance between speed and confidence.

## 8.2 Technical preparations

There are other technical aspects than tests that should be considered before starting a larger restructuring.

### 8.2.1 Use a version control system

A *Version Control System* (VCS) is one of the most important tools in a software project. A VCS is typically an application that keeps track of the all the changes made to files in a filesystem. The beauty of using a VCS is that it is possible to travel in time, at least back in time. It is possible to revert, or roll back, changes to a specific date, or to a specific tag or label made to name a version. The more often code is checked in, the more fine-grained history there is. Other names for a VCS are *revision control system* or *source control module*.

Even when we are working alone on our pet-projects, we use a VCS to spare us wasted time and agony. There are good VCSs available for free that are easy to set up so not using one is just plain stupid.

The Mikado Method uses the revert functionality of the VCS systematically and extensively, but regardless of whether the Mikado Method is used or not, we always want to have a VCS in place.

### 8.2.2 Know the refactoring tools

Most modern IDEs have automation support for an abundance of atomic and composed refactorings, from Martin Fowler's *Refactoring* [4] and other literature.

When we are working with a codebase, we make constant use of these tools to improve, change, delete, restructure and implement code. Since they are automated, we can make changes that span the entire codebase, changing and moving thousands of files with a single command.

The use we have for these tools when developing cannot be stressed enough, and we recommend anyone to learn how to make use of these in their specific development environment. It is possible to work without them, but we always feel crippled when we do. Every small step takes much longer to complete, and the risk of making errors increase drastically.

### 8.2.3 One workspace

The automated refactoring tools are extremely useful when changing code, but they only work on the code that is open within the workspace or solution in our IDE.

If our code is spread out over several different workspaces, changing the code using these tools can be very cumbersome, since some code is changed with kept integrity, while some is not changed at all, creating compilation or runtime errors.

Before starting to change any code, we try to move all affected code into one and the same workspace, to be able to make as much use of our refactoring tools as possible.

Into a workspace or solution, there are usually a number of projects, representing different packages or components of our product. In *Ch.5.7 Principles of packages (p.125)*, we discuss in more detail how to organize these packages.

### 8.2.4 One repository

We have seen cases where a single codebase is spread out over several VCS repository roots. This makes configuration- and version management more complex, and the risk for anyone involved to make mistakes increases. When making changes across repository boundaries, it is more difficult to maintain consistency and integrity across the different repositories.

For simplicity, robustness, and development performance, we strongly recommend moving all related code into the one and the same repository root. When not adhering to this advice, the benefits from having different repository roots must be carefully weighted against the increased overall complexity of the development environment.

### 8.2.5 One branch

When developing using any VCS, we always work in branches. The checked out code we are working on at our local workstation is an

*implicit branch* of the central main trunk. We know from experience that the longer we wait before checking in, the harder the merge of the branches becomes, since the trunk and our code will diverge from the point when we checked out.

It is the *divergence* of the branches we want to merge that causes us problems. The divergence will increase with the rate of change and the time between merges. When we change a codebase with modern refactoring tools, we can change and move hundreds or thousands of files with a single command, hence the rate of change will be very high. The only variable we can affect is the time between merges, which we want to keep as short as possible. Normally, we merge our local branch with the main branch more than 10 times a day to allow everyone on the project to have the very latest code to build their changes upon, and likewise for us to build our changes on the latest code.

A common, but *bad*, idea is to make a separate branch for the improvement efforts and keep on implementing new functionality in the main branch, with the ambition to merge these two branches when the improvements are done. The rationale is that implementing new features cannot be disturbed by the improvements.

This is far from optimal because the branches have different goals, hence they will diverge even faster. A common and equally sub-optimal solution to that is merging continuously, but only from the main branch to the improvement branch. As the improvements move in one direction and the new features in another, the merges will become more and more difficult, eventually choking the improvement efforts. Our preferred solution is to merge both ways, but if we think of it, that is like having only one branch!

We recommend that the number of 'central' branches is kept at an absolute minimum, preferably only one main branch, and possibly a branch that is created just in time when problems in the production environment arise and need immediate attention.

There are VCSs that can help us with complex merges, provided that there is a fine-grained checkin history in each of the branches. But even then, there will always be cases where that fails. We prefer to use this ability as an extra safety-net when we merge our minimal local changes with the main branch.

Learning how to work in a single branch is one of the things that make companies such as Google, Amazon and Yahoo successful at developing profitable software fast. The most common pattern to enable working in a single branch is the *latent code pattern*, where we make sure that we can turn on and off which features should be used. This means that we often deploy half-finished features that are disconnected from the execution flows. Also called a dark release.

### 8.2.6 Learn a scripting language

To be able to make complex changes, or to ease the automation of tedious tasks, knowing a scripting language certainly helps. The language that the actual system is written in might be a good scripting language in itself, or there might be one that resembles the current development language. Either way, choose a scripting language that handles text and regular expressions well in order to be productive.

### 8.2.7 Use the best tools

Get the best tools for the job and learn how to use them. In a for-profit context, the discussion of whether or not to buy a tool is usually more expensive than actually buying the tool. On top of that, they are usually so cheap that using them once pays the whole investment. If work is carried out in a *pro bono* environment, there is a plethora of free tools that are sufficiently good and sometimes better than their commercial counterparts.

The best tools are not necessarily the most expensive ones. Sometimes it can be more cost-effective to take a simple free tool and extend or modify it, than to buy an expensive tool that doesn't do exactly what is needed.

On some occasions, building our own tools is warranted. Just like for any application, we start out small, with a clear business case, and add functionality incrementally as long as we can warrant it.

### 8.2.8 Buy the books

The interest from investing in knowledge pays good interest, and books are a cheap way of extending knowledge. If the advice from a book is applied only once, the book has probably paid for itself. Managers should keep that in mind, and always approve when their employees ask if they can buy a work-related book.

### 8.2.9 Find the dynamic parts

Hard core refactoring is harder if there are dynamic, or reflective, parts in the code. For both statically and dynamically typed languages, automated refactorings can be put to use, but if dynamic or reflective features of the languages are used, more care has to be taken. It is important to know what parts of a codebase are dynamic, and to specifically verify those parts using automated tests.

Sometimes, these dynamic parts are just needless complexity that can be solved just as easily with proper abstractions. If so, removing reflective parts is usually a good place to start before embarking on a bigger refactoring journey.

### 8.2.10 First consequence, then improvements

It is much easier to make large changes to a codebase if the code is consequent. By having consequent code we mean that variables, members, parameters, methods, functions, classes and packages are using the same standards and idioms. We also mean that the code is formatted the same way everywhere, regarding line endings, blank lines, curly braces, and so on.

Some of this can be achieved with automated tools, whereas others can only be achieved by hard work. This might feel like an unnecessary thing to do when improvement is just around the corner, but standardizing these small things is time well invested, and gives flexibility on a higher level.

This is also true when adding new code to a system. If new code is added with a different look and feel each time, it worsens the situation. If code is added using the existing patterns and formatting, it is often easier to recognize and refactor at a later point. Using the same standards everywhere is a global optimization, simplifying later changes.

Practicing collective code ownership and pair programming is not only a way to spread risk and knowledge, it also helps keeping and getting the code into a uniform shape.

### 8.2.11 Learn to use regular expressions

Most development environments have tools to perform search and replace using *regular expressions*. Regular expressions is a very powerful text matching language, where selected parts of the matched text can be used in a replacing step. Some refactorings are not possible to perform without using regular expressions.

There are a few different regular expression dialects for different platforms, but they usually contain approximately the same set of functionality. To avoid overly complex regular expressions, start by making the code consequent, as described in *First consequence, then improvements (p.181)*.

### 8.2.12 Removing unused code

One of our favorite code improvements is removing code. By removing the code, the system reads better, compiles faster, takes up less disk space and working memory. Of course, the code removed must not perform any work that is relevant to the system.

Removable code is often easily spotted when we read the source files. But, if there is lots of it, it might blur our vision of the system to the extent that removing most of it in a raid is worthwhile. When raiding, we go for the low-hanging fruit, and don't try to clean sweep the codebase. The 80-20 rule, use 20% of the effort and get 80% done, is a good rule-of-thumb.

There are a few categories of unused code, and we have taken some inspiration from horror genre to describe them.

### *Ghost code*

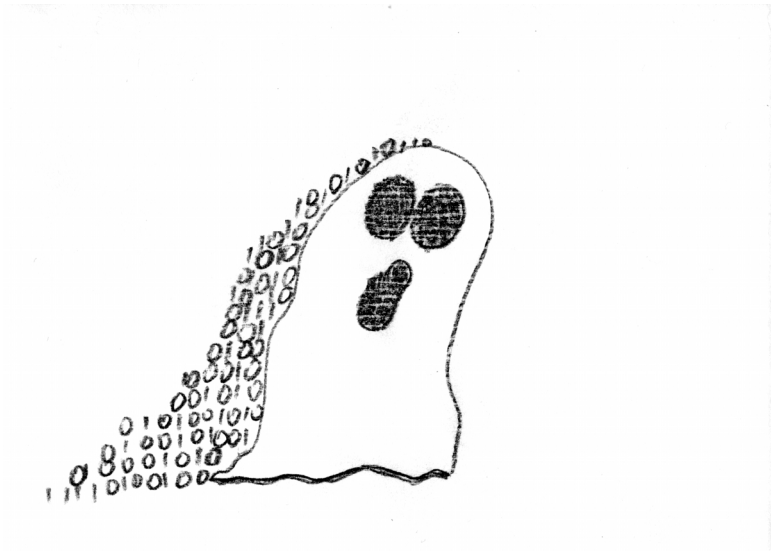

Figure 8.1: Ghost Code

Sometimes we stare at code that is commented out, thinking:

*"Why is this still here? What is it saying? It's probably very important, otherwise it shouldn't be here. Let's keep it."*

Programmers sometimes feel the urge to comment out a particular piece of code, perhaps to short-circuit the execution. But there is no reason for that commented out code to stay in that state for more than a couple of minutes. Code that is commented out is *Ghost Code*, and like any ghost, it should transcend to the other side as quickly as possible. In this case it should be deleted. If it's not, it will haunt us as

long as it remains in this world. On the upside, ghost code is easy to locate with a normal text search.

We recommend using a version control system instead of keeping old code in commented out blocks all over the codebase.

### *Zombie code*


Figure 8.2: Zombie Code

Zombie Code is the living dead code. It is not part of any business value flow, only called upon from automated tests, like unit tests, or other unused code. It is just walking around, feeding off the living code by taking maintenance resources into account, without giving anything back.

This kind of code is very hard to tell apart from real code. It looks like real code, but it isn't used for real. It is also hard to locate, since it is actually covered by tests, and may not be marked by static analysis.

To kill it, we cut off its head, usually the unit tests. This will make the zombie code plain old dead code.

### *Dead code*

This unreachable code is never called, nothing invokes dead code. There are no user calls from a UI, no calls from tests nor any other usage from anyplace else. If a compiling language is used it is somewhat easier to find these dead parts, because static code analysis can help us. Dead code is of no use at all and should just be given a proper burial, by deletion.

### *Finding removable code*

When we look for dead or zombie code, we try to run the system using the acceptance test suite, with code coverage turned on. Or better yet, if the system allows for some overhead, monitor it in production. This way we exercise the parts of the code that an actual user will use. Then we look through the coverage reports to find uncovered code. The covered code is clearly used, and can be ignored. The uncovered parts are candidates for a bit more detailed analysis.

Ghost code can usually be found using a regular expression in a text search. The trick is to look for start of comment, and then any characters that are common in code, but uncommon in normal text. Parenthesis and semicolons are good candidates, of course depending on our programming language.

## 8.3 ORGANIZATIONAL PREPARATION

There is an endless number of ways to initiate and maintain an effort to improve a codebase. We will present some things we like to keep

in mind when doing so, as the Mikado Method is only one part of the puzzle.

### 8.3.1 Mental preparation

Improving code takes time as well as determination and a vision, but most of all it requires courage. Being afraid of changing code is not an option, although respect for its complexity is. Errors will be made, and the important thing is to deal with them as soon as possible. Leading, and especially leading change, can be lonesome at times. Find strength by including the people around you, instead of pushing them away.

### 8.3.2 All things are relative

We try to always be sensitive to what other team members think is problematic in the code. Just because our last project was even worse off does not mean their concerns are not relevant.

The sense of always changing the code for the better is a great way to boost morale and feel good about what we are doing. If we stop improving because the code is in fairly good shape, we will get in trouble. First by losing the ability to improve, second for losing the good morale, and third for ending up with bad code. Our simple advice is to never stop improving the code, no matter how good it may seem.

### 8.3.3 Be aware of the code

To effectively reflect upon the code we create, we regularly need input from other programmers. One of the best ways to frequently see code with fresh eyes is to practice pair programming. Then we get the eyes of the two people pairing, and an additional perspective, namely the one where we reflect upon how we think others percieve our work. Pairing also enables informal discussions of newly produced code, in a fitting context, with a person who most certainly has knowledge that complements our own.

Another way to get a chance for reflection can be achieved by using the Pomodoro Technique [26]. The Pomodoro Technique suggests 25 minutes of work and then 5 minutes of some other activity. By focusing on something else for 5 minutes, the brain gets a chance to restructure new information and knowledge. Compare this to all the bright ideas that come to mind when standing in the shower, or when folding laundry. Pausing regularly makes it possible to use new information sooner and also gives the brain a chance to refocus.

Pair-programming, or pair-work in general, often lends itself to fit nicely with the Pomodoro Technique.

### 8.3.4 Create an opinion about what good code looks like

In order to improve, an opinion has to be present about what constitutes good code. We need to set a standard for ourselves. After that, the hard work begins, the job of keeping the code according to, or above, the standard we've set. Discuss the standard with others, and improve it when a better way to do things appear. The standard will never be perfect, but rather something that is always perfected.

### 8.3.5 Prepare others

We also need to start by communicating more with the people around us and tell them what is about to happen, and explain why there is a need to do something, in terms they can relate to. A good candidate to kick off the dialogue is to review the main goal of the Mikado Graph, and to explain in a broad sense how the goal will be achieved, perhaps showing a draft Mikado Graph. We want to take the time to explain the consequences if this isn't done, as well as the upside of it being implemented. There might also be a need to explain that this is the work of every single person in the company, especially the developers.

Sales teams and upper management often argues that detailed information about the implementation is uncessesary, and we agree. This is where the power of visualization comes into play. For instance, when

two different paths can be taken and both are dependent on a change in the code, a decision can be reached quicker if some up front work is done.

The developers can fairly easy, before meeting with the non technical staff, prepare two different Graphs that respectively shows how complex each change is. This is often sufficient because one can easily tell which one is more complex by merely looking at the size of the Mikado Graph which makes a discussion superflous.

### 8.3.6 The road to "better" often runs through "worse"

When we are working with small-scale improvements, the code improves incrementally and is in a little bit better shape after each change. But when a larger refactoring or restructuring is started, we sometimes feel as if the code is heading the wrong direction at the beginning. The stepping stones leading to better code actually make the situation look worse.

Imagine a situation where a lot of related functionality is spread across several places throughout the codebase, and we move it all into one big and ugly class, just to get an overview. At this stage, the code wouldn't get through any decent code review.

This is only a transient state though, and the code will eventually be moved into more proper places. When we start finding the connections and abstractions in the code, we can create better homes for the pieces of code, making it all look better. This is described in more detail in *Temporary shelter (p.97)*.

This also applies to when we implement new functionality. We want the code to tell us the abstractions we need in order to avoid overdesign. This often means that we create a small mess, just to let the code tell us what the right abstractions look like, and then we refactor to those abstractions.

If we are afraid to make the system look a bit worse initially, we might never find the patterns to make it look really good.

### 8.3.7 Code of conduct

Dealing a lot with legacy code can be frustrating, and it is easy to revert to *code bashing* or *code baisse*, such as "This code sucks!!" or "What kind of an idiot wrote this crap?!?". Phrases like that are often being formulated in our minds, uttered between gritting teeth, or even shouted out loud in teamrooms and over cubicles.

When we bash each other's code, eventually we might stop speaking with each other, not dare to ask for help, and actually become afraid of each other. Name calling only leads to negativity in a downward spiral.

For those situations when we still need to blow off some steam, we recommend having a *safe space* for it, like an agreement with a colleague that it is ok to be very open and direct.

Often, the real problem is not in the code, but in something else, and the code just gets the blame. Finding what we are really upset about often helps relieve some of the agony. Remembering that the code we are bashing might be that of a future friend, a just spouse, or that it might be our own, might help too.

When working with spaghetti code, big balls of mud, or just a plain mess, we find it much better to refrain from calling the code, or its authors, names. We want to keep the focus on developing good code.

And after all, it's just code.

### 8.3.8 Critical mass of change agents

If the behavior that led to our code problems isn't changed, the problems will never go away. We need to change the system, and that requires a lot of effort and determination. On a software development team, our experience is that between 25 and 50 percent of its members has to support a new way of working, to make a sustainable change.

If we read the fifth principle of the Agile Manifesto, it states:

> Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

It is the same for an improvement effort, find the people that want to make the improvements. If the goal is to gradually restructure the code, we want to pick the people that desire to work that way. If they cannot be found inside the company, our advice is to search outside the company, for consultants or for new employees. To improve the chance of finding the right people, we recommend letting the internal improvement change agents select the people they want to work with.

Detailed advice on that is way beyond the scope of this book, but a recommended read is *Fearless Change* [33] by Linda Rising and Mary Lynn Manns.

### 8.3.9 Collective code ownership

In eXtreme Programming [25], collective code ownership is mentioned as a practice, and we think it is one of the cornerstones of software development teamwork. It basically means that any person on the team in a software project is allowed to change any part of the code.

The opposite would be giving some people the exclusive right to change certain parts of the code, by conventions in a team, or by rules and procedures in a company. Denying developers access to some parts of the code might seem like a good idea at first glance, *what they can't change they can't ruin*. The problem is that when we restrain access to code, this increases the resistance to change it. Necessary restructurings become very tedious to perform, even to the point when building large workarounds is easier than fixing the real problem.

When doing larger restructurings with the Mikado Method we can end up in any part of the codebase and the system, and we need to be able to perform the necessary changes. At the same time, we must of course make sure the users of the system are not affected.

### 8.3.10 A ubiquitous language

We always strive to bring technical and business people as close to each other as we feel is needed by creating a common language and understanding for our domain. We also try to incorporate this language

in the code represented as names on classes, packages, methods and functions.

This often requires a closer collaboration between the developers and the business side than most organisations are used to. This is however outside of the scope of this book and we recommend looking into Eric Evans' *Domain-Driven Design* [3] and Gojko Adzic's *Specification by Example* [37] for further reading.

## 8.4 Measuring code problems

There are many tools out there that provides help when analyzing a codebase, and plenty of relevant metrics that can give a hint of the state of the code.

When we get confused by how many ways the execution flows due to conditionals, *cyclomatic complexity index* [18] will give us a number on that confusion.

When our minds are spinning from dependencies going back and forth, or when we want to understand the reason for some design principles in *Principles of packages* *(p.125)*, we look mainly at the numbers from Robert C Martins book *Agile Software Development* [12].

When we need to know how much we can trust the tests, we first look at the quality of the tests, then we check the *code coverage* of the tests. If the test look good in that they assert the right things, and most of the code is executed by the tests, all is good. Otherwise, we have to find what parts are not tested, and what the impact of that is.

All of these metrics are good in that they provide numbers we can learn from, and see how they change as we make changes to the codebase. They can give us some idea of where we might have problems, but there is more to it.

The Mikado Method starts with what needs to get done, and even though something scores well on complexity, package coupling and coverage, we might have to change it. At the same time, we might never have to read or change the parts with really bad numbers.

In addition, our *gut feeling* often tells us where the real problems are.

It might take some initial working with the codebase to know, but we usually bump into the worst problems every other day. Code metrics are a good way of getting some backing for that gut-feeling, but we would not base a restructuring effort on bad metrics alone.

### 8.4.1 An awakening

When Melinda started out programming she was doing a lot of web development. At that time there was not a lot of fancy web frameworks so there was a fair amount of text parsing, database handling and even some low level HTTP-stuff. The only frameworks available were the ones that she herself had developed.

> Melinda remembers: "I was working on a rather big web project. This time we had developed some sort of action classes, a dispatcher and support for views. Very basic stuff though. But it still made the development go faster."
>
> One day I was struggling a bit more than usual with some difficult code in a view and turned to a more seasoned colleague for some help. I explained the logic and the flow of the code. How this affected that and how the state of this parameter made the program enter this code. It was a huge mess with nested conditionals, state scattered all over the code and way too many responsibilities in one and the same class.
>
> After 10 minutes of explaining I looked at my colleague and sort of waited for him to point out what he had missed. I did not get the response I was looking for though.
>
> 'Melinda, this is too complex, I don't get this', said her colleague.
>
> 'It doesn't have to be this way ... '
>
> What, did he mean by that?
>
> It took at least five minutes for that to sink in. Maybe even a week for it to *really* sink in. What did he mean? I don't need to have 20 if-statements? How am I supposed to do it any other way?

> Looking back at this incident, with years of experience, it's easy to laugh. But for me to see this back then, while I was digging an ever deeper hole, creating a more complex mess every day by adding one if-statement at a time, it was a totally different story. I had a blindness to flaws that made me incapable of seeing that I was creating a complex mess. In addition to that, I had insufficient knowledge about what good code is supposed to look like. I also lacked that hard-to-describe skill that experienced programmers have, the ability to distinguish good code from bad code and that can be harder than one first think.

### 8.4.2 Gut-feeling

Nicely formatted code, well balanced comments and documentation could at first glance make the code look good. But it might also be a way to just 'pimp the code', and trick its readers into thinking it is better than it actually is.

Many developers have a good gut feeling for code, but without experience, that gut feeling is hard to trust. With some years of experience, some start to understand that they are actually right when they feel something is overly complex, while others just seem to get numbed to the pain.

We try to listen to what we feel about the code, questioning how things are done to find better ways. We don't want to get numb.

## 8.5 When the crisis is over

Once we have gotten our heads above the surface, we want to keep them there and start swimming, and eventually get dry land under our feet. The following sections describe the behavior needed to get out of the water and stay there. This is the behavior that takes a little bit of effort to keep up, but the joy of working in such a way gives much more energy back when things start to look good, feel good and work well.

Most of this is not Mikado specific, but just good software development practice.

### 8.5.1 Refactor relentlessly

For every change that is done to the system, we try to take a step back and see if there are more parts that can be simplified, split up or removed completely.

There are some great books out there describing smaller refactorings [4], refactorings to patterns [5], clean code [7], implementation patterns [8], test-driven development [9]. There are also great tools in most development environments that makes it easier to carry out a lot of those smaller refactoring and restructurings.

All this knowledge and all these tools are of no use unless they are put into practice. We often fiddle with our tools to see how they work. Playing around is without a doubt one of the best way to to learn how to perform refactorings and restructurings.

### 8.5.2 Design never stops

A software system is a model, or several models, of the real world, usually modeling the business we are developing for. As any model, it represents a few concepts of the real world to be able to perform some tasks. The real world will change, and so will our understanding of it. Our model should then change too, to best harness our knowledge, to favor our business.

For that reason, design is something that never stops. There is a constant need to rework the system, introduce new concepts and think about how to make the application serve its purpose most effectively.

Also, the natural habitat for a software system is in use. If the software isn't making money, saving money, protecting property or avoiding costs, little benefit will come out of it, and thus it has very little value. This means that we need to be able to stay out of trouble at the same time as we keep deploying new increments of value into production. This require us to design as we go.

### 8.5.3 You ain't gonna need it - YAGNI

This is one of the most important rules of software development. We never develop a feature because *we may need it later on*. We most likely won't, and if we will, we just implement the change then and there, later on. Few things add complexity as speculative development, it only makes it harder to change the code in the future. The cheapest software to write and maintain is the one that is never developed.

### 8.5.4 Red-green-*REFACTOR*

The test-driven development mantra is *red-green-refactor*, and it describes the three parts of test-driven development. First, we write a test for the functionality we want to implement. Then we run the test, expecting it to fail. When we run the test in an IDE, there is often an indicator or a progress bar that turns red to signal failing tests, hence *red*. Then, we implement just enough code for the test to pass, nothing more. This results in the indicator turning green as the test pass, hence *green*. If all tests run, we can safely *refactor* to remove duplications, re-organize our code and clean up. If any test would fail after such a refactoring, we revert the code to the last green state, and continue from there. This approach makes sure a lot of problems are discovered before the code reaches the other developers, or the users.

The tests created are good to have, but the refactoring step is immensely important. If we omit it, the code will be tested, but complexity is building up in the codebase. Eventually, we cannot refactor the built up complexity within a single task. This might force us to take on even more complexity in a workaround, and then we need to ask our stakeholders for permission to plan time for a major clean-up. We believe it is better to continuously refactor.

### 8.5.5 Continue using the Mikado Method

Even though the crisis is over, there are usually more things to take care of in a codebase that just got out of a crisis. The Mikado Method can of course be used for day-to-day work on non-trivial tasks as well.

In addition, it is good to keep our tools sharp, just in case we need them. When we use the Mikado Method continuously, we are up to speed when we *really* need it.

## 8.6 SUMMARY

Jake: I should go fast, but not faster than my feedback allows.

Melinda: That's right. Automated tests and the compiler can help you. Also, good preparations will allow you to go faster.

Jake: Sweet. And there is more to it than the technical parts?

Melinda: The technical parts are the most important, but preparing the people around you will increase the chance of success.

Jake: And when am I done?

Melinda: You are never really done. After you have saved your system from a painful death, you have to constantly make sure it is taken care of to minimize the risk of having to save it again.

Jake: Oh...

Melinda: What you can do is try to find some metrics that tell you how you are doing. But metrics can never be the only information to use. You must train your gut feeling about when things are going in the wrong direction.

Jake: Experience again.

Melinda: Yes, You can't get away from that. Experience matters. But the first step is awareness, and now you've got that part under your belt.

Jake: That feels good!

Melinda: I'm glad to hear that!

## 8.7 Try this

- What preparations do you have in place?
- What preparations would you need to do before starting?
- What preparations would you omit? Why?
- Which of the practices do you lack the most?
- Is there any preparation that feels more difficult than others? Why? What would make it easier?
- How do you verify your application?
- What metrics do you generate from your codebase? Do you know what they mean?
- Find the automated refactoring tools in your IDE, sometimes you might need a plugin. Try each of them. Think if you can find an occasion where they would have been useful.
- Find the regular expression option in the search dialog of your IDE. Write different regular expressions to locate things in your codebase. Can you find what you expect? Does formatting everything improve your success?
- Try to write regular expressions to replace code in your system, using parenthesis to pick parts of the found string to use in the replacement. Especially, find the negating match and use it.

# Chapter 9

# Technical debt

Jake: I feel like I'm set free, only to discover that I'm in a larger prison.

Melinda: That was poetic. What do you mean?

Jake: Now I need to get my boss to understand why we are drawing all these graphs all the time, they are starting to whine about it. Just hacking in a new feature takes longer than they want, but also puts us in even more trouble. Fixing things along the way takes even longer, but will make us go faster in the longer run. They have no clue what bad code does to productivity!

Melinda: That is a very common reaction.

Jake: So what do you do about it?

Melinda: Almost everyone can relate to financial stuff, like loans, debts and investments. You could use that metaphor to explain.

Jake: How do you mean, exactly?

Melinda: I use the term technical debt, it has served me well in the past.

Jake: Ok, sounds easy enough, tell me about it ...

## 9.1 Loans and debt in the real world

Outside of the software world, we take loans to finance different sorts of investments or expenses. We then have to pay back the principal, the money we borrowed, and pay an interest, a time-based fee for loaning the money. We take loans both privately, and in business-related situations.

Privately, we can finance a house, a new car, a boat, or a summer house by taking a loan that we pay back over a longer period of time. We also buy smaller things like new clothes, a TV or a stereo. If we run out of money by the end of the month, or the things we buy are a little bit too expensive, we might pay using a credit card. Using a credit card is like taking out a short term loan.

Sometimes, things just break, or external events such as accidents or forces of nature destroy our property. Even though it is not by choice, we might just have to replace or repair the wrecked property, and finance it with a loan.

In a business situation, we can use loans to make investments in new machinery for instance, or a new factory, hoping that the investment will pay back with an interest and some revenue. This is often referred to as return on investment or ROI.

In either case we are *in debt* for a while, and we will have to pay back the loan, usually with interest.

### 9.1.1 Why take a loan?

The basic idea behind taking a loan is to push the payment of a purchase in time, from when we don't have the purchasing power to a time when we do. Sometimes this means that we pay back a little at a time over a longer period, perhaps over decades for expensive things such as a house or a summer cottage. Sometimes, it means that we will pay back everything in one big lump within a week, or sooner.

When we borrow money to buy new machinery for a factory, we expect that the new tools will lead to a more effective production, enabling us

to build products faster, or that the new machinery is more efficient, and will save us money.

Either way, we expect that our investment will pay both the principal and the interest in the long run and maybe also increase our margins on top of that. Our investment should pay off, and in the end earn us more than we put in at the beginning.

Some of our motives are based on very basic needs, for instance when we buy a house it's because we need some sort of shelter i.e some place to live. If that house is a long way from work we probably decide that a car will make our life easier. These kinds of loans are often preceded by calculation on how to afford the periodic payments based on incomes and interests.

In some other cases, the motives are more based on a desire, like when we already have clothing that will keep us warm, but we want another set of clothes just because we like them. Or, when we want an expensive new TV set just to outshine the bragging neighbor, but we don't have the money right now. These motives are more consumption-oriented and they have little to do with investments since we rarely get more money from selling used clothes or TV sets than we originally paid for them.

## 9.2 Debt in software: technical debt

Imagine that we have added some functionality to our codebase, but left the code in a slightly worse state than we found it in, a state that is not optimal for the current situation. We also know that we will be doing more work in the same area in the near future. This makes another change to that particular piece of code a bit more time consuming, because it will likely take extra time to understand what the code does, and we will have to edit more code to implement the next feature.

By behaving this way we have just, metaphorically speaking, put ourselves in debt. Our future selves have to spend a little extra time to understand what the code does and put some extra effort to into making changes. This *extra* is equivalent to *interest*. The *principal* of the loan is the time it takes get the codebase back into shape again. This is why it is sometimes called *technical debt*.

> ***Technical debt*** is any existing technical structure that prevents us from effectively and efficiently make relevant and necessary changes to our software. The *interest* of the loan is the extra time we spend to deal with those structures. The *principal* is the preventing structure in itself. The *repayment* is the time it takes to refactor the structures to be fit-for-purpose.

The phrase *technical debt* was originally introduced by Ward Cunningham in the experience report "The WyCash Portfolio Management System" at OOPSLA '92 [14].

We, and we are not alone, use the metaphor 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 are unaware of the consequences of an ill-structured codebase, but they are well familiar with the do's and don'ts of debts. In general, they are terrified by the fact that they were put in debt without even knowing it, then grateful for the new insights, then slowly frustrated when they realize they have to pay it back.

### 9.2.1 Consequences of our definition of technical debt

If we don't need to touch a part of our system, that part might be some of the debt itself, but we don't need to pay any interest because we never need to deal with that part. It is likely unwise to repay that debt, since there are usually more urgent needs in other parts of the system.

Generally speaking, not all parts of a system will be well written, and by having well-tested modules with clear and limited responsibilities and apt interfaces, the risk of having to pay interest on those modules decreases.

From a debt point-of-view, it doesn't matter how we got into debt. We still have to pay the interest every time we come across that part of the system. If we ever want the interest payments to go away, we need to repay the principal. *Debt doesn't know where it came from*, but from an improvement point of view, it is important to know where it came from.

### 9.2.2 Lactic acid

Debt is not inherently bad, but if it isn't handled properly it will eventually introduce more problems than we bargained for. To illustrate this, we look at software development through our sports glasses for a moment.

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

The price to pay for a strategy like that is an increased load on our system. If we don't get a breather shortly thereafter, the lactic acid in the system will serve just the opposite effect, it will slow us down. If we go for too long without clearing the acid, we might even hit the wall and collapse.

Deliberate use of bad structures or cutting corners can be used as a tactic to gain a temporary advantage over the competition. It can be viewed as the equivalence of spurting, which will result in the accumulation of lactic acid, i.e putting ourselves in debt. Used properly it can truly be the performance booster, but using it as a strategy will bring us to our knees.

The body has a natural process for ridding the system from lactic acid, but software doesn't. If we want to keep our product in a healthy state, we need to take into account the time to rid the codebase from the accumulated "lactic acid".

### 9.2.3 What are the interests?

Kenny has several loans; one on his house, 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 had come across some money, where should he start to pay it back?

In this case, it might look rather easy. Paying back loan sharks is likely

to be a higher prioritized activity since their kinds of loans come with a very high interest, along with very persuasive methods for getting their repayments.

Technical debt in software projects also comes in different shapes and with different interests, but the interest depends on what we are set out to do, not what type of debt we have. In fact, the same type of debt might have different interest rates at different times. It can be zero interest if it is in a part of the code we hardly ever touch, or of "mob-interest" if it stands in the way of delivering important features.

When we spend time paying back debt by randomly refactoring code, we might be prioritizing the loan to mom and dad before the loan to the mob. In software development, the difference is much more subtle. In order to identify our mob-debts, we need a method to find them. The Mikado Method helps us sort out the critical paths in our change effort, to find our "mob-debts".

## 9.3 How we get in debt

As a metaphor, debt is quite easy to understand, but 'debt' has a bad connotation. Even though debt doesn't know where it came from, understanding where different code problems come from can help us mitigate the root cause, or perhaps just get us at peace with the stressful situation of having to deal with it.

There are several ways to categorize debt to bring some order. The following sections divide debt into acceptable, unavoidable, unnecessary, and bad. These are not clear-cut categories, but way of looking at the origin of technical debt.

### 9.3.1 Acceptable debt builders

Acceptable debts come from some sort of gain, short or long term. The best would of course be to not be in debt at all, but in the context of debt, these are the good ones. Debt doesn't know where it came from, but from an investment point of view there can be a huge difference in where debt originated from.

### *Making a trade-off to hit a market window*

One category of investment-based technical debt is when there is a known conflict between creating the most apt technical structure and a delivery dead line. If there are considerable financial or market position gains to be made from making the deadline, the cost of having to deal with a non-apt implementation might be worth it.

An example of this would be when we quickly implement new functionality by adding another couple of conditionals, like *if-elses* or *switch-statements*, instead of taking some more time to create the proper abstractions. The prior might just mean bloating existing conditionals a bit more, adding to the overall complexity of the application. The latter could mean that parts of the system need restructuring to use the new abstraction.

When we are faced with a delivery dead line with a conflict as the one described above, debt can be accepted to make the delivery, while knowing that the debt should be repaid after the delivery. This is a bit like drawing lactic acid and letting the body rest to get rid of the acid.

Alternatively, we accept paying interest on that piece of code until we make pay it back the loan by refactoring the code. If deadlines are constantly stacked up and there is never any time to repay accepted debt, this is not an investment any more - it has turned into credit-financed interest payments. It will keep us floating for a while, but sooner rather than later, the debt has accumulated and we risk having a seriously crippled delivery ability. As with lactic acid, if we do that for too long, we face the risk of hitting the wall. In some businesses this means closing down the shop, for most companies it means getting a worse return on invested capital.

### *Waiting for the right abstraction*

Avoiding debt often boils down to finding the right abstraction to avoid duplication and finding the right semantics for an application. In some cases, the abstractions, or generalizations, are obvious from the domain. Other times, we need to make a few different implementations to see where we can abstract and generalize our implementation.

This means that we might incur some temporary debt while waiting for the epiphany of a good abstraction. We believe this is a good thing, because jumping to conclusions can cause an even worse situation.

#### *Learning*

Most developers always have new ideas on how they should have done it after they are done. They still did it the best they could at the time, but from learning about the domain and the system, they know their system could be even better off.

This is good because it implies that we reflect over our work, and our solutions. With this knowledge, and perhaps using the Mikado Method, we can morph the system into this new and better shape.

### 9.3.2 Unavoidable debt builders

Some things in life just can't be avoided, and especially not in a real-time business situation. There just isn't enough time to get everything in place, things are the way they are because they got that way, or things just happen outside of our control.

#### *Working on the edge of chaos*

Software development is about constantly making decisions of which path to choose. Generally, the less chaotic the environment and the more informed the decision maker, the less is the risk of making bad decisions. Unfortunately, software development is a near-chaos field and the problems are ill-posed, at best. This makes it very hard to create the solutions that we in hindsight will consider the best, or even good.

Our best ways of avoiding trouble is to expose problems early, reflect upon them and take action as swiftly as possible. Symptoms get harder to see the longer we are exposed to them. This part of the book is here to remind us to step back for a moment and take a look at the

situation, how we got where we are today, and maybe see things we haven't already seen.

### *Inherited debt*

Often, we get to handle troubled legacy code written by others. This is not a delight, but something we have to deal with as a professional software developer. If the people involved in creating the inherited version are still around, we can get a hint of why things got the way they are. The reasons may still be valid, and something we have to be aware of to avoid ending up in the same situation again.

### *Staff turnover*

Figure 9.1: Staff turnover

Whenever we visit code for the first time we can feel awkward. We are not 100% sure what it does, or how to implement things. It always take some time to get up to speed on a codebase. When replacing members on a team, some unique knowledge goes missing. The new people, especially when under dead line pressure, might add things in ways that add to our problems.

Staff turn over is in general unavoidable, but it isn't bad *per se*. Short term there might be a dent in productivity when getting someone up to speed, but long term the added knowledge and perspective often improves speed and structure in a solution.

### *New regulations*

Changed regulations or laws can sometimes require significant changes to the architecture and implementation of a product.

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

These kinds of changes are hard to predict. When we know we are working in a segment that is heavily regulated, we can only prepare for it and not anticipate it. We always recommend automating as much as possible of building, packaging, documenting and verifying our product. When it comes to these types of environments, it cannot be emphasized enough.

### *New requirements*

New requirements might come from fiercer competition, sales, or an updated vision. In order to keep up with competition, extend our advantage on the market, or accommodate new sales, our company updates its visions and goals for the product we are developing. If we are unlucky, we could end up in a debt situation, where the previous solution worked just fine, but the new requirements invalidate previous

designs and architecture. Up until that moment, the application was fit for purpose, but from then on, adding more features in the old way just adds to the trouble.

Sometimes there is a wish to be a full service company, i.e offering basically the same thing as competing companies do. We once ran across a young online bank who decided to add another type of account to their offerings. What once was a codebase rather fit for purpose immediately felt a bit more cumbersome and unweildy.

*Changes in demand*

The market, or the customers, do change their minds from time to time. Things go in and out of fashion, new technology makes obsoletes existing products. There is usually little we can do about it but try to adapt.

### 9.3.3 Unnecessary debt builders

Some debt originates from the excessive ambition of developers, lack of experience, or systemic problems that come from the way things have become. These are often unnecessary, but not inherently bad, since there is a positive and ambitious undertone in the activities that caused the debt.

*Stress*

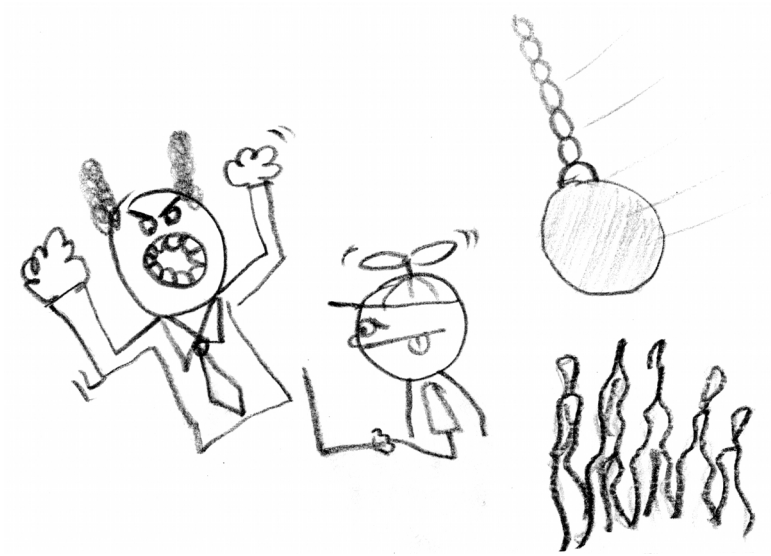

Figure 9.2: A situation of stress

Stress makes us less aware of the current situation, creating a tunnel-vision where decisions made are often based on an incomplete or distorted set of data. 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 invites us to take short cuts, make unnecessarily short-sighted solutions, adding to the problems of the solution. Sometimes we are aware of the stress, but other times stress creeps up on us.

Excessive stress, frequent or extended, is often a systemic problem and can only be solved with changes to the system. Are the developers in the team pushed towards deadlines without being able to affect their own situation? Are the teams not responsible for the 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. Not caring, or apathy, would be much worse. Management is always responsible for creating a healthy environment. The excellent book Behind Closed Doors [24] gives great insight into how to become a better manager. But, the problems cannot only be blamed on the system or the managers; there is a personal responsibility to warn when working conditions are unhealthy and trouble code is created.

### *Insufficient knowledge*

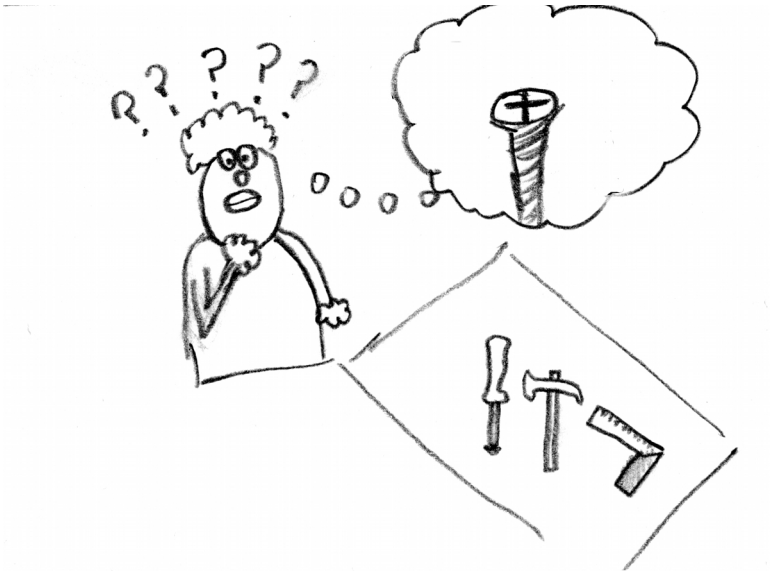

Figure 9.3: Insufficient knowledge

Sometimes lacking of tool or programming language knowledge creates an implementation that is unnecessarily bulky or just not the way to do things. There are often libraries for, and constructs in, languages that simplify a solution. When good technical knowledge is not applied,

solutions tend to get unneccessarily rigid and complicated.

Software developers with no, or poor, understanding for the business and the domain they're working in are likely to produce worse results than a person with a good understanding. In the same way as the lack of information might contribute to bad constructs, insufficient domain knowledge leads to similar results because we are missing important pieces of information. We try to get together with the business side, or the customers, to learn how they talk about and interact with the domain.

Once at a university, a graduate student founded his entire doctoral thesis on the wrong assumptions. This was discovered when he presented his thesis, rendering most of it trashcan material. However, he was of the persevering type, started afresh, and is today a professor at the university.

The lack of correct information can lead to anything from minor defects to having the wrong assumptions laying the ground for important parts of a product. The cost to fix this at a later stage might be high hence we try to verify our assumptions with real code, real tests and real users using the the system as early as possible.

### *Premature optimization*

Making optimizations before we have tested our application with real data often leads to an overly complex design and rarely targets the right bottleneck. In a complex system, it is virtually impossible to foresee where the bottleneck will be [21]. So, we structure applications well and write as clear code as we can, then we optimize if we must. Surprisingly often, good structure leads to good performance as well.

### *Not invented here*

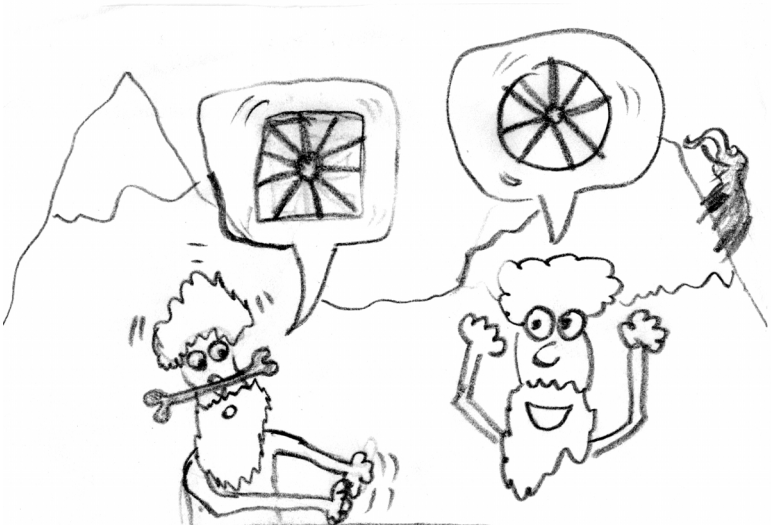

Figure 9.4: Not invented here

The 'Not invented here' syndrome manifests itself as an unwillingness to use already existing products or knowledge, such as using existing software components or already made frameworks. Time and energy is spent on writing proprietary code for components that are commonly available, instead of focusing on the core business. Since the component is not core business, the maintenance of it is likely to suffer, sooner or later making it a debt rather than an asset.

### *Proudly found elsewhere*

The opposite of 'Not invented here' is 'Proudly found elsewhere'. This leads into a *framework frenzy* where no piece of code can be developed, but everything must be about analyzing and choosing an existing system containing the feature. The development becomes a *scrapheap*

*challenge* where all the parts do what is needed and a million things more. To navigate and improve such a system is often hard because the implementations are not changeable and just penetrating the used system means wading through unnecessary complexity.

The decision of an introduction of a framework must be balanced not only with the cost of developing our own system, but also the cost of decreased control and increased complexity. In Strategic Domain-Driven Design [3], Eric Evans explains when an existing framework is more effective, and when we should consider developing our own. It comes down to drawing the consequences of the strategic mission statement for the product. If the code is what creates our competitive edge, we should develop it ourself, even if there is software available that does the job. We must be able to adapt and change the software to keep that edge.

### *Vendor lock-in or framework leakage*

When using a framework and letting that framework leak into most corners of our codebase, we are suddenly in a *vendor lock-in* situation. We have let the third party library take over our application instead of using it to solve a problem.

If we want to use a third party library, we want to let the abstractions come from our domain, then we implement that abstraction to encapsulate the library. That way, we won't let the library dictate our implementation. The library merely becomes a tool to assist an implementation of a specific problem and will be much easier to replace if necessary.

Some vendor lock-ins can be acceptable, such as choosing a platform to use. Care should be taken so that for instance licensing for the platform doesn't limit the development, test, or deployment of the solution.

### *One-way code*

Sometimes codebases are unnecessarily restrained in that we can only do things one way, and one way only. Where abstractions are used,

they are used to limit the freedom of an implementation rather than creating options. Performing any task, however simple it may be, might involve changing the entire codebase. When the customer asks for a new feature, it is just not possible or feasible to implement it within the existing architecture. It is really hard to do the right thing. This is *not* a template situation for being productive in creating good software.

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

One-way code is riddled with limitations. Either the author deliberately restrains the flexibility without having a business case for that, or it just happens due to the programming style of the author. In either case, if these restrictions are not 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 least 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 little constraint as possible on the code to minimize the risk of having to change it when adding more code.

To resolve such limiting code, there often are a lot of dependencies that need to be dealt with. This is a situation where the Mikado Method can help to get on the right track.

### *"Clever" code*

"Clever" Code (n.b. irony) is code where the author has used all available tricks known, creating a solution that is way more complex than the problem solved.

> Melinda was once in a project where she and her 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 it's cleverness, but because they had such a hard time understanding the code. They just handed over the code to its originators with a 'We don't know what to do, you guys will have to fix this yourselves!'.

One would think that a clever solution is preferable. After all, 'clever' has a positive connotation. But as programmers we would take a clear, easy to read and understandable solution over a clever one any day.

*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 so, clearness is more important than removal of all duplication, because exaggerated use of duplication removal might result in unreadable, *shrink-wrapped* code.

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

To deal with 'clever' code, we change the program piece by piece by making sure the intentions of every piece of code are clear. This will be like unrolling the implementation. We might have use for the Mikado Method in doing this, but more often it is work on a smaller scale, method by method or class by class.

### 9.3.4 Bad debt builders

Some debt is bad in the sense that it comes from an lack of respect for the customer, co-workers, managers, employees, or for shared resources. It is not always a clear-cut line between any of the debts in this listing, but when we see these types of debt we know for sure that the problems go deeper than just the technical parts of the application at hand.

#### *Lack of communication*

When different parts of an organization isolate themselves, or formalize their communication excessively, the other parts of the organization build structures or behaviors around that. Instead of people just talking to each other, we get not one, but *two* unnecessary filters between the people doing the communication. This leads to distrust, disrespect, us-vs-them mentality, and poor flow of information. In turn, this leads to the parts respectively trying to solve the problems themselves, creating multiplied structures for the same thing within the organization, often including technical solutions. That is yet another course that leads to technical debt situations.

#### *Technical policies*

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

In the eyes of the manager of the operations department, this is often

a rational decision. But in the eyes of providing the best value for the money, or to be able to compete with the best in the market, this could be catastrophic scenario. Sometimes, it will work just fine, or the policy product is sufficient, but in other cases this will incur a debt that will take a more or less serious toll on the profitability of the product. This means that an interest has to be paid throughout the lifetime of the product, and all because of a policy rather than a case by case decision.

We believe that in any competitive market, technical decisions must be made based on the product and its usage. Any technical policy should be a preference rather than a dictate.

### *Unprofessionalism*

Technical debt is not a free ticket to deliberately letting a system or development environment decay into a mess, while blaming it on dead lines. We should always do the best we can and apply good development practices such as just-in-time design, test-driven development or the like, write clean code, take care of the continuous integration environment and so on. To not take this responsibility for a system is just plain unprofessional.

The positive thing is that the knowledge of what should be done actually exists, it just has to reach all the way to the system.

From time to time we might create some messes anyway. We have to bring them up and fix them right after the next release, in a professional manner.

### *Habitually cutting corners*

| | |
|---|---|
| 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... |

We have all been asked for estimates, just to get them lowered in the hope that this will result in work being done faster. To be done sooner, do less instead. When we cut corners we are starting a downwards spiral, where we steadily add to the unnecessary complexity of the application. It is very easy to end up in a situation where we compensate for previously cut corners by cutting new corners, adding complexity to complexity.

This is like taking credit to pay the interest on our loans. If we don't started improving the code, but keep cutting more corners, we will get deeper and deeper in debt for every change we make.

### *Breaking windows*

In criminology there is a phenomena often referred to as *the broken window syndrome*. This phenomena was first presented in an article called "Broken windows" by James Q. Wilson and George L. Kelling [27]. 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 on 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. Just like improvements, deteriorations often start with one small thing, like a broken window.

When we face a situation where we already have messy code, it's a real demotivator. Creating trouble-free code in that situation is very cumbersome. The easiest way is to fix the codebase by fixing one class or module at a time, and then to not break them again.

### *Lack of respect*

Sometimes a team member knows what is expected, and is capable of complying with those expectations, but still he or she plays by private rules. This can be anything from checking in code that doesn't compile to refusing to spread knowledge to other developers. This is a way of saying "I'm more important that you", and for a software development team that kind of behavior is bad for several reasons. It will make the current state and the expectations more unclear. It increases the risk of having different rules for different parts of the codebase, adding to our problems. It also causes adding unnecessary structures just to deal with the rogue developer.

Sooner rather than later, this will lead to disagreements, and the chances of resolving those disagreements in an environment without respect are slim.

### *Unresolved disagreements*

When there are unresolved disagreements, for instance on how to implement certain things, there is great a risk that we will start building technical debt. If a team cannot settle on an implementation, our application is likely to have two or more ways to implement a feature. It may be that every way of doing it is perfectly valid, and the difference is a matter of taste. 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 infected, 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 are not resolved.

## 9.4 SUMMARY

Jake: Oh, man. It's seems like we have a lot of things to discuss in our team.

Melinda: Yeah, you probably have ...
Jake: Only ... is it always possible to solve the problems with the team?
Melinda: To a certain degree it is but most of the time problems must be attacked someplace else
Jake: Right, so when I need to tidy up code, it doesn't necessarily mean I was responsible for making it a mess in the first place?
Melinda: That's right, you're probably a victim of a systemic failure ...
Jake: Systemic? You mean I'm just a cog in a machine?
Melinda: On the contrary ... you're part of an organisation, which is a system
Jake: Oh, what a relief ... Well, I guess that means I can make a difference only it might take a while?
Melinda: Yes, let me show you what can be the cause of your headache.

## 9.5 Try this

- Can you think of times when you introduced technical debt in a system?
- What types of technical debt can you see in your current, or a previous, system? Can you find some from investment, unprofessionalism, ignorance, external events or learning?
- What types of events caused your debt?
- What parts of your code is troublesome to read or change? Do you read that code often?
- For a piece of code that has some debt, can you estimate how much less time you would spend there if you could improve the code? What would that improvement constitute? What stops you from changing it?

# Chapter 10

# Origins of change

Jake: I can't be responsible for all this ...
Melinda: Do you mean the technical debt?
Jake: Yes. I do my best, but it seems like bad code is appearing from nowhere.
Melinda: Well, that bad code was put there for a reason.
Jake: But the abstractions are just ... not good!
Melinda: Maybe they once were?
Jake: You mean they once where but now the world has changed?
Melinda: Yes, that exactly what I mean.
Jake: Ok, so blaming all kinds of problems on others might not be what I want to do. But making decisions that serve all can't be done with a method, right?
Melinda: The Mikado Method can't stop enemies at the gates. Do you have a minute ... I want to show you something else. It's a model I sometimes use.

## 10.1 Influence and type

As we can see, technical debt does not only come from technical work. It also comes from the surrounding organization, from the integration

of supplied APIs, or from the greater society we live in. Another way to look at technical debt is to assess it in *Influence* and *Type*.

**Influence** is to what extent our organization had control or influence over the process. Was it externally imposed, out of our control, or was it internally incurred, created by ourselves in our organization?

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

We can visualize this in a matrix, such as in *Fig. 10.1 (p.224)*.

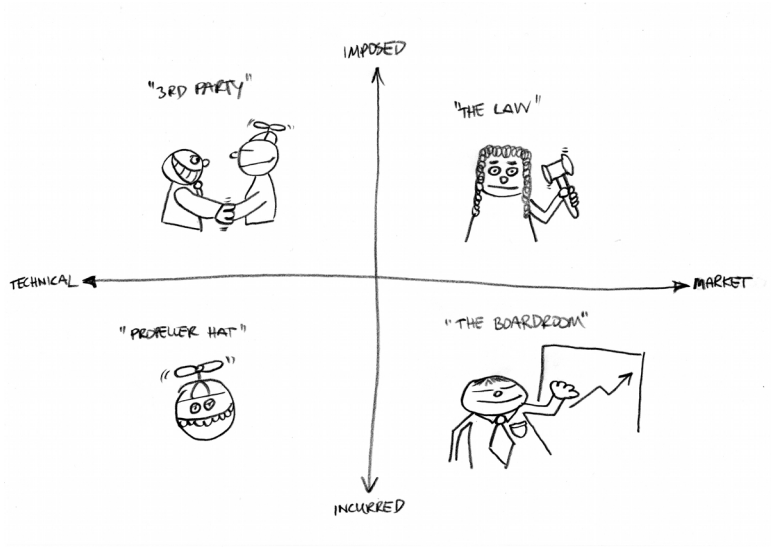


Figure 10.1: Influence and Type Quadrant

The matrix consists of four categories. Let's look at them a bit closer starting with the upper left corner.

### 10.1.1 Third Party - Technical and Imposed

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

Debt from this column can be very expensive to pay off, depending on how our system is structured. But still, it must be done if we don't want to throw our solution away and start afresh, which often is a bad decision, see *Rewrites (p.140)*.

We were once developing a product in which a map engine was integrated. Unfortunately, there was a bug in the map engine version we had. There was a bug fix available, but only in the latest major version, where the APIs had changed significantly. Since the old API had leaked into several places of the application, there was quite a lot of work to refactor those parts of the code. We had put ourselves in a bad situation. We had ended up in a *Vendor lock-in or framework leakage (p.214)*.

If an application is built up by calls to different frameworks, like described in *Proudly found elsewhere (p.213)*, the risk of getting in trouble increases. The risk of running into problems with discontinuation or other major changes should be balanced with the gains of introducing a framework.

### 10.1.2 Big Wig/The Law - Market and Imposed

The rules or regulations have changed and there is not much we can do about it. We either comply otherwise our product is unusable, or in the worst case, illegal.

Things keep changing around us and we had better get used to it. New laws are created which could mean that an application that was really fit for purpose yesterday has to change a lot in order to be able to comply. It could also be that the market is changing, or customers are

changing their behavior. We can try to change the course of events, but unless we work in a very large organization, we will 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 place, i.e 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 nor predict. Lets see how this change is imposed on companies. Consider a simple thing like a registration form for instance. A form that once was fit for purpose a couple of years ago with fields where you entered your phone number might now need attention because the phone number validation has changed or the number that once was optional is now considered the most important one.

More examples of debt builders that relate to this category are *New regulations (p.208)* and *Changes in demand (p.209)*.

### 10.1.3 The Boardroom - Market and Incurred

Software developers often experience the effects of quick decisions and swift changes from management in order to keep up with competitors. The challenge is to keep up with the changes, to embrace the change in a good way.

There are some business decisions that can have a bigger impact than initially expected such as corporate policies for instance. These decisions, or rather lack of options often steer us towards early limitations in projects sometimes even before the projects have started. *'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 simply 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 (p.205)*, *Technical policies (p.217)* and *New requirements (p.208)*.

Consider a loan institution, like a bank or something similar. They're humming along nicely and one day the owners of the bank stumble across an opportunity. They receive an offer to buy their competitors customers at a very reasonable price. The deal is quickly closed, and soon enough the only thing left is to merge the customer base into the existing back-office system.

A larger portion of the new customers have similar contracts as the ones already in place but there is a small portion of them that are different. Some have a 'loyalty bonus' in the form of an individual, significantly lower, interest rate. There are also customers that aren't amortizing on their loans during their first 6 months, they're just paying interest. None of these two contract types are supported.

Deciding to expand 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 once fit for purpose isn't as fit any more.

This fine line that separates over engineering and just enough design, needs to be re-asessing and investigated contigously, i.e very close to the actual business decision.

Even though turning down or following through on a deal like the one described above is a business decision, the impact of it can, however, be more readily handled if the technical side is involved early on.

### 10.1.4 Propeller Hat - Technical and Incurred

After all, this is where most of the technical debt we have seen comes from. It can be anything from sloppy coding, to don't-touch-it-it-works mentality, to ambitious and well-intended over-engineering they all create technical debt. Choosing a graph engine or an MVC-framework too early is a bad idea, likewise is staying too long with a home-grown one equally as poor.

What at one time looked like a good decision is now one of the biggest hurdles when it comes to effective software development. We are forced to create workarounds that add complexity to our solution without giving anything in return.

The most sensible technical changes often start with a hunch and maybe a feeling of missing a point. The code feels awkward, inflexible and stale like yesterday's bread. Sooner or later that feeling is followed by an 'idiomatic shift', that is, a new way of looking at things.

These shifts can be anything from using recursion instead of for-loops to accessing persisted data in a different way. Sadly these shifts are rarely implemented in their entirety, leaving the codebase feeling semi-disrupt. We've experienced and seen our fair share of framework-shifts. One 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 it's caretaker.

We've also seen different [versions of] programming languages come and go, different ways of reading and writing XML and more. The details of these implementations rarely interest non-technical people which makes them harder to have constructive conversations around. While sound idiomatic shifts stand the best chance on leading to improvement, the change itself can, however, add to the debt unless everone is aboard and the shift is fully implmented.

Watch out for these examples of debt builders and red herrings that relate to this category of debt: *Waiting for the right abstraction (p.205)*, *Insufficient knowledge (p.211)*, *Premature optimization (p.212)*, *Not invented here (p.213)*, *Proudly found elsewhere (p.213)*, *One-way code (p.214)*, *"Clever" code (p.216)* and *Breaking windows (p.219)*.

### 10.1.5 Borderliners

Some of the of the sources of technical debt cannot easily be put in a single category, they are rather spanning several of these quadrants, or their origin differs from time to time. Often, this is due to the fact that technical debt is often created when internal or external borders of communication are crossed.

## 10.2 ATTACKING THE TECHNICAL DEBT SOURCE

Poor internal structures constitue a multitude of problems which might all require to be handled sooner or later. If we look closer we find that there are no less than four different ways to deal with a problem. We can either:

- Absolve
- Resolve
- Solve
- Dissolve

*Absolving* is choosing to do nothing, just forgive the people or the process.

*Resolving* is about taking past knowledge from a similar situation and applying it to the current situation. This is the dominating way of dealing problems and is what we are taught and tested for in school.

*Solving* a problem is when we sit down and analyze the problem and try to find a solution that is better that the current one.

*Dissolving* a problem means we eliminate what cause the problem.

In order to show how a problem can approached in several different ways we would like to illustrate with a story, once told by Russel L. Ackoff in *Idealized Design* [34]. This story takes place in a large European city that uses underground trains and double-decker buses for public transportation. Throughout the story we'll see examples of absolving, resolving, solving and finally dissolving the problem.

> In London, double decker buses have a driver and a conductor. The conductors collect fares and issue receipts from the back as passengers board the bus. When all have boarded, the conductor signals to the driver through the front mirror. This tells the driver that everyone is on board and they are ready to go.

To make things a bit more complicated, there's also an incentive program in place, one for the driver and one for the conductor. The driver is paid more the closer he stays on schedule. While the conductor is paid more the less fares he misses to collect, which is controlled by undercover inspectors. We quickly realize that 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 collect 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 choosed 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. Actually the problem got worse so they had to do something.

They decided to change the incentive system, which they thought was causing the problem in the first place. 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.

So, now the 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 pay checks. The company wouldn't have to pay more and it would encourage the conductors and drivers to work together. If they would have 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, 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, or actually he stepped back took a broader

view of the system, which is really important when we 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 where bus stops. Conductors should get off the bus and stay at the bus stops!

If they got off the bus to collect 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 would get on the bus a lot faster.

When the peaks where over the conductors could work on board again and the problem had been dissolved.

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

### 10.2.1 Get to the bottom of the problem

In general, we want to solve the *essential and novel* problems that are at the core of our mission statement and domain, since that is what our customers are paying us for. In general, we do that by applying a series of resolutions, previously applied and known solutions to parts of the problem we are solving. In addition, we want to dissolve any unnecessary or accidental problems.

For the parts with technical debt where we don't have to pay any interest, we choose to absolve the problem, i.e do nothing. To pay back debt, we usually apply technical solutions or resolutions to the problems. The Mikado Method is a tool to help solve or resolve your technical debt problem.

But, the *creation* of technical debt is usually the kind of problem we want to dissolve. To do that, we will likely have to change the system we are working in, how work is arranged, how change requests enter our organization, how decisions are made, and so on. A shift in attitude

is required, and we need to step back and take a good hard look at the situation and the work system.

This can be difficult when we are deeply involved in the details of the implementation. Going to the bottom and dissolving technical debt requires a slightly different approach for each and every one of the previously discussed quadrants. *Fig. 10.2 (p.232)* gives some hints of phrases that tells where the origin might be.

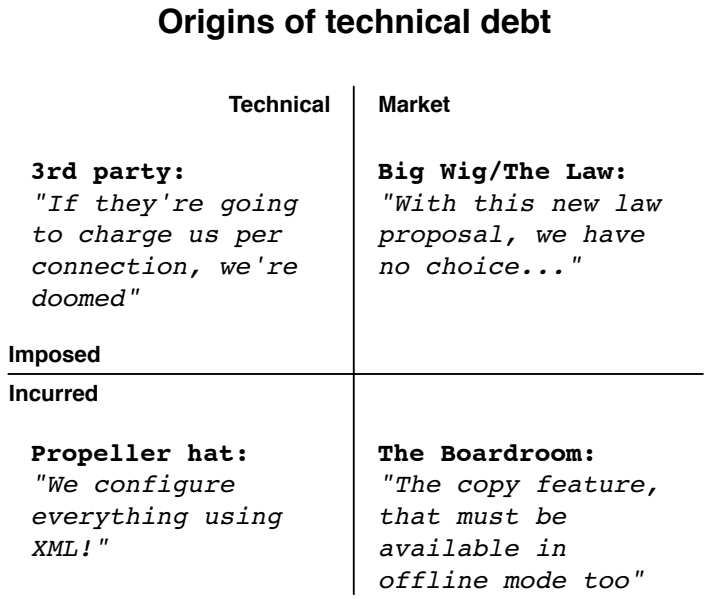


Figure 10.2: Phrases that hint where technical debt originates from

### 10.2.2 3rd party - Create defensible space

The disadvantage of being tangled up in API-calls, or tied to a specific solution to a problem, becomes painstakingly real when we are forced into change after change after change due to changes in the underlying

API. The trick when working with 3rd party software is to minimize the collateral damage of changes we can't control and identify situations which could lead to problems that spread.

If we are not careful in our use of for instance 3rd party libraries, the external dependencies can leak into our codebase, much like a wildfire spreads over a prairie or savanna. Much like a wildfire, 3rd party libraries can be hard to contain once they have started spreading. The wildfire is also fueled by the broken window principle. ORMs, 3D-engines or Report Tools can easily be become the wildfire of a codebase.

Firefighters know that stopping fires is a lot easier if the correct prevention techniques are implemented. Education, careful handling of open fires and defensible space are some of those. Defensible space acknowledges the fact that a fire needs fuel in order to live. If we remove the fuel, any fire will fade away and eventually die. Trees cut down in long corridors in a forest, or a big enough space between buildings in a suburb are examples of defensible space.

*Abstractions* or *Layers* are the defensible space of a computer program. Eric Evans calls them *Anti-corruption layers* [3]. When we are 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 3rd party API changes later on.

### 10.2.3 Big Wig - Probe and prepare

Market research and keeping in touch with customers is probing, and so is listening to news and keeping up to date with current political decisions.

In general, these are the changes that are difficult to anticipate. What we can do is to make sure our organization is able to respond quickly to prepare for *any* change. This means letting the people doing the job make the decisions on how to do it, build in slack into the process, having automated validation and verification processes in place, and try to have the organization work in a way that doesn't require expediting for anything.

When our decisions are heavily influenced or regulated by external or governmental instances we recommend having those variation points

easily configurable. We keep our options open by separating the logic that vary from external decisions from the logic that stems from internal decisions.

### 10.2.4 The Boardroom - Talk and learn

Changes to business models and great new ideas is probably one of the most natural and sane causes that can initiate change in a codebase. But when the program manager continuously makes decisions that causes a large change to our program, we have a problem. It is equally a problem when the program manager finds out that a third, seemingly identical, part will take as long time as the first two ones to implement.

If technical people 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 implementations details, and techies need to understand the relevant parts of the business domain to make good technical decisions.

To achieve this, communication between non-techies and techies cannot be expected to be built on an occasional argument. They both need to teach the other, and learn from the other, in order to build the knowledge and the respect needed to create awesome products.

The easiest thing to get started is to invite the other part to talk about both business and technical details such as technical debt, with the ambition to learn and understand the other part's work and challenges. Once we are talking more of the same language, we are more likely to make the right decisions, on both sides. This also helps mitigate the problem of creating technical policies that damage development.

### 10.2.5 'Propeller hat' - Form an opinion and align

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

The road to better code starts with forming an opinion about what great solutions look like. An opinion is nothing static, even though people tend to believe so. It is actually work in progress; it is always subject to improvement. By continuously and actively working on a teams opinion and aligning development and design ideas with that opinion, it is easier to come to agreements on where to take the existing code, what parts to improve, and how to implement new things. To form a team opinion takes some time and a lot of effort, but without talking about it we will just be running without direction, an even worse use of our time.

Looking at other peoples code, or other paradigm, for inspiration and ideas is an excellent way to broaden our perspective. We look for the qualities we want in our own work, and then figure out what parts of our current opinion to replace.

## 10.3 The way out of technical debt

Sustainable change cannot be forced, it requires a **willingness to change** and this goes beyond a software development team, it includes the whole organization. If we want to stop the behavior that led to the debt, we need to change.

Apart from a different behavior we also need **discipline and integrity** from the people who develop the software. Real improvements to a messy codebase can only be done by moving away from the current situation, and then keep moving for a long period of time. For this, the developers need **support and commitment** from management.

We also need to be **patient**. We don't expect an organization to change over the course of a week. Rewiring our brain normally takes around a year with plenty of practice, and it is easy to see that discussing and changing old habits will 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 get the product in shape at a competitive price.

Technical debt *is* a tricky thing. We can't just put up a lot of money to pay the debt and hope that the problems are gone the next day. We do

need financing of our activities, but primarily we must pay with **sweat and and hard work**.

Technical debt is **not limited to the code**, but in fact, technical debt can manifest itself in anything from code to build scripts, folder structures, version control system, tests, environments, operating systems, configuration files, and more. When we are 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.

## 10.4 Summary

Jake: It's not only the developers that cause technical debt!

Melinda: No, it also comes from business-level decisions inside and outside of the company, as well as from technical events out of your control.

Jake: Does that mean I should always be prepared?

Melinda: To a certain degree, but you can't foresee everything that can happen. Keep your code in reasonably good shape, and be prepared to take action when necessary.

Jake: Ok, I will do that. And I will try to be attentive to all the things that will get me into debt. I don't want to be reckless.

Melinda: That is good, Jake. That will take you a long way when dealing with your code.

Jake: I guess the code I come in contact with when using the Mikado Method is the one with the highest interest.

Melinda: Exactly! If you must deal with it, it is your loan shark debt. The Mikado Method helps you find it and do something about it.

Jake: It will help me outrun the goons of technical debt!

Melinda: Haha, yes, or actually wrestle them into submission!

Jake: Sounds even better! You know, I am starting to feel like I know the Mikado Method pretty well now.

Melinda: Good, I was hoping you would. Now you've got to get out there and use it!

Jake: I will do that!
Melinda: Good luck, Jake!
Jake: Thanks for everything, Melinda!

## 10.5 Try this

- What alarm bells are ringing for you in your current environment?
- What alarms are sounding most frequently?
- What alarms have you experienced in a previous project? What did you do? What should you have done?

# Appendix A

# Software Katas

## A.1 Katas in martial arts

The term *Kata* comes from Japanese martial arts and is the name of a type of exercise that aims at practicing a particular set of actions, over and over again. A Kata is usually a choreographed set of blocks, kicks, punches and more, that are used for practicing those moves. They can be practiced individually, in pairs or in groups. Katas also come in many shapes and difficulties, some are relatively simple, and some more advanced. The simple Katas are more suitable for beginners, but even advanced practitioners practice simple Katas to achieve perfection.

## A.2 Katas in software development

Software Katas are the same thing, but instead of kicks and punches, a programming problem is to be solved. Historically, software katas have been about learning test-driven development and pair-programming, but any kind of standardized exercise can be used.

## A.3 Coding dojos

The training room in (Japanese) martial arts is called the *Dojo*. The concept has been transformed to programming as a name for a place where people gather to practice programming together. This is called a Coding Dojo.

Just like in a martial arts Dojo, many forms of practice can be performed in a Coding Dojo. Katas is groups, pairs or individually are some ways of practicing, but one can also do sparring or individual training on certain techniques.

When doing a Kata in a Coding Dojo, it is common to do it *Randoori style*, meaning that one pair at a time is doing the programming in front of the whole group, preferably using a projector on a wall.

Other ways of doing it is that everyone is doing it synchronized, led by the trainer. Another is where everyone is practicing in pairs, at their own pace.

# Bibliography


[1] Michael Feathers *Working effectively with legacy code*

[2] Dave Thomas, Andy Hunt *The pragmatic programmer*

[3] Eric Evans *Domain-Driven Design*

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts *Refactoring: improving the design of existing code* ISBN 0-201-48567-2

[5] Joshua Kerievsky *Refactoring to patterns*

[6] Eric Gamma, Richard Helm, Ralph Johnson, John M. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*

[7] Robert C Martin *Clean Code*

[8] Kent Beck *Implementation patterns*

[9] Kent Beck *Test-driven development*

[10] Martin Fowler
*http://martinfowler.com/bliki/StranglerApplication.html*

[11] Larry LeRoy Constantine
*http://en.wikipedia.org/wiki/Larry_Constantine, reference International Bibliographical Dictionary of Computer Pioneers*

[12] Robert C Martin *Agile Software Development: Principles, Patterns and Practices.* Pearson Education. ISBN 0-13-597444-5

[13] Arthur J. Riel *Object-Oriented Design Heuristics*

[14] Ward Cunningham *The WyCash Portfolio Management System (http://c2.com/doc/oopsla92.html)*

[15] Martin Fowler *http://martinfowler.com/bliki/TechnicalDebtQuadrant.html*

[16] Kent Beck *http://c2.com/xp/CodeSmell.html*

[17] http://www.catb.org/ esr/jargon/html/Y/yak-shaving.html

[18] http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

[19] http://en.wikipedia.org/wiki/Code_coverage

[20] Mike Cohn *Agile estimating and planning*

[21] Dr. Eliyahu M. Goldratt *The Goal*

[22] Gordon E. Moore *http://en.wikipedia.org/wiki/Moore%27s_Law*

[23] *http://en.wikipedia.org/wiki/Side_effect_%28computer_science%29*

[24] Johanna Rothman, Esther Derby *Behind Closed Doors*

[25] Kent Beck *Extreme Programming Explained 1st and 2nd editions*

[26] Staffan Nöteberg *The Pomodoro Technique Illustrated*

[27] James Q. Wilson and George L. Kelling *Broken windows*

[28] Mary and Tom Poppendieck *Lean Software Development - An Agile Toolkit*

[29] Joel Spolskys blog *http://www.joelonsoftware.com/articles/fog0000000069.html*

[30] Ken Pugh *Prefactoring - Extreme Abstraction, Extreme Separation, Extreme Readability*

[31] Christopher Alexander et al *A Pattern Language - Towns Buildings Construction*

[32] Leo Brodie *Thinking Forth - A Language and Philosophy for Solving Problems* ISBN 0-9764587-0-5

[33] Linda Rising and Mary Lynn Manns *Fearless Change - Patterns for Introducing New Ideas* ISBN 0-201-74157-1

[34] Russell L. Ackoff *Idealized Design: How to Dissolve Tomorrow's Crisis...Today* ISBN 0-13-196363-5

[35] Brian Foote and Joseph Yoder *Big Ball of Mud Fourth* Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97) Monticello, Illinois, September 1997

[36] Jez Humble and David Farley *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* ISBN 0-321-60191-2

[37] Gojko Adzic *Specification by Example: How Successful Teams Deliver the Right Software* ISBN 978-1617290084