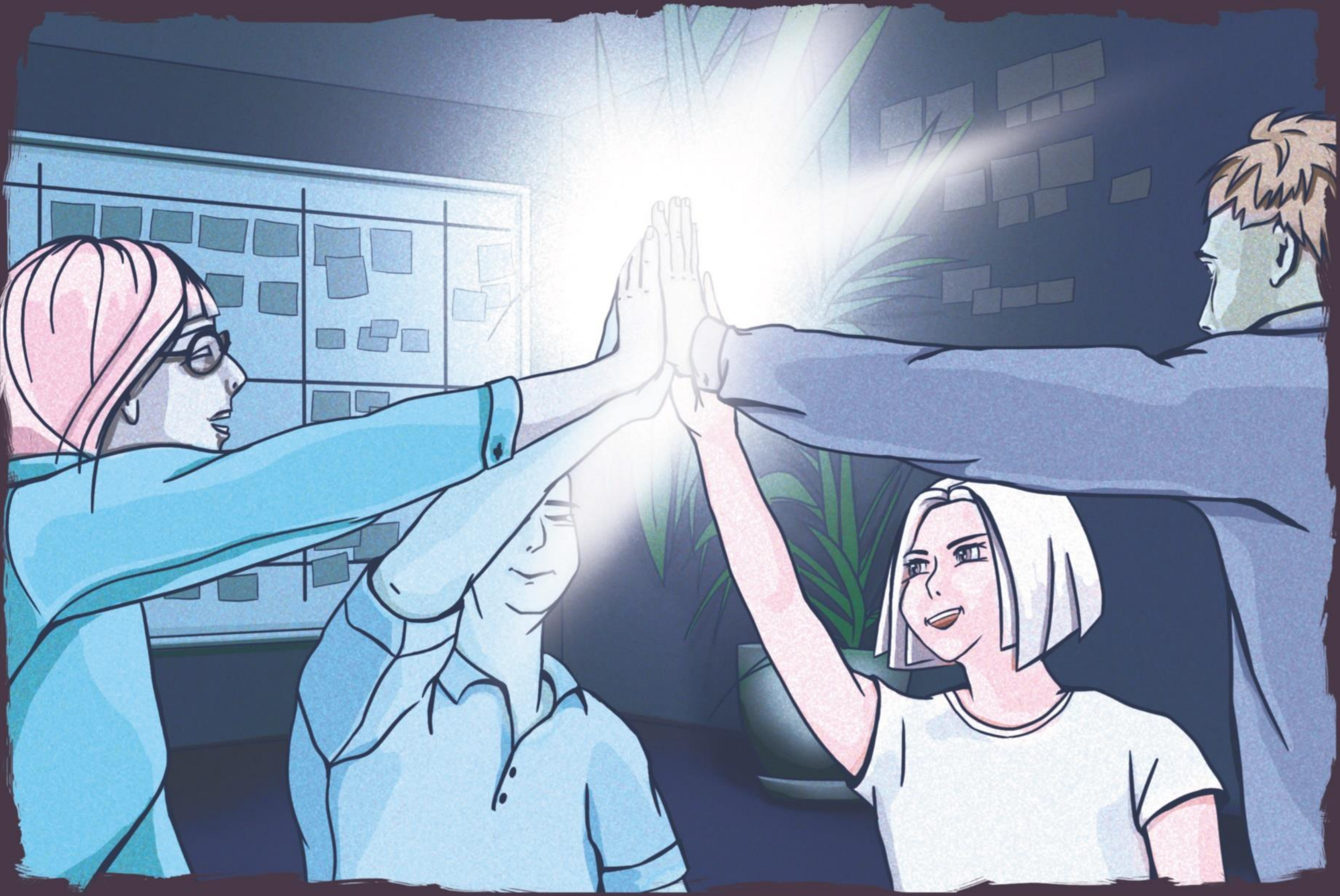


Quick Glance At: Agile Engineering Practices

Part 1: Writing Software

David Tanzer

Books are better read on paper. From time to time, this book will ask you to write something down directly in the book. And even where it does not ask you to do so, you will remember the content better if you write down your ideas, questions and remarks in the margins.

So, if you want to write directly into the book, or if you want to support me financially for writing this book, **get a printed copy here**:

Amazon US

Amazon UK

Amazon DE

Amazon FR

Amazon ES

Amazon IT

Amazon JP

Amazon CA

If you like the book, please **recommend it to your friends**, followers and colleagues.

But please do not send them the PDF. The e-book version is free for now, so anyone can download it from the book’s page in exchange for an email address:

https://www.quickglance.at/agile_developers_practices.html

You can use the following links to directly recommend the book:

- Recommend the book with a Tweet
- Recommend the book on Facebook
- Recommend the book on LinkedIn

Thanks to my family, who have always supported me on my journey. I know it wasn’t always easy for you.

Thanks to all the people who have taught me, trained me and helped me learn these practices and skills. I learned from you on Twitter and via our online and “real-world” conversations.

You showed me things I didn’t know, helped me learn them and inspired me to go beyond my comfort zone.

I want to especially thank those of you who allowed me to use your quotes in this book. You are:

Cassandra H. Leung, Emily Bache, Esko Kilpi, GeePaw Hill, Jason Yip, J.B. Rainsberger, Ron Jeffries, Samir Talwar and Stephanie Hurlburt.

Thanks to Irina Linuza, who drew the cover illustration and the illustration in the Communication section (the other illustrations were drawn by me).

Finally, a big thank you to my editor, Susan Ryan!

How do you get better at writing software? How will it help your team when you focus on quality code? What does it take to develop software as an agile team?

I think about this a lot... But what do you think? What are your questions?

And what does that mean ?!

Worried that TDD will slow down your programmers? Don't. They probably need slowing down.

J.B. Rainsberger – www.jbrains.ca

Do you write notes in the margins of your books?

Why This Book?

I teach "test-driven development" and "agile developer's skills" to different groups of people.

I have done so for a few years now, and I teach these sessions in a completely hands-on way. Most of the time, the attendees actively participate in some form, either through drawing, programming, discussions or research. I only show a few slides throughout. Most attendees love it!

But sometimes, an attendee will complain about the training materials. They will want more slides, or they will ask me where they can learn more about the "theoretical" aspects of the skills I teach. I guess they want more text (even more than the notes they have made themselves) to re-read later.

I do have slides, and I create a lot more of them on the fly. During these training sessions, we create many flip-chart pages together, all of them full of information. But for some, that's not enough.

I always jokingly tell the attendees: "Please take notes. You will get all the slides and pictures of everything we created together, but these will not contain everything **you** want to remember. I could write a book and there would probably be things missing that you would like to keep in mind."

Well, now I have decided to **write that book**. And I have intentionally incorporated lots of white space—both in the margins and in between chapters—so you can add your notes and remarks. Because no book can tell you everything **you** want to remember, especially the thoughts, feelings and ideas you had while reading it.

I have tried to write this book in such a way that it can be used as further reading material for my training sessions, but also for independent learners to use on their own as part of their education (by purchasing it on Amazon, for example).

Please let me know your thoughts on the book or contact me if you have any questions. You can get in touch either by email: business@davidtanzer.net or on Twitter: @dtanzer.

In the tradition of the other *Quick Glance At* books, this book will be short, to the point, and it will give you various **ideas for further research** wherever it does not contain the answers itself.

David Tanzer, December 2019
https://www.devteams.at/

This book tries to give you an introduction to the many different things you **will learn**—or have **already learned**—during your career.

Some topics might seem trivial to you because you have already learned them. That does not mean that they are trivial to everyone! Skip over them or read them to see if I have a different perspective.

I hope that some of these topics are new to you or that you can at least learn some new aspects of already familiar topics.

Sometimes, this book will ask you questions. Write your answer down. You will learn more when you write things down **by hand** than when you only read about them.

Use a pen/pencil to write the answer down on paper. Handwriting is the most effective approach here.

Table of Contents

Skills

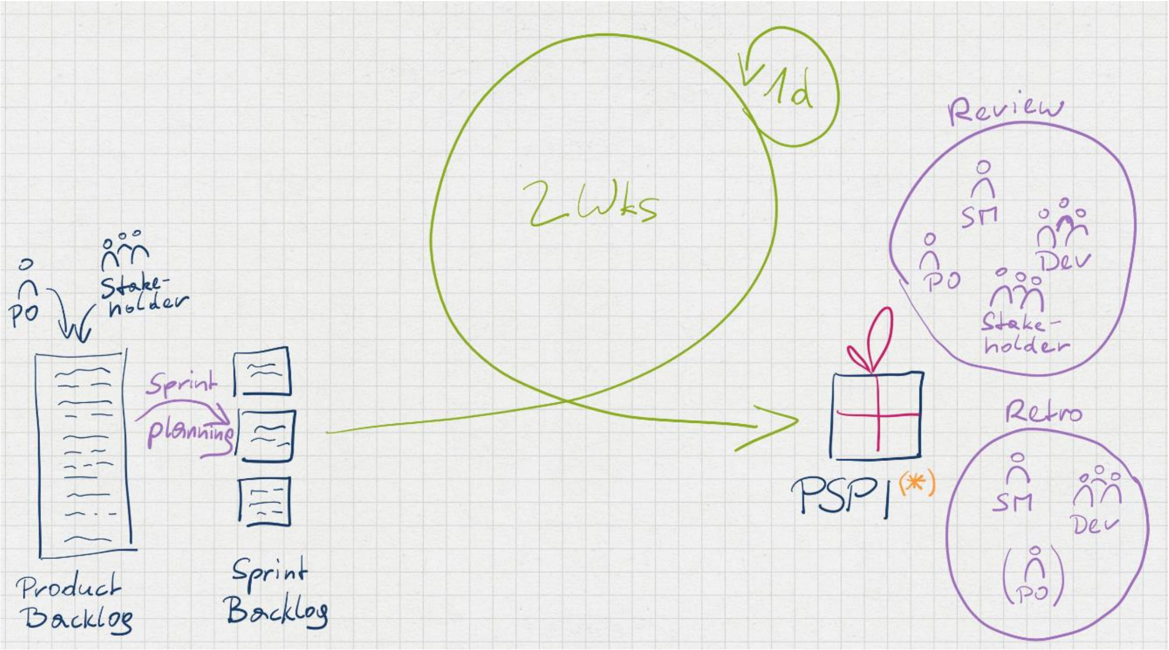
‘‘There are many skills which a developer needs to have. There is one unique skill which an agile developer needs to have, not taught in any other part of the profession. That skill is delivering ‘working software,’ repeatedly and sustainably, every week (or even day).’’

Ron Jeffries – ronjeffries.com

Deliver Working Software

Deliver. Working. Software. This is the skill that we as agile software developers must learn! And it is easier said than done.

Look at the picture below about one of the agile methodologies. What do you see?



That little package (the thing marked with a (*)), that potentially shippable product increment or PSPI, is supposed to be working software that is finished and packaged.

But did we deliver working software when we created “potentially shippable” software?

“Potentially shippable” is not enough! Only when the software is finished, tested, shipped to real users and is running in production can we say we are done with the current task. Only when the software is running in production can we say we delivered it.

This is the skill we must learn as agile developers. We must strive to deliver real software to real users on a weekly, daily or even more frequent basis.

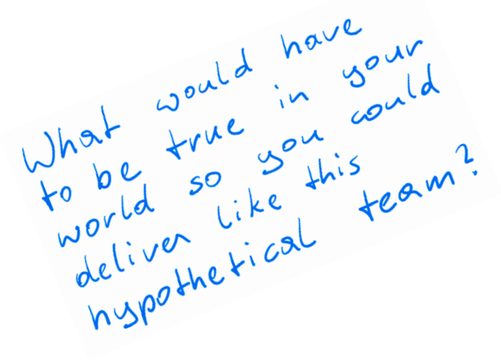
Suppose there is a team that, when we need a small change from them, can implement the change and **reliably** deploy it to production within half a day. And suppose they do that for **every** change, so they have multiple production deployments per day.

What must be true for this team so that they can do that?

When they already have small, independent work packages (like user stories)—and creating those is a difficult-to-learn skill on its own—the team must be able to:

- *Quickly* find all the places in the code where a change is required
- Agree on the changes that will be made
- Test for technical regressions *quickly*
- Check whether this feature does what the user/PO is expecting *quickly*
- Confirm if all other features still do what the user/PO is expecting *quickly*
- Deploy to all environments *quickly*

The skill to deliver working software every week (or more often) requires us to get increasingly better at the above tasks, and possibly at other tasks, too. So, how do you learn that skill?



What would have
to be true in your
world so you could
deliver like this
hypothetical team?

Skills vs. Practices

How do you learn a skill? How can I teach a skill?

You can learn a skill by practicing. But I don’t believe that any teacher or any book can teach you a skill. We can only teach you practices that develop a skill and support you when you practice. So, what should you practice in order to learn the skill of delivering working software?

First, you should practice delivering working software more often, even when it hurts! In agile software development, the phrase “if it hurts, do it more often” (commonly used since early 2000s) is commonly used. And while you’re doing it more often, fix all the problems that make it hurt.

But that is not what this book is about (maybe a later part of the series). This book is about practices that will help you deliver working code more frequently. This book tries to teach you the effective and efficient strategies that other people or teams use. I will show you:

- How to improve **software development** itself.
- How **your tools** and your skills with those tools matter.
- How **test-driven development** and **refactoring** can help you become more effective.
- How to use **testing** and a testing strategy.
- And more…

You need more to deliver frequently, so these might be topics for later books:

- How **software design** can make or break your ability to ship faster.
- Why you also need to think about **software architecture** constantly!
- What you can do to **work more effectively together** as a team.
- What else it takes to **ship software**.

Yes, this is a lot. And most of these topics require a lot of practice. Test-driven development alone, for example, can set you out on a life-long journey of learning and improving. Not every practice will work for you or for your current situation. But these are the things that work well for other people. So, learn and understand them anyway. Understand why they don’t work for you, and **only then** ask yourself, “Should I drop the practice or change my situation so that it will work for me?”

Book: “Continuous Delivery” by Jez Humble and David Farley

Become the Developer You Want to Be

Think about all the developers you know. Some care more about our craft than others, right?

“Just Add Another If”

Many years ago, I was investigating a tricky defect. The GUI of our application crashed when the user performed a certain task.

The original developers used observable values and callbacks that performed some work when a value changed throughout the code, which made debugging the problem very tricky. “Wait, clicking this button only sets a value… Oh, there’s a callback when this value changes! And it sets another value, triggering another callback!”

After two days of debugging, I had a huge graph of everything that happened when the user tried to finish their task. There was a cycle of almost 100 callback calls, where the last call of the cycle changed the original value again. Another cycle would start, ultimately resulting in a stack overflow error.

I showed it to the team and presented a solution that would require removing observable values from parts of the application. It would clean up the code but require a lot of work. The original developer said:

“What are you doing here? Just add another ‘`if`’ here to break the cycle!”

In this case, I was overruled. But, I decided that I did not want to be *that* developer who just adds another ‘`if`’, making the code worse. At least I try not to be. I’m sure I did the same thing later—a quick fix that made the code worse—because I make mistakes, too. But I work on not doing it, and on leaving the code cleaner than I found it.

When you work with your team on a project, you are creating a legacy. How do you want your team to talk about your code? How do you want the next developer—which might be your future self—to feel about working with the code that you wrote?

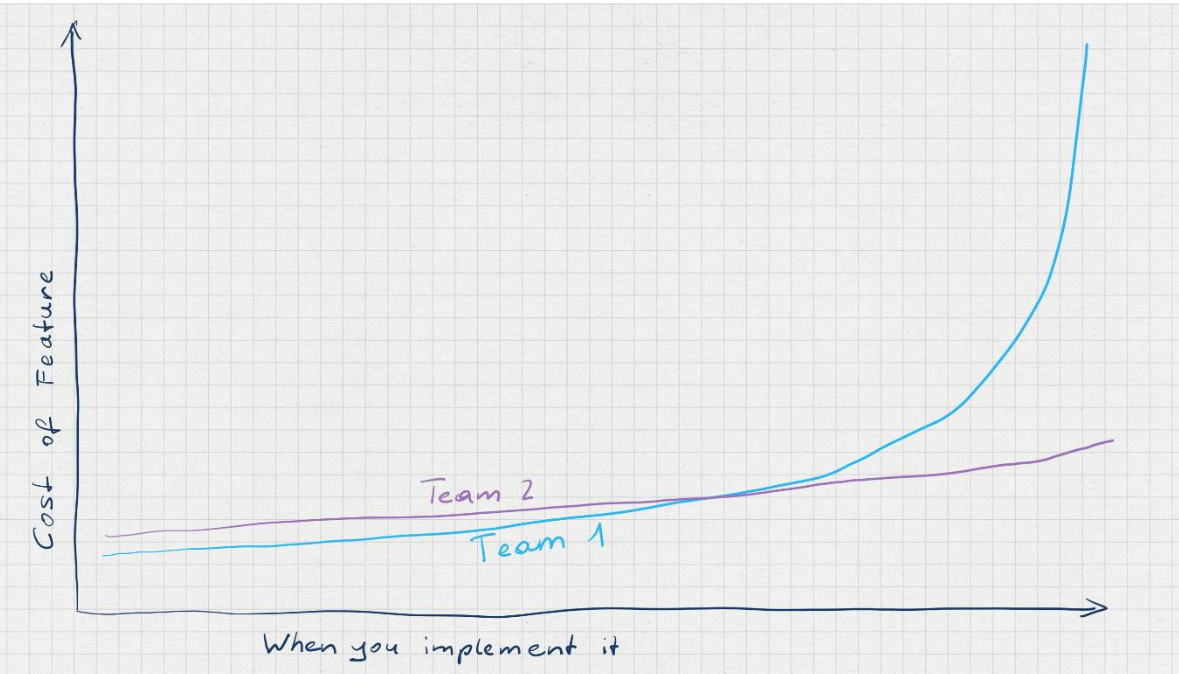
And how do you want your teammates to feel about you, working with you and communicating with you? Do you want to be the careless one? The grumpy one? The person who writes clever code that nobody understands? Or are you the person who helps everyone? The teacher and mentor, the person who cares about the little details and the overall architecture at the same time?

Sustainable Code

How long does it take to implement a simple feature? Like a "share this page on social media" button?

How much does this value change with time? For example, the change would have been trivial (just a few hours or so) when we first started with this project. But now, five years in, we have so many different page types and special cases that it will take us a few days to make that same change.

The answer to this question will change over time. Features will get more expensive when you implement them in a huge codebase.



I once encountered a team that had been working on the same software for several years. Every six months, they delivered a release to their customers.

They did not really take care of their design. They were just shipping features for years and years.

Sure, the last few releases contained fewer features than the first few, but they thought that everything was fine.

Then, there was a Fall release in year X that was not ready for their customers. They had to skip it. And in Spring X+1, they still had nothing to ship. And again, in Fall X+1. That next release took over a year and a half to ship, but the trouble did not end there.

They (and some other real teams I've worked with) were like Team 1 in the graph, so for them, the hockey-stick part of the curve had happened extremely fast. The code was already bad, but when changes were made to the team, they were suddenly not able to ship software **at all** anymore.

Well-crafted code will ship faster.

That's what the hypothetical Team 2 realized. Team 2 took care to do things right. They took smaller steps, maximized the work not done and cleaned up all the time as they went. All of these things enabled them to keep progressing at a good pace for a long time.

The cost of a single feature was a little bit higher than for Team 1 for most of the graph. But:

1. They are minimizing the risk of total failure.
2. They might still be faster if they can waste less time on other things (like developing a feature for too long without feedback, just to find out that nobody needs it).

In agile, we do not want to optimize the cost of a single feature. We want to optimize the ROI of everything we do over a long period.

Try to be more like Team 2. If your team is more like Team 1 or somewhere in the middle, find ways to help them become Team 2.

You can find a complete list of books and linked websites at the end of the book, starting on page 128; e.g., to find the author and title of the linked essay below ("[Blogs 1]"), go to page 128 and find the list entry 1.

Book: *The Nature of Software Development* by Ron Jeffries
[Blogs 1] https://www.devteams.at/2015/12/14/well-crafted-code-quality-speed-and-budget.html

Sustainable Pace

Writing your code in such a way that **you** can quickly work on it and maintain it in the future is not enough. What about the team? The people who write the code, who test it, run it and the people who manage development?

As an agile team, you must strive to create an environment, a culture and a way of working together that allows you to **keep your pace indefinitely**.

When done correctly, nobody will be over-worked because everyone takes care of themselves and each other. And because nobody is allowed to push work onto the team, it is the **team who decides** how much can be done in each day, iteration, quarter, etc.

People care not only for themselves and each other but also for their product. They try to deliver the best possible outcome within the shortest possible time. Therefore, they make continuous improvements. They work hard on getting better at delivering a product quickly.

You must strive to move faster as an agile team, but not by working harder. You go faster by working smarter. This is accomplished by maximizing the work not done, reducing risks and running small experiments with fast feedback.

Working harder is not sustainable. When people work longer hours for more than a few days, they inevitably get slower and slower and quickly lose all of the speed benefits of working longer. They will get frustrated and overworked. The company will lose money because their people will have more sick leave, and some will quit.

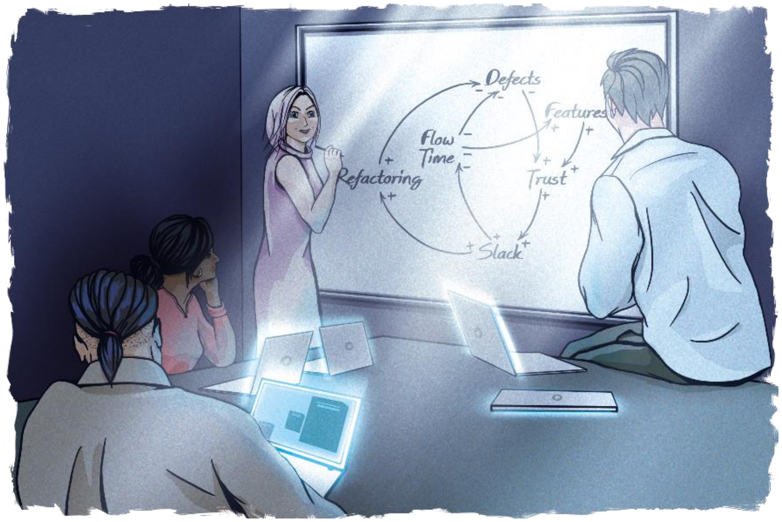
Trying to squeeze more into an iteration, which is just another way of working harder, is also not sustainable. Teams will cut corners, and internal and external quality will suffer. Because of the external quality, your customers will start to complain. Because of the internal quality, you will end up in a situation like Team 1 from the previous chapter as changes will become harder to implement over time.

Work smarter. Try to eliminate unnecessary work, processes and waiting time. Run small experiments all the time and get valuable feedback as early as possible.

Communication

Have you ever heard the myth that software developers do not like to talk to other people and work alone in dark caves? That does not work anymore. It probably never has.

Agile teams *communicate* a lot. I mean, **a lot**.



Developers in an agile team communicate with each other, with testers, other specialists, product owners, scrum masters, managers, customers, users and other people.

They communicate face to face using electronic tools like video conferencing, email, source control or chat and using offline tools like flip charts, whiteboards, scrum boards and other information radiators.

If you want to become a great software developer, you should be good at communicating—*especially* with the supposedly "non-technical" people in your workplace.

Know how to communicate differently with different groups of people. Learn how to present your work, the problems you are facing and your proposed solutions. Practice explaining stuff quickly, staying on point and delivering your ideas at the appropriate level of abstraction.

Staying Relevant

The world of software development is changing at an incredible pace. How can you ever keep up with it? Do you even need to keep up with *everything*? Can you remain a developer or tester or software architect throughout your career, or is your only chance to stay relevant to move into management at some point?

What do you need to learn today to stay relevant as a software developer?

Agility, and especially business agility, are here to stay. In a changing world, companies must become agile to remain relevant. So, to stay relevant as a developer, you must learn to work well on an agile team. You must:

- Learn to communicate with developers and other stakeholders. Learn to share your knowledge, to be a mentor, to help junior team members become senior.
- Learn to learn and adapt. Learn to understand new code, technological developments and new business concepts quickly. Learn to share your knowledge.
- Learn to create sustainable code. Learn to write code that is easy to understand, easy to extend and easy to maintain. Learn to automate tests for your code on different levels—tests that provide additional value and are easy to understand and easy to maintain.

Those three points are sorted in order of priority. Be nice and communicate well is the most important one. Sustainable code and quality come last, even though they are still important.

It is also crucial that you learn to learn and adapt. Try new things. Embrace new concepts that are counterintuitive for you. Ask yourself, “what would it take for this to work?” Talk to your peers, attend meetups and speak at conferences.

Use tools and strategies that will help you learn faster, such as code katas or an engineering notebook.

The only constant is change. To stay relevant, learn to learn and adapt. If you can learn and adapt when you need to, you do not need to keep up with *everything* all the time.

Book: *The Passionate Programmer* by Chad Fowler

Slow Down to Move Faster

Doing things *right* will be slower at first, but it will be faster in the long run.

Some teams spend a lot of time on re-work. That is, they are working *again* on something that they thought was *finished*. Perhaps they misunderstood the user's requirement, are fixing a defect, or are improving the performance of the system because they found a bottleneck.

When I say re-work, I am not talking about building a simple feature, showing it to users, gathering feedback, making it better, polishing, scaling and optimizing. I would expect this kind of iteration from every agile team. I am also not talking about a situation where a feature was done, people used it in production, but the world changed, so the feature must be changed. Agile teams must be prepared for that.

When I say re-work, I mean that the team **thought they were done** but then had to re-open the feature again because there was a regression or because they implemented the wrong thing.

I have seen way more than 50% re-work with some past clients, and I have read numbers of 70%-80% from other teams that I do not personally know. But those teams often do not classify the work as re-work. They call it "change requests", "new user stories" or "regressions". But that is just hiding the fact that they are working *again* on something that was previously deemed finished.

Doing things *right* will be slower at first.

I am not talking about gold-plating or over-engineering. I am talking about implementing the smallest possible part of a feature as quickly as possible, implementing the simplest technical solution that could possibly work, refactoring before starting to implement to make the next change easier, automating everything, creating small and simple updates that are still crafted and tested well.

Iterate to get feedback but do things right internally. Design, test, craft, refactor and automate.

When you pair-program, do TDD, write business-facing tests and scenarios, document early. When you create something small, deliver it, gather feedback, make it slightly bigger, deliver it again, gather feedback again. You will be slower than when you plan the whole feature, implement it, and deliver it as part of a large batch.

When you do things right, every single feature will take longer than in the old, plan-driven world.

> *From an efficacy perspective, this increase in development time is offset by the reduced maintenance costs due to the improvement in quality (Erdogmus and Williams, 2003), an observation that was backed up by the product teams at Microsoft and IBM.*
>
> Nachiappan Nagappan et al., *Realizing quality improvement through TDD*

However, if you take this approach, you will not implement every feature *fully* and then, after delivery, realize that your users wanted something different. You will know when to stop because you have gathered feedback early. And you can change more easily when you realize that you have misunderstood your user's requirements.

If you can stop working on features before they are fully implemented—because they are good enough—you will save time. When you do things right—when you keep your software architecture, your software design and your code clean—you can avoid that huge cost further down the line. So, even if you are slower during development, you will be **faster overall**.

Bad code and bad software architecture can slow you down considerably. Say you want to implement a small, new feature. If your codebase is already rotten, it will take you a long time to find all the parts that need to be changed. And even then, each change will take a long time, and it will be riskier, just like in the hockey-stick curve example in the Sustainable Code section.

And remember when I told you that some teams spend a lot of time working on things that they thought were finished? That is time that you can now claim back. The need for re-work can be drastically reduced if you've done things right from the outset. If you get that number down from 75% to 50%, you have twice as much time for implementing new features. So, remember, even if you are slower in development, you will be **faster overall**.

> *"The general principle of software quality is: Improving quality reduces development cost. You don't have to choose between quality, cost and time—they all go hand in hand."*
>
> Steve McConnell, *Code Complete 2*

Writing Software

“Technical Agile Coaching usefully describes the work I do. Agile Coaching helps your business to become more successful, often by improving the way you plan and deliver software. Technical Agile Coaching focuses on how people write code.

My aim is the same as other agile coaches: to improve the agility of the organizations I work with. I use coaching, teaching and facilitation techniques just as other coaches do. My focus is on the way the code is written.”

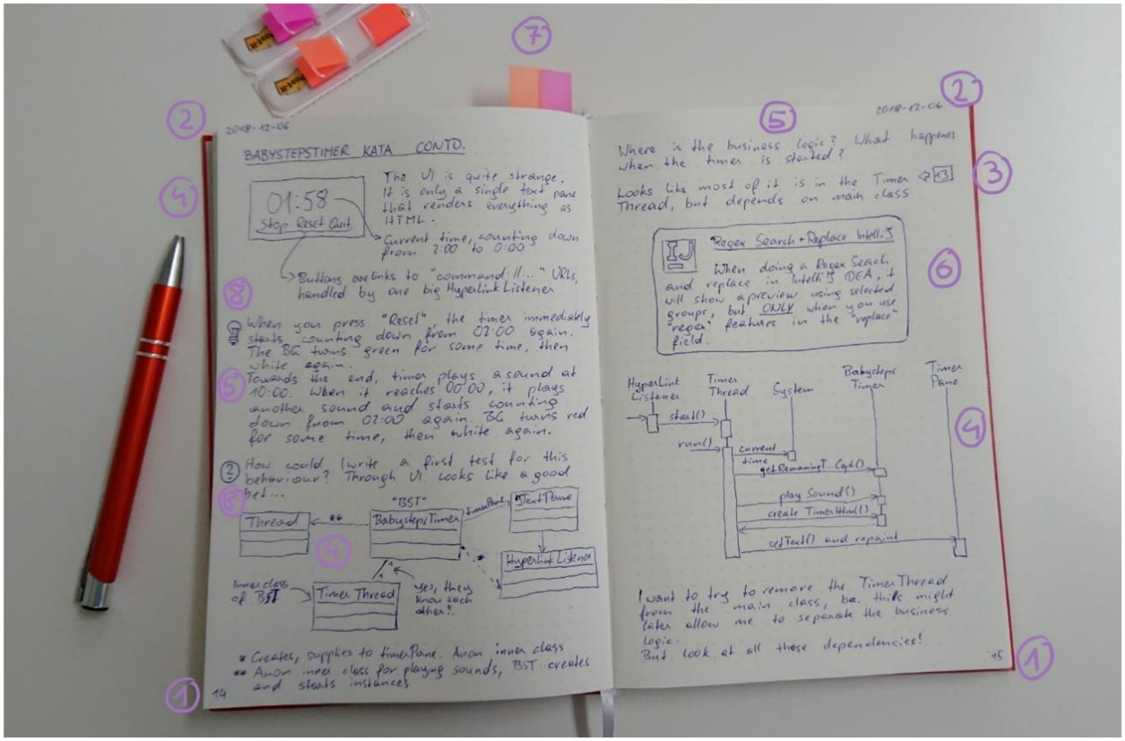
Emily Bache – @emilybache

Engineering Notebook

If you want to try just one practice from this book, then start an engineering notebook. In a notebook, write down:

- What you are working on and your decisions
- Everything you have learned
- What you have accomplished
- Ideas for improvement
- TODOs
- A glossary

If possible, handwrite these in a paper notebook. Here's an example of how I keep such a notebook:



2018-12-06

BABYSTEPSTIMER KATA CONTD.

The UI is quite strange. It is only a single text pane that renders everything as HTML.

Current time, counting down from 2:00 to 0:00

Buttons are links to "command://..." URLs, handled by one big HyperLink Listener

When you press "Reset", the timer immediately starts counting down from 02:00 again. The BG turns green for some time, then white again.
Towards the end, timer plays a sound at 10:00. When it reaches 00:00, it plays another sound and starts counting down from 02:00 again. BG turns red for some time, then white again.

How could I write a first test for this behaviour? Through UI looks like a good bet...

* Creates, supplies to timerPane. Anon inner class
** Anon inner class for playing sounds, BST creates and starts instances

14

2018-12-06

Where is the business logic? What happenes when the timer is started?

Looks like most of it is in the Timer Thread, but depends on main class

Regex Search + Replace IntelliJ

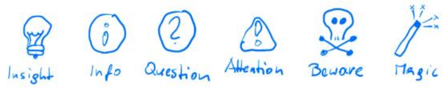
When doing a Regex Search and replace in IntelliJ IDEA, it will show a preview using selected groups, but ONLY when you use "regex" features in the "replace" field.

I want to try to remove the TimerThread from the main class, bc. this might later allow me to separate the business logic.
But look at all these dependencies!

15

The elements you can see on these pages are:

1. I number the pages. This makes it easier to refer to previous or next pages.
2. I write down the date I started that page at the top. Sometimes I write multiple pages in one day, other times it takes me a week or longer to fill a page.
3. I use arrows to refer to other pages.
4. I draw GUI mockups, UML-like diagrams or just boxes and lines, whatever helps me think.
5. I write down questions, things that come to my mind and things I have learned.
6. I write down things I have learned, even if they have nothing to do with the current topic.
7. I use sticky notes to mark pages I might want to come back to for future reference.
8. I also sometimes draw symbols in the margins. These are symbols that I can draw quickly to denote different types of text or information, such as:



Writing this notebook helps me to get into new projects and unknown codebases faster. It also helps me to learn more quickly and to remember things better, even if I don't look them up again later.

Interestingly, most of the time, the things I do look up later are things I deemed unimportant or side notes when I wrote them down. So, I try to err on the side of writing down too much. You never know in the moment what will be interesting or helpful later.

But what if you have terrible handwriting? You can still write this notebook by hand as long as you can read it yourself. It's just for you.

And you can practice writing "nicer": When I was 20, I did not like how I wrote some letters. So, I learned and practiced writing them differently.

[Blogs 3] https://bowperson.com/wp-content/uploads/2014/11/SixTrumpsArticle220101.pdf; "Writing Trumps Reading"

Learning What/How to Code

Whether you are starting a greenfield project or joining an existing team that has been working on a large system for years, your biggest challenges will all be about learning. You must learn:

- What the different stakeholders (i.e., users, customers, operations, legal, etc.) need from your software.
- How to build features that address those needs.
- How the current architecture/design works and where to find, change and add code.
- How to change the current architecture and design to fit new requirements, and how to do so *safely.*
- And much more!

Once you have learned all those things—once you know exactly what to build, how to do it and where to make the changes—programming is easy. The hard part of programming is not writing the code. The hard part is learning what code to write, learning to write and change software safely and learning to get better at all of those things.

"How can we learn faster?" That is a question that fellow trainer Peter Gfader often puts at the center of his training sessions.

Learning faster is one of the reasons why I want you to keep an engineering notebook. Write down the things you have learned about the code, the design and the architecture. Write down what users and other stakeholders have told you.

Write down the decisions you made as a team and the decisions that were imposed on you by the higher-ups. Write down whether you agree with those decisions or not, and why. This is not so you can say, "I told you so!" later, but to help you remember why you made specific decisions so you can get better at making good decisions.

You will remember things better when you write them down by hand. And should you not remember them, you can easily revisit your notes to refresh your memory.

That's the reason why
I usually buy paper books
and write in the margins

Keeping a notebook is one way for you to accelerate your learning. But think of other ways you, personally, can learn faster. Brainstorm with your team to come up with ideas for how the whole team can learn faster.

Naming Things

Think of all the times you name things while developing software. You name variables, functions, classes, files, directories, modules, commits to your version control system, deliverables, versions, bug tickets, etc.

You name things all the time. But are you good at naming?

Naming things is hard. When we want to do it right, when we want to find great names, it's one of the hardest things to do.

And naming things is **important**. A good name can make the code more readable and maintainable. A good name can explain why the code is like that to the next person. A bad name can make it harder to write the next feature. It can be the reason for a future bug.

However, many programmers I know—far too many—name things carelessly. They do not think nearly enough about the names they create.

So, how should we name things? What makes a good name? What makes a bad name?

Think about it. Write down your answer before you proceed.

Names should **fully describe** the things they are naming. This example on the left misses the fact that there is some validation happening. We can do better by encapsulating the validation in a domain class. In the example on the right, the validation happened long before payTaxes is called:

```
public void payTaxes(String taxAccountNo,
        int amountInCents)
        throws TaxAccountNumberExc {
    validate(taxAccountNo);
    /* ... */
}
```

```
public void payTaxes(
        TaxAccountNumber taxAccountNo,
        MonetaryAmount amount) {
    /* ... */
}
```

If your name is too long, what you are trying to name is too big. See this as a refactoring opportunity.

Names should be on a slightly higher level of abstraction than what they name.

A name that describes what the variable/method/class is doing does not create a higher-level concept and does not help us to understand the software better. But when the name is too abstract, too disconnected from what it is naming, it will also create confusion.

The name you choose should help the reader understand what happens in the code. It should also help them to understand why the code is the way it is. Names should **reveal intent**. Compare these three names. What do they tell you?

```
for(int i=0; …) {
for(int row=0; …) {
for(int searchRow=0; …)
```

1. The code seems to do something with indexes.
2. The code iterates over rows, probably in a list or a table.
3. The code searches for something in row data, probably a list or a table.

Names should use **language from the domain**. If a businessperson can understand the name, you are probably on the right path. Compare these two names:

```
boolean isDefault;
boolean isPreferredShippingAddress;
```

Do not use abbreviations (except for very well-known abbreviations from your business domain). Sometimes "`i`" is appropriate for an index, but most of the time, you can come up with a better name.

Do name intermediate results. When you have multi-part calculations or Boolean expressions, extract the parts to variables or functions. Then you can give them a meaningful name:

```
boolean hasOrders = orders != null && !orders.isEmpty();
boolean booksOrdered = hasOrders && orders.stream().anyMatch(x -> x.isBook());
boolean magazinesOrdered = hasOrders && orders.stream().anyMatch(x -> x.isMagazine());

if(booksOrdered || magazinesOrdered) { … }
```

We could have written all these checks inside the "`if`" condition, but naming all the intermediate results is more readable. It reveals our intent and informs the reader of the business concept behind the expression.

In relation to that, you should even name the small things. A calculation is never too small to be extracted to a local variable. Code is never too small to be extracted to a method. But beware of adding too much indirection. When it becomes hard to find the place that *actually* does the work because of all the abstractions and delegations involved, then you went too far.

These are just a few guidelines for finding proper names. When it comes to naming, do not forget the overall goal. By choosing good names, you are making the code more readable and more maintainable. You are showing the next person not only what the code does, but also why it is there and how it connects to the business domain.

You are trying to reduce the cognitive load of a future reader by hiding some details. When someone reads the name of a class or a method call, they should **not** be required to also read the content. They should know everything they need at that point from the name.

When choosing names, you want to help future readers and future maintainers. And that person might even be your future self, so take extra care when you are naming things!

[Blogs 4] https://hackernoon.com/software-complexity-naming-6e02e7e6c8cb
Book: *Clean Code* by Robert C. Martin

Pure Functions

A pure function is a function where…

- The return value is always the same for the same arguments
- The function does not cause any side effects

The first property means that the function cannot use any local or global state to calculate its result. It cannot use any user input or persistent data. It cannot call some service over a network, not even a pure function as there are too many ways the network call might fail.

The second property means that the function is not allowed to change the outside world. It cannot mutate a local or global state. It cannot persist data to a database of a file system. It cannot even log something (if we are really strict about the term “pure”).

Why would we even want pure functions? It seems like they cannot do anything!

First, pure functions are inherently easy to test. You never need a mock object. You do not have to worry about how to set up your system under test. All you must do is prepare some inputs, call the function and compare the outputs.

Second, pure functions are safe to call. They do not change the outside world, so you can call them wherever you want and as often as you like, and you will never break unrelated code (except possibly for performance regressions).

Third, pure functions are often easier to understand when someone reads the code later. They are data in, data out. There is no magic.

So, design or refactor your code to use pure functions as much as possible. Push mutable state and side effects like I/O or persistence to the edges of your classes and even software system. Create a functional core that is easy to test, easy to debug and easy to understand.

And if you find a pure function in your code, **never** change it so that it becomes impure!

[Blogs 5] https://tommikaikkonen.github.io/impure-to-pure/

Side Effects

“Every time we fix a defect, something else breaks. Even when we simply add functionality, sometimes we introduce regressions in a completely unrelated part of the code.” Have you ever heard a sentence like this from a co-worker? I sure have.

Most of you will have encountered this very situation. If you have ever had to deal with legacy code, you will have experienced this. One of the most annoying things about legacy code is that you cannot change anything without breaking something else.

The changes we make in a system like that will often have unintended side effects.

So, when designing software, one of our goals must be to limit these unintended side effects.

We must **make changes predictable**. When you change some part of the code, you should be able to easily find out which other parts of the code will be affected by this change. Good names and good abstractions will help the reader find out what will be affected by the change. Proper technical documentation—documentation that is short, to the point and accurate—can also help.

We must **limit the impact of changes**. We must take care to design our systems in such a way that changes do not ripple through the code. We must strive to decouple the parts of our system and make the elements more cohesive.

And we must **make finding regressions easy and fast**. This means we need automated tests as our safety net. We also need different types of tests on different levels.

In the rest of this book series, you will learn more about techniques that will help you achieve those three goals.

But for now, when you face a change that ripples through the system, write down what happened. Try to find the root cause. That is, the specific design decisions in the past that now make the change difficult. If you can, revert your code and refactor the existing code to make the change easier. Then try to implement the change again.

“Clever” Is (Often) the Opposite of “Good”

“Clever” code—code that does its job with the shortest number of lines or in an over-general way, or code that adds some feature to existing code in a “magical” way—is often less readable and harder to maintain, even though people often write clever code with the intent of making it maintainable and enabling re-use.

And the clever code can cause other problems because it is harder to debug, harder to figure out and harder to review.

The Bottleneck of the App Is... `toString`?!

I was working for a client on a data migration project while most of their own developers worked on re-implementing a core part of their application. They had some performance problems, and I had some profiling tools installed that I had needed for another project.

Their custom-made application framework added a lot of convenience, such as automatic validation and dependency injection, generated `toString`, `equals` and `hashCode` functions, automatic caching of values from the database, automatic logging and metrics gathering and more. It was a very **“clever”** framework that had some features that were not available in off-the-shelf solutions back then.

I remote-profiled their server. Then, I went to one of their architects.

“Your application spends more than 40% of its time in calls to `toString`.” – “What? That’s impossible!”

It turned out that their automatic database cache would load some values only when they were first accessed, and the generated `toString` would often access values. Most of their `toString` calls made an unnecessary database call.

To me, clever code is code that does something in a non-obvious way. Like a design that tries to anticipate every possible future and that is as general and abstract as possible. Or, as in the above case, adding capabilities and functionality to existing code in a “magical” way.

Many modern application frameworks, persistence libraries and other tools work like that. So, you cannot eliminate “clever” completely. But every bit of “clever” code makes your application harder to understand. And it makes it harder to find problems, like the performance problem above.

Libraries and Frameworks

Using libraries and frameworks can simplify your work a lot. But every dependency you add also increases the overall complexity of the system you are working on. Libraries and frameworks can be assets and liabilities at the same time.

They often provide some benefit, but they always come at a cost. You should only use or keep a library when the benefit outweighs the cost.

Which benefits and costs am I talking about?

- **Minimalism**: External libraries are never tailored exactly to your use case. They will provide more functionality than you need and add some baggage to your project.
- **Documentation**: Often, external libraries are very well documented—better than most internal libraries I have seen so far. On the other hand, you might not need such comprehensive documentation if you have a stable team that developed the code by themselves.
- **Hiring/training**: If you use library "X", you can try to find developers who already know "X" when hiring. Or you can book off-the-shelf training for your team.
- **Learning curve**: Because libraries are never tailored exactly to your use case, and because they often do not use language from your domain, they can make learning *your* system harder for new hires.
- **Lock-in**: Some libraries and frameworks make it difficult for you to write code that is independent of the library. They try to lock you into their ecosystem. You'll have to write additional code to decouple your core functionality from the library. If you do not do that, your system will often become harder to test and harder to change.
- **Updates**: You must invest time in constantly updating all your libraries. Not keeping them up to date is dangerous, even if the update adds or changes functionality that you are not even using.

Those are just some examples to get your thinking started. Now meet with your team and look at the libraries, frameworks and technologies you are using. What benefits do they provide and at what cost?

Living with Legacy Code

Legacy code can be defined as…

> *“Code without tests.”*
>
> Michael Feathers, *Working Effectively with Legacy Code*

The idea behind this definition is that, without tests, code does not give us a way to verify its behavior. Without automated tests, a programmer cannot easily verify whether the code does what it is intended to do. But is not having tests enough to classify some code as “legacy code”?

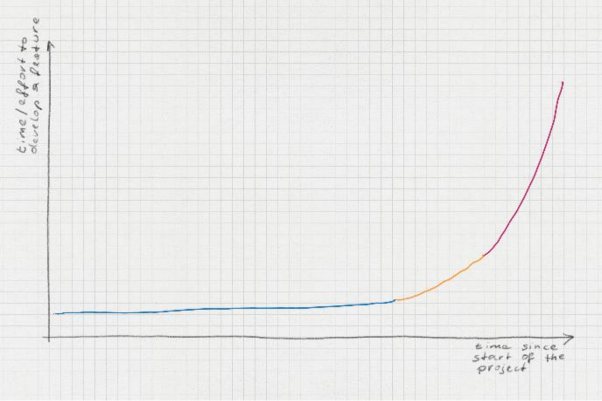
> *“Profitable code that we feel afraid to change.”*
>
> *J.B. Rainsberger*

If the code is not profitable or valuable, throw it away. If you are not afraid to change the code you do not like, change it. If the code is profitable **and** you are afraid to change it, you have legacy code.

But how did you even get there? Code does not rot; it does not decay. When you do not touch it, it stays exactly as it is. Code gets worse when you change it or when you add new code. So, whenever you change or add features, you must make a conscious effort to maintain its quality. You must invest time and money just to keep the quality exactly as it is.

Code is like plastic or toxic waste. It will stay where it is and you will have to attend to it, even if you do not need it anymore. You must dispose of old code.

This is essential because low-quality code will slow you down. Slow is expensive. If you are slower at developing new features, the features themselves become more expensive. And since you cannot start work on the next feature, you are adding opportunity cost. With bad code and design, the number of defects in your system will rise, and defects are ridiculously expensive.



But this increase in expense is not immediate. Your design gets a little worse, but nothing changes. Worse again, and still no change. This continues until suddenly you cannot get any software out the door. Until you end up in a place where nothing is cheap or easy anymore.

When you must live with legacy code, you must learn how to get over your fear of change. You must learn how to make safe changes and how to improve the design in small steps. I will explain how to do this in the coming chapters (and maybe in later books in this series) using:

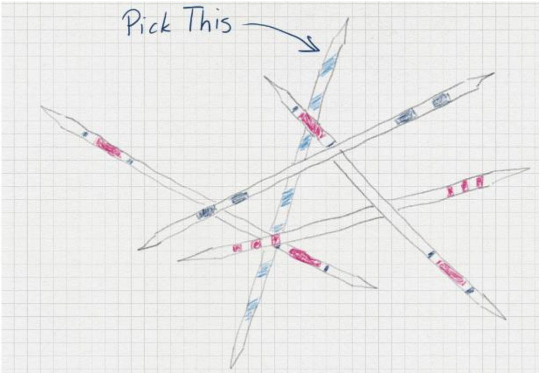
- The Mikado method (page 36)
- Testing legacy code (page 104)
- Testing seams (page 107)
- Dependencies (page 109)

But do not forget that you must invest time and money to get back to a point where adding features or changing code is easy again. Because right now, you are probably at a point where nothing is cheap or easy anymore.

Book: *Working Effectively with Legacy Code* by Michael Feathers

The Mikado Method

Do you know the game where there is a heap of sticks and you try to pick one up without making any other stick move? In some parts of the world, this game is called Mikado. The sticks you pick up all have a different score, and there is one stick that is the most valuable: the Mikado.



So, your goal when playing is to pick the Mikado without making any other stick move. Most of the time, you cannot achieve that goal directly, because there are other sticks lying on top of the Mikado. You must remove those first, meaning those sticks are your sub-goals.

When any stick other than the one you touch moves, you have lost the round.

Often, software development feels like playing Mikado. You try to achieve a goal, but no matter where you pull, something else moves, too. You must fix that other thing first, but when you forget to fix one of the moving parts, you lose the round.

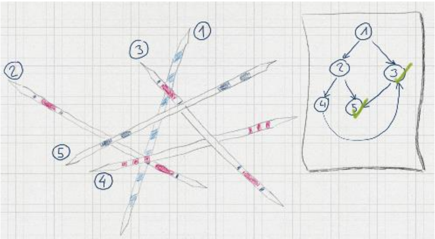
Think about fixing a defect. You change some code and the defect is fixed, but there is an unintended side effect somewhere else in the system. You found some closely coupled code, but you should have fixed the coupling before fixing the defect. If you forget to fix the side effect, you will lose because you have introduced another defect into the system.

Alternatively, think about refactoring legacy code. Let's say you want to move a method to a new class, but since the code is closely coupled, it has too many dependencies and its responsibilities are distributed all over the place. Everything around the method breaks when you want to move it. Now you must fix all those breakages, but you know that fixing them might cause more breakages and side effects. It's refactoring by compiler error. Wouldn't it be nice if you had fixed all those dependencies before you made that move?

So, how would you play Mikado if you had a time machine?

My description of Mikado here is actually an over-simplification. Your goal is not to pick the Mikado, but to pick a sequence of sticks with the highest value. You could still do better at that with a time machine.

If you had a time machine, you could just try a move, any move, like picking the Mikado itself. If it didn't work, if other sticks moved, you could simply remember which sticks moved, then go back in time and try one of those sticks instead. By doing so, you create a tree with your main goal (pick the Mikado) on top and branches of sub-goals below.



At some point, you will reach a stick that you can just pick up without any other stick moving. You have found a leaf of the tree. You made a successful move. You must now remember this point, because when you use your time machine again, you should go back to *this* point.

In software, we often have a time machine. You can go back in time using version control, like *git*.

When you do a complicated refactoring or make any other big change to the software, use the Mikado method to split it into small, safe steps:

1. Write down your goal at the top of a sheet of paper (or in an electronic tool). *Move method to new class.*
2. Try to achieve your current goal in as straight-forward a way as possible. *Cut and paste the method or use your IDE and ignore all warnings.*
3. Things will break. Analyze the errors (compiler errors, test failures, etc.) and write down a list of problems that prevented you from doing the straight-forward change. These become your sub-goals. *Uses private variables that cannot be moved yet.*
4. **Revert your code** in your version control system.
5. Try to achieve your next sub-goal in the most straight-forward way. *Cut and paste the private variable.*
6. If it fails, go back to step 3. If you succeed, commit your code using your version control system, strike the sub-goal through on your piece of paper and start to work on the next sub-goal.

If there are no cycles in your Mikado graph, this method **will** break your problem down into a sequence of small, safe steps.

[Blogs 6] https://www.davidtanzer.net/david%27s%20blog/legacy_code/2018/05/21/legacy-code-mikado-method.html
Book: *The Mikado Method* by Ola Ellnestam and Daniel Brolund

Refactoring vs. Rewriting

Refactoring legacy code seems tedious. It will take you and your team a long time. During that time, you will have a mix of old and new code, creating an unclean state. Because of the side effects of changing code, you do not know if you are going to break something.

Writing more tests and using structured methods, like small refactoring transformations or the Mikado method, will help, but…

Wouldn’t it be easier and cheaper to just throw away the whole thing and rewrite it from scratch?

Well, not so fast. You should at least consider a few things before you even think about rewriting a bigger software system from scratch. And after taking the following factors into account, you will likely come to the conclusion that a rewrite is probably not a good idea for most software systems…

- You can only consider a rewrite when you know *exactly* what the new system should do. “It should have all the features of the old system, only more modern” is a **huge red flag**. There is surely some hidden knowledge in the old system, so a project like this will become a software archeology project.
- Rewriting the software will be more expensive than you think, even if you add a buffer. When you compare the cost of rewriting vs. refactoring, the comparison is not fair. For refactoring, you will be pessimistic because you know the current mess. Estimates for the rewrite will always be overly optimistic because you think you will make everything better.
- During the rewrite, you must at least support the old system. This will cause your rewrite to take longer. Also, if you want to add new features, you must add them to both systems, because most of your users will use the old, proven system for a long time.
- Beware of the “second-system effect” where teams want to make *everything* better during the rewrite, so they create a gold-plated system that takes longer to build than necessary.

Rewriting software is almost always riskier than refactoring. During refactoring, the whole team works on the same piece of software, and you can add features or fix bugs while you are refactoring. Only rewrite if you have no other choice or when there is a crystal-clear business case for it.

[Blogs 7] https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/
[Blogs 8] https://stackoverflow.com/questions/2281619/tips-for-avoiding-second-system-syndrome

Throw Away Your Code

Be prepared to throw away the code that you wrote. While re-writing entire systems is often a bad idea, throwing away small bits of code that you just wrote will often result in higher-quality software. And it can even enhance your productivity.

Refactor by Compiler Error

I was working at a client on a greenfield application, and even though we were only a few months into the project, we already had some code that nobody liked. It was a central part of the application, so for some time, nobody dared to touch it, making the code around it worse.

We had already created "legacy code". It was not entirely our fault, because we were required to use a custom-made legacy application framework for *everything* and were constrained by the design choices of the framework.

But we could do better. So, I decided to refactor that central part.

I changed some interfaces and method signatures and started to fix all the compiler errors. When I fixed those errors, I had to change more code which resulted in more errors. After half a day, large parts of the system did not work anymore. I had written myself into a corner.

It took me two more days to get everything to compile again, and another day to fix the defects I had introduced.

Would I have been faster if I had thrown away my code after the first few hours and started over again? We can never know, but nowadays I'm pretty sure. And it would have been safer because of the defects I introduced when I did refactor by compiler error.

When you write yourself into a corner, throw away the code and start over again. When you solve the same problem for the second time, you will be faster and find a better solution. But how much code will you lose? That depends on how long it took you to write yourself into a corner and how much unsaved work you created along the way.

So, you can minimize the risk by creating "safe points" along your way. Keep your code in a working state and commit your code every time you've made some progress. Should you want to throw away code later, you can go back to one of those commits and start from there. Using the intermediate commits, you'll lose only a few minutes or hours, not days.

Write Throwaway Code: Spike!

Once you are prepared to throw away your code, you can take this concept to the next level. You can start to write code with the intent of throwing it away!

Suppose your team has been discussing two different designs as the solution to a problem. Both designs would come with advantages and drawbacks. Some on the team prefer one design, the others prefer the other design. Neither group can convince the others, so your team is stuck. Or suppose you need to solve a tough problem, but you have no idea how the final design will look. You do not even know which test to write first or where to start.

Create a “spike solution” when you want to learn something quickly. A spike is very simple code that you only write to learn something, e.g., to figure out a tough technical problem or to decide which of two designs to implement.

Decide who will work on the spike, set a timeframe and start to implement the solution to the tricky problem in a quick-and-dirty way. Or, in the case of a team with two design ideas, implement both design ideas for a limited time.

When you implement and manually test the solution, you will learn more about the problem and the solution. Once you are done with this quick-and-dirty solution, or when your specified timeframe is up, you will have gained some insights. Now that you have a badly designed solution, you will know what a better design should look like. You will have some ideas for tests that you could have written first and steps you should have taken to come up with a better design.

Keep the learnings but delete the code. Most spikes are not good enough for production. Remember: You always planned to throw the code away when you started the spike. Now, start over again, but this time, do it right. Write that first test. Take small, safe steps. Implement that better design.

This approach—planning what to learn, setting a timeframe, creating a spike, throwing away the code, starting over and making it right—will often be faster and result in cleaner code than simply planning what to do and doing it once.

Sometimes you keep the code. Sometimes you “only” keep the learning. This is OK. Sometimes the code is not worth keeping and the learning is more valuable than the code.

Unit Tests/Microtests

You, the developer, must make sure that…

- What you wrote down in the source code is what you wanted
- What the computer understood is what you wrote down

Sounds simple, right? Well, not exactly. When the systems we work on get larger, and when there are more moving parts, it gets harder to keep everything in one’s head. And when we change existing code, we also must take care to keep all existing functionality in place.

To do that, we write *microtests* or *unit tests*. These tests verify the correct behavior of the system for a very small, independent unit of code and/or for a small, independent piece of functionality.

> Microtest or unit test? I use the terms somewhat interchangeably here. It seems to me that both terms often describe the same thing, but from a slightly different perspective.
>
> A unit test is a function that checks whether a small, independent piece of functionality is implemented correctly. Most of your unit tests should be fast, require minimal setup and work on a single piece of production code.
>
> A microtest is “a small fast chunk of code that we run, outside of our shipping source but depending on it, to confirm or deny simple statements about how that shipping source works, with a particular, but not exclusive, focus on the branching logic within it” (GeePaw Hill, “The Technical Meaning Of Microtest”).

The result, whether the tests pass or fail, depends only on the correctness of the implementation of one piece of functionality—**not** on the correctness of large sub-systems or even external dependencies.

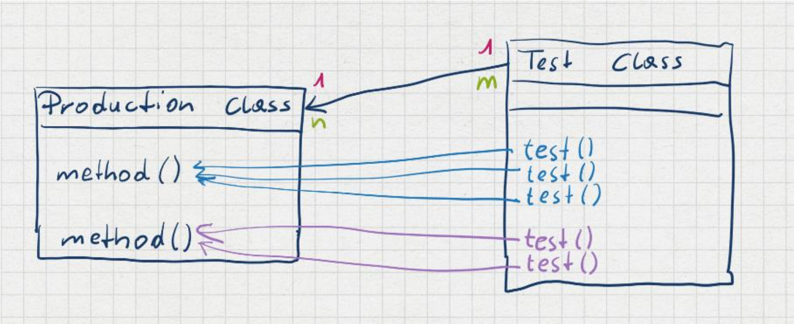
These are technical tests that support you, the programmer. They show you that the computer understood what you wanted. That a small piece of code is technically correct.

A unit test is just a single function that is not part of the production code. It sets up and runs a small piece of production code (the system under test) and checks the result. The test usually has three phases: arrange, act and assert. Make clear which part of the test code belongs to which of the three phases to enhance the readability of your tests.

Phase	Or	In BDD	
Arrange	Setup	given	Create the system under test with its dependencies; bring it into a state where you can run a test on it.
Act	Execute	when	Run a single command or perform a single action on the system under test.
Assert	Verify	then	Check whether the system under test has performed correctly. Whether the test has been passed or failed is decided here. This should be the **only** place where your test can fail.
	Teardown		Clean up any resources or states that your test has left behind.

A unit test or microtest should not need any teardown phase; it should never execute an action that leaves some state behind. If you wrote a test that needs a teardown phase, you probably have an integration test and will need to apply different rules than those used for microtests (see page 99).

We often group these tests into classes or test suites so we can handle and find them more easily. You can create different relationships between the classes in your test code (which are "just" groups of tests) and the classes in your production code.



A microtest usually runs a single public method/function from the system under test during its *act* phase. Usually, there will be multiple microtests or unit tests that call and test the same method. Each of those tests checks a slightly **different aspect** of the functionality, and they do not overlap.

There can be different relationships between the test classes and production classes (or test modules and production modules):

	Advantages	Disadvantages
1 … 1	Test class is easy to find (especially if you have a naming convention). Test class can be documentation for a production class.	Coupling is possibly too strong between the test code and production code.
1 … n	Might be the result of a refactoring or a design decision to decouple tests from production code.	Maybe your test class is too large or your tests are not focused enough.
m … 1	You try to create logical groups of tests for a production class.	Maybe your production class is too large.
m … n	Possibly very loose coupling between test code and production code.	Confusing.

Which of those relationships is “right”? It depends very much on your current situation. And you will probably find most of them in a sufficiently large code base.

You are pursuing conflicting goals here: Your tests should be easy to understand and serve as executable documentation of your production code. On the other hand, you need to avoid tight coupling between your tests and your production code; otherwise, your tests might get in your way later.

Getting this right from the start is almost impossible. So, start simple. Refactor when you realize that there could be a better way to structure your tests and your production code.

> Other types of tests will be covered in later chapters, in particular “Test-Driven Development” (page 59) and “Testing” (page 89).

[Blogs 9] https://www.geepawhill.org/2018/11/14/the-gold-of-microtests-the-intro/
[Blogs 10] https://www.geepawhill.org/2018/04/16/the-technical-meaning-of-microtest/

What Did You Learn?

Writing software requires you to make many, *many* small decisions. And while you are doing that, you must learn *how* to write the software you are writing and which software to write. If you already know what to write and how to write it, writing the software becomes easy.

The hard part of writing software is not programming. It is learning what exactly is needed and then learning how to create that.

You must learn to take smaller steps and to break large tasks into a series of small steps, ensuring that each of them is recoverable. You must learn to decouple different parts of the code, make them testable and write fast, small tests.

What did you learn from this chapter?

Write down your
key learnings here

Your Tools

“Git’s a very powerful tool, and rewriting history is a pretty dangerous endeavor. Put your steel-toed boots on first.”

Samir Talwar – @SamirTalwar

Master Your Tools

Do you know that feeling, when you know a tool well and you are working with someone else who is using said tool inefficiently?

For instance, when you are pair programming with someone and they run a command with the mouse, but you know the shortcut. It’s frustrating, right? You see how much faster they could go, but they still take the long route every time.

Have you ever wondered whether and *when* you are that slow person to someone else? Whether you could do all those boring, tedious tasks faster and more efficiently?

Which tools do you need for your job? This list might include:

- A general-purpose editor, such as Visual Studio Code or vim. Pick one that is good at general-purpose tasks but also has support for your programming languages.
- An IDE like IntelliJ IDEA, Eclipse, Visual Studio, etc.
- A way to navigate the file system. I mostly use a Unix-style shell, but I have seen others who had wizard-like skills in Total Commander, Finder or Windows Explorer.
- A version-control system. I used to prefer Mercurial over everything else. Nowadays, I use Git whenever possible because it has become much more widespread.
- A way to automate stuff, like shell scripts.
- A web browser. Websites to answer your questions (Google, Stack Overflow, etc.).

Pick your tools and **learn them well**. Do some tutorials. Learn the shortcuts. Your web browser, for example, might support more shortcuts than you know. And there are even tools for mouse-less browsing for many browsers. In my IDE, I have a plugin running that tells me the shortcut whenever I do something with the mouse.

But not touching the mouse is not always the goal. Sometimes you are faster using your mouse.

Whether you use your tools with or without your mouse, learn them well. You will become a more efficient software developer **and** spend less time on boring tasks.

[Blogs 11] https://www.davidtanzer.net/david's%20blog/2012/04/02/cheap-plastic-drills.html
[Blogs 12] https://www.davidtanzer.net/david%27s%20blog/2012/08/21/kitchen-knives-and-other-tools.html

Typing/Your Keyboard

Typing is usually not the bottleneck of developing good software. You will spend more time reading code, discussing and communicating with colleagues, business people and users, as well as thinking about designing an architecture, deciding what to test next, etc.

Since typing is not the bottleneck, your typing speed does not always reflect how effective you are as a developer. I have met some great developers who are slow at typing.

However, you should still learn to touch-type.

When you are pair programming or mob programming, you must get the writing out of the way, since it is the easiest part. Also, typing slowly will frustrate your pairing partner or your team in the long run.

Even when you work alone, being able to type faster and never look down at your keyboard can be less frustrating for you, personally. I learned to touch-type in school, but I was not very good with special characters. That frustrated me when I began programming, so I practiced. Now, I only look down at my keyboard every now and then.

I have even talked to people who switched to different keyboard layouts, like Dvorak, to become even faster.

Learn to touch-type to get the writing of the code out of the way. Then, you will have more time to think, discuss and decide, as well as time to take a pen and write, draw or doodle on a sheet of paper, a flip chart or in your engineering notebook.

[Blogs 13] https://en.wikipedia.org/wiki/Touch_typing
[Blogs 14] https://en.wikipedia.org/wiki/Dvorak_keyboard_layout

Your Operating System (OS)

Are you using Windows, Linux or Mac OS? It does not really matter which one you choose. Yes, for some jobs, you will need a Mac, like when you want to develop software for the iPhone, but there are also things that can only be done well on Windows or on Linux.

In general, it does not matter which one you choose. You can develop software just fine with any of them. What matters is that you learn your operating system so well that it does not get in your way anymore.

Can you find your **documents and other files** quickly?

Every operating system requires you to store and organize your files differently. You should learn what the OS expects of you and mostly stick to it. Inside those rules, you must find your own way of organizing them further.

Also, learn to navigate the file system quickly. I have seen people who are extremely good at using Finder or Windows Explorer. They know all the shortcuts and navigate to their destination in almost no time. When I saw that, I knew that I had to learn something. I got better, but I am not there yet.

Do you have all the programs you need, and do you know the **additional tools** that can make your life easier?

For most operating systems, there are several small tools that either come with the system or that you can install later. Here are a few that I learned to love:

A program for taking screenshots, a clipboard manager, a screen ruler, a screen color picker, a tool to disable unnecessary downloads and syncing when my computer is tethered to my cell phone and a password manager.

Can you find the most important **settings** and change them to fit your current needs?

For example, can you change how the computer treats different screens, e.g., during a presentation? Do you know how to remove saved credentials?

Your OS is there all the time when you use your computer, so it should not get in your way.

Book: *The Pragmatic Programmer* by David Thomas and Andrew Hunt

The Command Line

Some things are done better at the command line. Some tools provide **more options** when you use the command-line version. Often, it's **easier to automate,** and sometimes working on the command line will **help you think** about your problem because you must write the solution down.

Every operating system comes with a terminal/console program and a *shell* (the programs that interpret your input and run commands).

On Windows, you have *cmd* and *power shell*. On Unix-like systems (like Linux or Mac OS), you have *bash*, *zsh*, *fish* and others. Most of the Unix-like shells are available on Windows, too.

All shells provide features that allow you to write scripts. Small programs to automate your tasks when interacting with your operating system. You can enumerate files, loop over them, write `if` and `switch` statements, write and call functions and call into other programs.

Learn the scripting language of your shell. You don't have to know it perfectly, but learn what is possible, learn to automate simple tasks, and learn where and what to search for when you need help.

In addition to the scripting language, your environment will provide some command-line tools that will make working with the command line more effective for you. I use the Unix tools on Windows, and I run them in ConEmu. Those tools usually read from standard input (or from files) and write to standard output. One can chain them with pipes, where the standard output of one tool becomes the input of the next. Some of these useful tools include:

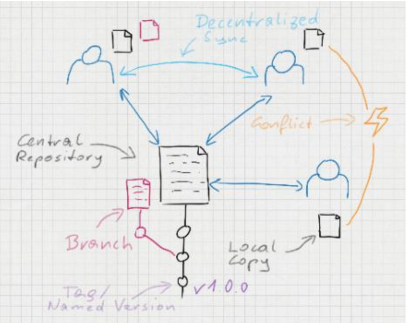
- `find` – Traverses directories and searches for files/directories with certain names. It can also execute another program for each file that was found by passing `–exec`.
- `grep` – Search for text/regular expression in its standard input or in one or more files.
- `less` – View/search/analyze the contents of a file or standard input. Also have a look at `less +F` – For following files that another process writes to.
- `xargs`– Takes the contents of a file or stdin, splits it into lines and calls another command passing every line of the input as a command-line argument.
- `...`And many more. Check the documentation and tutorials to find out what else the command line can do for you.

Version Control

You need a way to track changes to the files of your program. This tracking must be atomic, meaning that when you change three files together, it must be guaranteed that someone else working on the files also sees the three changes together. A good **version-control system** (or VCS) must also track who made the change, allow for branching/merging and give you a way to jump back and forward in the files' history.

Today, all teams use some type of version control—or, at least, I hope so! The days of copying files to and from shared folders should be over. But some teams are using version control ineffectively or not to its full potential.

> Let's be realistic. **`Git`** has won. So, even if your team uses another product, make an effort to learn `git`. Knowing `Git` can be beneficial in your next job or when you want to use or contribute to open-source software.
>
> Also, `Git` exposes its internals if you want it to.



All VCS allow you to have a central repository that you can synchronize your files with. Some systems, like **Subversion**, require it. With decentralized systems, such as **Git** or **Mercurial**, the centralized repository is optional.

Your VCS allows you to synchronize files with a remote repository and resolve conflicts when multiple developers work on a file. Some require you to lock files. Do not use them, as it prevents collaboration. You can also have branches (where multiple versions of the same file exist at the same time) and named versions (often called "tags").

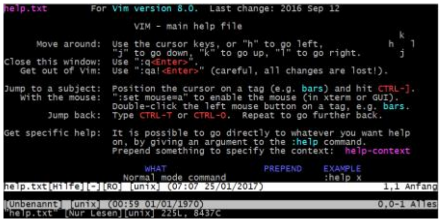
Using version control well is crucial for working together effectively as a team. In addition to that, a version-control system allows you to find out retroactively when something has happened (e.g., when a bug was introduced) and what exactly has changed. It also allows you to undo your work when something has gone wrong, e.g., when you use the Mikado method.

Book: *The Git Book* by Scott Chacon and Ben Straub
[Blogs 15] https://www.mercurial-scm.org/wiki/UnderstandingMercurial

A Text Editor

Some people do almost all their work in a text editor, while others prefer an IDE for most tasks (see next chapter). But even if you prefer an IDE, you should learn to use a text editor *really* well. Some tasks are easier with a text editor, and sometimes, a text editor is all you have.

Learn `vim` (or `vi`), even when you do not plan to use it as your main text editor. When you use the command line, being able to quickly change some characters in a file without switching to a different window is great.



```
help.txt        For Vim version 8.0.  Last change: 2016 Sep 12

                        VIM - main help file
                                                                         k
      Move around:  Use the cursor keys, or "h" to go left,            h   l
                    "j" to go down, "k" to go up, "l" to go right.       j
Close this window:  Use ":q<Enter>".
   Get out of Vim:  Use ":qa!<Enter>" (careful, all changes are lost!).

Jump to a subject:  Position the cursor on a tag (e.g. bars) and hit CTRL-].
   With the mouse:  ":set mouse=a" to enable the mouse (in xterm or GUI).
                    Double-click the left mouse button on a tag, e.g. bars.
        Jump back:  Type CTRL-T or CTRL-O.  Repeat to go further back.

Get specific help:  It is possible to go directly to whatever you want help
                    on, by giving an argument to the :help command.
                    Prepend something to specify the context:  help-context

                          WHAT                      PREPEND    EXAMPLE
                      Normal mode command                      :help x
help.txt[Hilfe][-][RO] [unix] (07:07 25/01/2017)                        1,1 Anfang

[Unbenannt]  [unix] (00:59 01/01/1970)                                   0,0-1 Alles
"help.txt" [Nur Lesen][unix] 225L, 8437C
```

Writing/editing a commit message for your version control system directly in the command line using `vim` can be a huge timesaver.

`vi` uses some interesting concepts for editing text. It has different modes (writing, selecting text, entering commands, etc.) and behaves differently in each of those modes. Every mode is optimized so you can use it very effectively.

Even if you do not use `vi`/`vim` as your main editor (like me), it will pay off when you learn it. You will learn an editing paradigm that is different from almost everything else and that can be very effective. And you will also know how to use an editor that is available virtually everywhere.

Some people choose to use `vi`/`vim` as their main editor. That can be very effective *if* you know the editor, its commands and how they work. But when you do not want to use `vi`/`vim` as your main editor, you should choose an editor that:

- Allows you to do most things without touching the mouse
- Has at least some support for your programming and scripting languages
- Can work on large files without losing too much performance
- Has a reasonably fast startup time

Then, you can learn the editing paradigm, the most important keyboard shortcuts and some clever tricks. Learn how to select, search and replace text. Learn the regular expression syntax of your editor and regex replace. A text editor can be one of the most versatile tools in your toolbox, so get great at using yours.

An IDE

Sometimes, a text editor is not enough. Sure, you can do almost everything with a good text editor and some additional tools, but an integrated development environment (IDE) can sometimes provide features that make using it more effective than using just a text editor plus some tools.

Whether an IDE (Like IntelliJ IDEA, Cursive, Visual Studio, Eclipse and others) makes sense for you depends on what technology you are using and whether you prefer other tools around those technologies.

Personally, I would not write Java code without an IDE, but I prefer an editor for working with JavaScript, C# (I use an IDE), Python, an editor, and so on.

I like to use some tools separate from my IDE, even though IDEs have support for them. Take `Git,` for example. I almost exclusively use it from the command line. The only things I use the Git integration of my IDE for is to annotate who changed which line (blame) and to quickly diff versions of a file.

If you use an IDE, **learn to use it well**. Browse the menus and the available plugins to learn new features. Learn keyboard shortcuts for all the functions you use regularly.

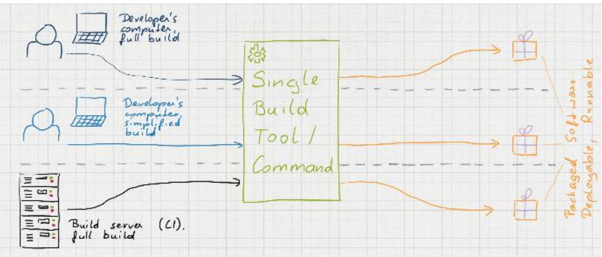
Some IDEs even have plugins that teach you keyboard shortcuts, like Key Promoter X for IntelliJ IDEA and related IDEs.

Do you know people who are "IDE wizards"? What would you like to ask them?

What could you teach others about using your IDE?

Single-Command Build

When you want to deliver quickly as a team, you need a **single-command build** that is reasonably fast. Without it, doing continuous integration or continuous delivery will be cumbersome at best and error-prone and laborious at worst.


Developer's computer, full build
Developer's computer, simplified build
Build server (CI). full build
Single Build Tool / Command
Packaged Software, Deployable, Runnable

Having a single-command build means that you can run a single command that will build a deployable/runnable version of your software from scratch (i.e., a clean checkout from version control) and works on *any computer* and in *any environment*, whether that is a developer's machine, an integration server, etc.

This build command must be able to:

- Resolve dependencies between modules, libraries and versions
- Ideally, fetch missing dependencies if necessary
- Compile the source code if you use a compiled language
- Run tests
- Run static analysis tools
- Package the software for every target environment (or, ideally, create a single package that can be run in *any* target environment)

For almost every language, there are tools that will allow you to create this single-command build. Some examples are Maven and Gradle for Java, npm and yarn for JavaScript, etc. These tools often do not implement all the above features by themselves. They might run other tools instead. But to you, the developer, this is transparent. For you, they provide a single command to build everything.

Sometimes you want to run simplified builds on your own computer (e.g., without running all the test/analysis steps). Sometimes you might even build your software using your IDE.

But do not *rely* on your IDE or simplified tools to build your software. The only *official* way to build is with a single-command build.

[Blogs 16] https://en.wikipedia.org/wiki/Continuous_integration#Automate_the_build

Internet Skills: Skimming, Searching, Researching

Internet skills? Isn't this obvious? Almost all of us grew up with the internet!

Don't dismiss this chapter too early. The internet, especially search engines and blogs, is one of your most important tools today. Yes, almost everyone in our industry has grown up with the internet, but maybe you can improve how you use some of these tools or avoid some pitfalls.

Nowadays, you can find almost anything online. And you should learn to find those things quickly.

You probably use a **general search engine**, like Google, daily. Searching for information is the easy part—though you'll probably have to experiment with search terms a bit. However, this search will turn up more information than you need. Finding the *right* result is the hard part.

A colleague and I were searching for something online. I opened a result, quickly scrolled through the page, only stopping briefly at some points, and then told him: "This is the library we are looking for. It solves problem X by providing functionality Y." He was baffled. "How can you know that? You can't possibly have read the article so quickly!"

Learn to **skim pages** quickly to filter the results of a general search engine! I read the first sentence of each paragraph, headlines and everything that's in bold. This is often enough to know whether the page you are looking at is an interesting result or can be discarded.

You will also use forums where you **can ask questions and find answers** to your problems. Learn to ask good questions. Provide context: *What were you trying to do?* Show people that you already did some research: *What did you already find out?* Help people to reproduce the problem: *What exactly is the question?* In other words, make it as easy as possible for others to answer your question.

When you look up answers on one of those forums, also look at the second or third answer for additional insights.

And when you answer a question, be welcoming, friendly and patient. Never be condescending or tell the person who asked that what they tried was easy anyway. You don't know where they are in their journey. What is easy for you right now might still be hard for them.

Most importantly, learn about **copyright and licenses**. Know exactly what content on the internet you can use, how you can use it and why.

What Did You Learn?

How well do you know your tools? Can you use the most important features without touching your mouse? Do you know at least some advanced features? Have you explained features to your colleagues or even to some of your more senior colleagues?

Can you quickly find solutions to your problems and identify new features of your tools using internet search engines?

Write down your key learnings here

Test-Driven Development

“A microtest proves, for an almost comically small subset of the shipping app, that what the geek said is what the computer heard is what the geek wanted.

That’s all. That’s what a microtest proves.

I mean, c’mon, really? That’s all a microtest proves? The geek said X. The computer heard X, the geek wanted X?”

GeePaw Hill – @GeePawHill

Why Do We Unit Test?

Writing unit tests can have two benefits:

- Writing unit tests helps us understand the problem and the solution and develop faster
- Unit tests can help us catch problems and find their causes later

Writing tests before and while we develop new code can help us **understand the problem and the solution** better. By writing the tests (and having a list of tests we intend to write), we know what to do next and what exactly is still missing. We also know when to stop. Writing tests first can prevent us from gold-plating. In other words, from implementing too much functionality or design for the current use case.

Writing the tests also helps us to break down a large problem into small steps and to verify that every step was correct. It gives us the possibility to go back in time when we did something that led us to a dead end. We can go back to a step where all the tests were green and continue from there.

To be able to do that, we must write tests that each take us a tiny step closer to our goal. We must also write production code that only satisfies the current test. When we write more code than is strictly necessary to pass the current tests, we lose the advantage that tests can guide us towards the final implementation.

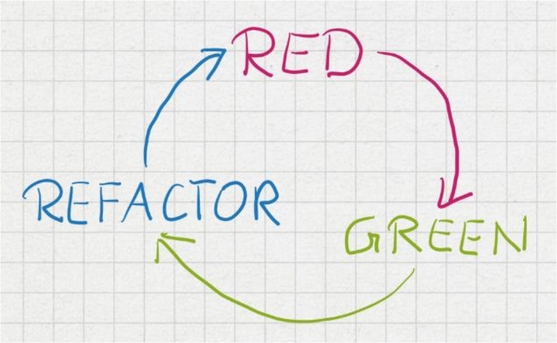
Later, after we have finished the implementation of the current feature, our tests can help us **catch problems and find their causes**.

When a test fails, the message the test gives us should make it easy to find out what the expected result was and why the actual result was different. This means that for every problem that could occur, we need tests that fail when it does. And for every test, there should only be one problem that makes it fail.

But we must also be aware of false positives. These are tests that fail even though the production code still behaves correctly. Those tests could be “flaky” tests that fail from time to time. They erode your trust in your tests quickly. Alternatively, they could be tests that did not survive a refactoring. They were broken by a valid code change and will slow you down. Work hard to avoid both.

Getting all this right is hard. But you can get better at it. Take small steps to start improving now!

Test-Driven Development



Write a test. Run it. See that it fails and that no other test fails. **RED**.

Write the simplest possible code that could make the test pass. Run all your tests. **GREEN**.

Look for opportunities to make the code better. Look for code smells. Improve your code, keeping all tests green. **REFACTOR**.

Test-driven development, at its core, consists of these three steps. Sounds easy, right? So, why do some programmers and teams struggle to implement it? Why do they say it does not work? How can it even work?

Test-driven development (TDD) **can** help you to write better code, to create better designs and to ship features faster and cheaper. But only if you do it right.

TDD will not automatically lead to better, nicer, cheaper code. In fact, if you do it wrong, it can even have the opposite effect. It can cause you problems down the line, making future development slower.

Test-driven development sounds simple, but it is not easy.

On the other hand, it does not take much to get started. In the following sections, I will give you some details that might be helpful when doing TDD. But to get started, let’s just begin. Write a test. See it fail. Make it green. Improve your code. Now, you’re doing TDD.

Try This: Implement a small exercise using test-driven development.

Implement a program that can play the game “hangman”. Your program will take over the part of choosing a secret word, letting the player guess, showing them a hint and keeping track of the game state. Do **not** add any user interface; only implement the rules.

https://en.wikipedia.org/wiki/Hangman_(game).

Basic Rules

How did that feel? Was it hard? Did you only write production code when there was a failing test? Did you write only as much code as was needed to make the test pass? Did you refactor?

Again, test-driven development sounds simple, but it is not easy. In my training sessions, I ask people to write some code using TDD after I explained the red/green/refactor cycle. Most of them struggle.

They find it hard to start writing a test when given a completely blank slate. They take steps that are too big. They write too much code for the current test. They forget about refactoring. They find it hard to decide which test to write next. They test too much and too little at the same time.

After this first exercise, that *you* just did too, it is time for some basic rules:

- Think before writing code, and this includes test code. If you are not sure where to start, draw a diagram, write down a list of requirements, write down the happy path flow, or do whatever else might help you to better understand your next step.
- Test-drive the happy path first. Start with something important to the user (like rendering the hints in “hangman”) and finish that. Leave the special cases and error handling for later, but **do not forget** about them.
- Test through the public API only. If something is hard to test through the public API, you often have a design problem. Your class/module/function under test is often either doing too much or too little.
- Only test *your* code. Do not write any tests for results that your code cannot influence. Do not write any tests where the results solely depend on the functionality provided by your runtime or a library.

Keep in mind that you must iterate quickly. Do not overthink when moving from red to green. Just write some code. Spend more time thinking about **refactoring** and about writing the **next test**.

Take Smaller Steps

When you test-drive your code, make sure you take the smallest steps possible. Say you are working on the hangman game and you want to test generating the hint, which contains the letters that the user guessed correctly and placeholders for all other letters of the word.

A first test and corresponding implementation in pseudo-code might look like this:

```
@Test
public void singleLetterHint() {
    h = new Hangman(“a”);

    hint = h.generateHint();

    assertThat(hint).isEqualTo(“_”);
}
```

```
public String generateHint() {
    return “_”;
}
```

This test code tries very hard to test the simplest possible example. It uses a very simple word (“a”), and it generates the hint at the start of the game.

The production code tries very hard to implement the simplest possible solution to make the test green. It returns a constant value.

Why am I writing “wrong” code here? I mean, I will have to change that single line of production code in five minutes when I’m writing the next test!

Yes, I did write very simple (even comically simple) code here, but I made some progress nonetheless:

- I made progress on the API. With this first test and simple implementation, I have defined the API for generating hints.
- I made progress on the tests. Before, I had no tests. Now, I have a test that tests **one** crucial aspect of generating hints and that I expect to stay green from now on.
- I took smaller steps than in a test-after approach. Small steps help me to write only the necessary code and to make sure all code is tested.

Triangulation, Specific/Generic

What I did in the last section is the first part of a technique called **triangulation**. I am using indirect measurements from at least two sources (tests) to lock down the desired behavior.

I wrote a concrete example of the expected behavior. I wrote the first test. Then, I wrote the simplest possible implementation that will satisfy this test. I will only add more code when I have more examples and when I have more tests (pseudo-code again):

```
@Test
public void twoLetterHint() {
    h = new Hangman(“it”);

    hint = h.generateHint();

    assertThat(hint).isEqualTo(“_ _”);
}
```

```
public String generateHint() {
    return secret
        .map(ch -> ‘_’)
        .interleave(‘ ‘);
}
```

This implementation is still not complete. It can **only** generate hints at the start of the game when no letter has been guessed correctly yet. But it can already do that for **all** possible secret words.

I will then add more examples to more tests to drive the correct implementation. I *triangulate* the proper implementation with many tests, all of which contain concrete examples of the desired behavior.

I cannot directly test whether the `Hangman` code contains the correct secret word. The code is supposed to encapsulate that knowledge. However, I can use indirect measurements to make sure the code knows the secret by observing what the code does with the secret.

In doing so, I follow the “specific generic” rule, which states that “As the tests get more specific, the code gets more generic.”

The first version of the code was not generic at all. It only worked for single-letter words. Then I added a second, more specific test. And this test forced me to write more generic code—code that works for all possible words, but only if the user has not guessed a letter correctly yet.

[Blogs 17] http://feelings-erased.blogspot.com/2013/03/the-two-main-techniques-in-test-driven.html
[Blogs 18] https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

Who Tests the Tests?

So, if we test all our production code using automated tests, and those tests are code again, who tests our tests? People often ask me this question jokingly. They do not even realize that this is not a joke. Instead, it is an *excellent* question!

We usually do not test our tests on an ongoing basis, as we do with production code. However, we make sure that we see each test fail once—when we wrote the test but had not yet written the production code. This is our **only** indication that the test is verifying something important. So, make sure that you **always** see the red bar!

And make sure that you see the red bar on a correctly written test. The easiest way to check this is to go from red to green without modifying the test. If you cannot do that, your red test was wrong. Time to start over.

Sometimes, writing the next test that I want to write is not possible with the current implementation. At least, I can't write it in a way that would cause it to fail. Remember: Seeing the red bar is important. So, what can I do? Often, I can write another test and see it fail. And adding a very simplistic implementation will allow me to write the test I first wanted to write **and** see it fail.

Suppose that, in the hangman example above, you want to confirm that the hint did not change when an incorrect letter was entered. You cannot write this test right now in a way that is both **red** and **correct**.

But you can write a test that checks whether the first letter is shown when it is entered correctly:

```
@Test
public void guessedFirstCorrectly() {
    h = new Hangman(“it”);

    h.guess(“i”);

    hint = h.generateHint();
    assertThat(hint).isEqualTo(“i _”);
}
```

```
public String generateHint() {
    if(guessedLetter != null) {
        return “i _”;
    }

    return secret
        .map(ch -> ‘_’)
        .interleave(‘ ‘);
}
```

Now you can write more tests about guessing letters correctly. **And** you can also write the test you wanted to write first—the one testing that the hint did not change when an incorrect letter was guessed.

With this almost-too-simple implementation, you can even write it in such as way that it will be both **red** and **correct**.

And this is another reason why making small steps is so important. These small steps enable us to write the next failing test.

Seeing the test fail once, in the beginning, is often our only indication that the test itself is correct. It is our **only test for the tests**.

And that is enough, at least most of the time. Seeing the test fail once is all we need to know to have confidence in our tests. There is no need to spend more time and energy testing our tests.

But what if we *really* want to test our tests?

In that case, we can do mutation testing. A mutation test tool will introduce changes in your code and then run your tests to check if the tests catch the mutation.

When at least one of your tests fails, the code change was found by your tests and the mutation is “killed”. When the tests stay green, the mutation has “lived”.

That means a mutation testing tool will run your test suite against modified versions of your code. And it will expect your tests to break for most of those modifications.

So, if you *really* want to test your tests, take a look at mutation testing.

[Blogs 19] http://pitest.org/
[Blogs 20] https://www.devteams.at/red_green/2019/03/18/red_green_part_3_why_red.html

Good Names for Tests

Read the tests I wrote in the last few sections again. Take a close look at the naming. Do you like those names? What is good about them? What is bad?

What qualities should a name for a test have? When and why do you need the name?

- The name does **not** have to be short. You will write it only once. You will hopefully never change it and you will probably never re-read it. As long as the test is green, you do not need to care about the name. Still, choose the shortest name that conveys all the necessary information.
- You will only re-read the name when the test is red or when you want to find out again how the system under test is expected to behave. So, the name should support you in those two situations.
- The test is a concrete example of how we want the system to behave. So, the name you choose should convey some information about the example. What are the pre-conditions? Which action does the test execute in the system under test? What is the expected result? Why is this example significant?
- When the name of the test answers the questions above, it will help you find out what went wrong when the test was red. Just by reading the name, you will know what the test tried to do, in which context it tried to do it, and what the expected result was.
- Use a name that is an English sentence. The sentence should describe the correct behavior of the system under test. When the test is green, this statement about the system is true.
- Try to avoid redundant prefixes or postfixes, like “test…” or “should…”.
- Maybe use underscores to separate parts of the name and make the name more readable.

Given these qualities, what would be good names for the tests in the previous sections?

```
singleLetterHint      →      returnsSinglePlaceholder_ForOneLetterWord_AtStartOfGame
twoLetterHint         →      returnsTwoPlaceholders_ForTwoLetterWord_AtStartOfGame
guessedFirstCorrectly →      updatesHint_WithGuessedLetter_AfterCorrectGuess
```

Are those names perfect? Probably not. Would you name the tests differently? That is OK. But **then**, think about the quality attributes of the test names.

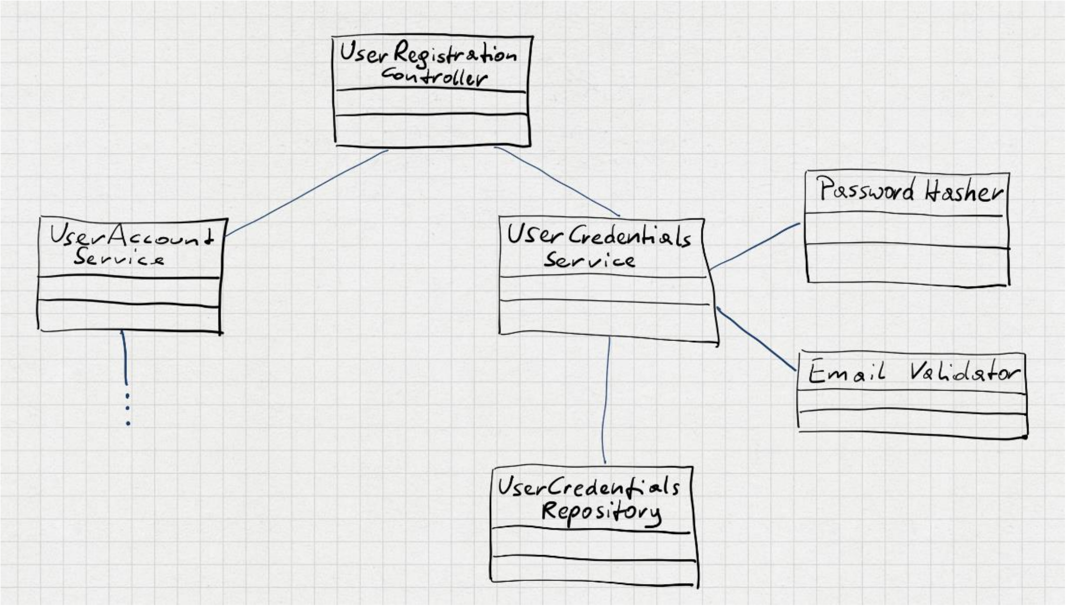
Outside-In vs. Inside-Out

Up until now, I have talked about test-driving a single class or module. But how does that translate to larger applications? How do you implement a whole system with lots of classes/modules in a test-driven way? There are two main approaches, outside-in and inside-out, and some shades between.

What I have shown you so far has been inside-out, test-driven development. We started with a class/module/piece of code on the inside of the program (the "game rules" class). You implement this in a test-driven way, then you move further out, possibly building your new classes and tests on existing code.

When you take an outside-in approach, you start at the other end. You start with the code that deals with user-visible behavior, like a controller for a UI or rest service. You start at the outside of your system and slowly move further inside, inventing the parts that you will need later as you go.

Let me compare the two approaches with an example. Say we want to implement a user registration process for a website. The end result will look somewhat like this with both approaches:



And here is how you could proceed using one or the other approach:

Outside-In	**Inside-Out**
Start with the class that is closest to the user. That is **UserRegistrationController**.	Start with a class that has no dependencies or where all dependencies are already implemented. Maybe **PasswordHasher**.
Invent the dependencies of the class under test (**UserRegistrationController**) and the interfaces of those dependencies as you go. Use your judgment to decide which dependencies and interfaces you will need.	Invent the interface of the class under test (**PasswordHasher**) as you go. Use your judgment to decide which interface all potential callers of this class might need.
Since there is no implementation of your dependencies yet, you can either fake it until you have a better implementation or use test doubles, like stubs, fakes or mocks (see next chapter).	Since your class under test does not have any dependencies—or since all dependencies are already implemented—you can build upon these implementations.
Come up with tests using what you know about how the class under test will be used. You know the requirements of your current feature. Further up the graph, you already wrote tests for the classes using the class under test, so you know how it will be used.	You can come up with tests as examples for what the class under test does and as examples of how you want others to use the class. You know what this class should do and how others should use it because you have thought about its responsibilities and collaborators.
When you are done implementing what you need to **satisfy the current requirements** (and tests further up the graph), you move further **to the inside**.	When you are done implementing the **current responsibilities of the class**, you move further **to the outside**.
You continue to implement **UserCredentialsService**, **PasswordHasher**, **EmaileValidator**, **UserCredentialsRepository**, **UserAccountService**, **etc.**	You continue to implement **EmailValidator**, **UserCredentialsRepo.**, **UserCredentialsService**, **UserAccountService**, **UserRegistrationCon, etc.**

So, which one is better? All right, you probably already know that this question is a trap.

Both have their advantages and disadvantages. They work well in some situations and not so well in others. In a real-world project, you might need both approaches to get the best result.

But do not mix the approaches. Do not add inside-out style tests to a group of tests that were written using the outside-in approach and vice-versa.

So, what are the pros and cons? When do you use one approach over the other?

Inside-out works well when you already know which parts you will need. With outside-in, you invent the parts as you go, and you will have an even-more emergent design. For the same reason, inside-out works well when you want to test-drive the implementation of an algorithm you already have in your head.

With outside-in, it is easier to know when to stop. You start from a user requirement, and at each stage, you only implement what you need to satisfy that requirement. Inside-out, on the other hand, might result in more consistent APIs.

With outside-in, you must take care that the tests at each point on the graph reflect the expectations of the objects further outside. The expectation is that you codified in the fakes or mocks when you wrote the tests further outside. With inside-out, you build tests upon existing production code, so your unit tests are not entirely independent of other units. You must take care not to have cascading test failures.

Most people find inside-out easier to learn. It probably requires less design experience than outside-in.

As a developer on an agile team, you should know both approaches. So, practice them. Often, one of the two will feel more natural to you. Use this approach more often, and only use the other one when your current situation requires it. You should also discuss these approaches with your team when you talk about your testing strategy.

[Blogs 21] https://8thlight.com/blog/georgina-mcfadyen/2016/06/27/inside-out-tdd-vs-outside-in.html
[Blogs 22] https://medium.com/@erik.sacre/clean-architecture-through-outside-in-tdd-64a31de17ccf

Test Doubles

Sometimes, in your tests, you want to replace a real dependency with something simpler. An object that looks somewhat like the real object, but only for the aspects the test cares about.

Using the real dependency might not be possible at all, maybe because it would be too dangerous or because it only exists in certain environments. Or perhaps because the real dependency would make your test unstable because it never works on Wednesday evenings. Or because using the real dependency is just impractical as it is too slow or too complicated to set up.

In those cases, you use a “test double” instead of the real object or sub-system.

But not all test doubles are equal. According to Martin Fowler’s classification (originally coined by Gerard Meszaros), there are five categories of test doubles:

- **Dummy**: An object you need to make the test work, but the test does not care about it.
- **Stub**: When the system under test calls a stubbed function, that function will always return canned answers. For example, it would always return “5” or return “5” on the first call and then always throw an exception.
- **Fake**: The fake has an implementation, but one that is simpler or safer than the original one.
- **Mock**: An object pre-programmed with expectations of what calls it will receive.
- **Spy**: A stub that also allows you to verify later whether the system under test used the spy exactly as you expected. For example, you can ask the spy whether a certain function was called or not.

Note that this definition is somewhat different from how some modern mocking libraries use the terms. In *Mockito*, for example, a “mock” is an object that can be verified later (called “spy” in the above list), while a “spy” proxies a real object and also records the calls for later verification.

To me, the important distinction here is that stubs and fakes **answer calls** while mocks and spies allow us to check whether **an interaction happened** in the system under test.

You can create all these test doubles without a mocking library. There is no magic involved. A mocking library, should you use one, only removes some of the boilerplate involved in creating test doubles. It usually lets you configure the behavior of objects and verify whether they were used correctly by the system under test with very little effort.

Tests with mocks usually have the three phases, too, but here they are often called “stub”, “execute” and “verify”. The exact phases and syntax depend on the mocking library you use, but in Java-like pseudo-code, the test will look somewhat similar to this:

```
user = mock(User.class);
us = mock(UserService.class);
when(us.getUser()).thenReturn(user);


classUnderTest = new LoginService(us);
classUnderTest.login(…);



verify(us).loggedIn(eq(user));
```

“mock” creates a new **mock object** (or spy).

“thenReturn” **stubs** the method *getUser* to always return the same user.

The class under test/code under test uses one or more mock objects.

“verify” **checks** whether the class under test has called the method *loggedIn* with exactly the argument *user*.

Never verify a call that you have stubbed. Whether the test uses the stubbed value correctly or not will have an effect you can observe without verifying. When the stubbed value is not necessary for the correct behavior of the code, you should also not care about whether the function was called.

Test doubles can help decouple the classes in your production code. They can help you to write independent tests and to make your tests repeatable.

Mock objects allow you to do interaction-based testing. They allow you to verify in your tests that an *interaction*, usually a method call, took place. This enables you to write different tests than in state-based testing, where you check for the *result* of an action. A single object can serve more than one category. You may have stubbed some methods and verified others using the object as a mock. However, do not verify a function that you also use as stub.

Also, do not mix interaction-based and state-based testing. Do not **verify** an interaction and **assert** a result in the same test, and maybe not even in the same test group. Most of the time, it will make the tests harder to understand, which will make the test code and production code harder to maintain.

[Blogs 23] https://martinfowler.com/bliki/TestDouble.html
[Blogs 24] https://martinfowler.com/articles/mocksArentStubs.html
[Blogs 25] https://site.mockito.org/

Over-Specified Tests

Every unit test should check the correctness of exactly one small piece of functionality and not care about anything else. Each test should have only one responsibility. Sometimes you will hear the rule of “one assert per test.” Simplify your production code, then test code so that you can test every interesting aspect of the functionality with one assert. After that, write a test for each aspect.

When you add more assertions or checks than are strictly necessary, your test becomes over-specified.

```
repo = mock(PersonRepository);
when(repo.find(1)).thenReturn(person)

personController.show(1)

assertThat(personView.getPersion())
    .isEqualTo(person); //(1)

verify(repo).find(1); //(2)
```

```
person = createPerson()

controller.show(person);

assertThat(view.find(“first-name”))
    .contains(person.firstName()); //(3)

assertThat(view.find(“edit-button”))
    .isEnabled(); //(4)
```

The first test uses a stubbed person repository that returns a certain person when called with the id **1**. It then asks a controller to display the person with id **1** and asserts that the correct person is shown in the view `//(1)`. That’s the assert that the test is interested in, but it also verifies that the stubbed method was called `//(2)`. This call to verify makes the test over-specified (Remember, never verify what you have stubbed).

The second test tests a similar functionality but checks whether the correct first name is shown `//(3)`. It also checks whether the edit button is enabled, which has nothing to do with the first name `//(4)`. This makes the test over-specified. Do not add extra asserts that have nothing to do with what the test should check.

But why is that bad? Over-specified tests often overlap, so one problem breaks many tests. And they are also more likely to **not** survive a valid refactoring. So, avoid writing over-specified tests.

[Blogs 26] https://jasonrudolph.com/blog/2008/07/01/testing-anti-patterns-overspecification/
[Blogs 27] https://osherove.com/blog/2008/7/12/over-specification-in-tests.html

Economics

Can test-driven development—writing a single *failing* test, making it *pass* and *refactoring* the code—be faster and cheaper than **not** doing it? And if so, how?

TDD can help you get feedback faster, add tests to your regression test suite, force you to take smaller steps, help you with software design and save you debugging time.

When you work on a feature, you need feedback that tells you whether you are on the right path or not. You can run your program and test it manually, but to do that, you must first implement a large chunk of code. Running and checking is also tedious and slow. With TDD, you can get **good feedback faster**. You can create a failing test and make it pass within minutes.

When you work on a large system, you need a way to know if the change that you just implemented has broken an unrelated part of the system. You need a regression test suite. Even though you will not write *all* the tests you could ever need with test-driven development, it will help you to create many **important, fast-running regression tests**.

You must design your software. Even if you do not put effort into designing your software, you will still end up with a software design—just not the one you wanted or needed. The feedback about your design that you get from doing TDD—feedback about coupling and responsibilities and abstractions—can help you **create good designs faster**.

Having many tests that run fast can **save you a lot of debugging time**. Instead of starting the program, navigating to the place you want to test, performing exactly the steps you planned and observing the results, you start a fast-running test that already performs all the necessary checks. In the worst-case scenario, you debug the fast-running test and not the complete production code, where getting to the point you want to debug will take some time.

Or, as Steve McConnell writes in *Code Complete 2:*

> *"Writing test cases before the code takes the same amount of time as writing test cases after the code, but it shortens defect-detection-debug-correction cycles."*
>
> Steve McConnell*, Code Complete 2*

What Did You Learn?

In this chapter, you learned a few things about test-driven development. I wrote about why we do unit tests, why small steps are important and why you must see a test fail. I showed you different approaches for doing TDD, what you should think about when naming tests and how test-doubles can help you create independent tests.

You also learned about the economics of test-driven development, some rules for doing it and how to not over-specify your tests.

Now, take a few minutes and think about what you have learned.

Write down your key learnings here

Refactoring

“My own interpretation of agile as doctrine is:

1. *Reduce the distance between problems and problem-solvers*
2. *Validate every step*
3. *Take smaller steps*
4. *Clean up as you go”*

Jason Yip – medium.com/@jchyip

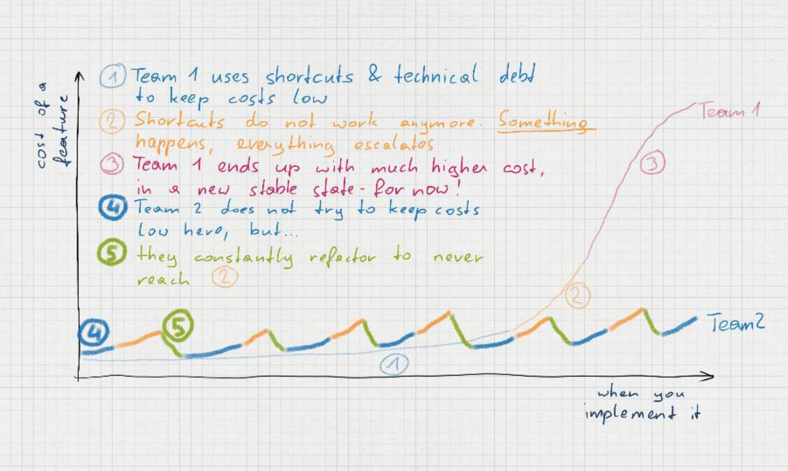
Refactoring

Refactoring means changing the structure of the code without changing its behavior. But… why would you want to do that? How does changing the code without *actually* changing it add value? Is there a business case for refactoring?

Think back to the “cost of a feature/when you implement it” curve from the beginning of the book. How expensive a certain feature will be dependent not only on *what* you implement but also on *when*. Software development becomes more expensive over time.

Sometimes you over-design. Sometimes you add code that is right at the time but gets in your way later when you learn that a different design is required. Sometimes there are features nobody uses anymore, but you must still support the code. Sometimes you add code in a hurry that will become a liability later. Sometimes people add “clever” code that nobody else can understand.

And all of that **slows you down** over time. This continues until you reach a point in time where you become very slow. Maybe a key person has quit, or you have accumulated so much technical debt that you have crossed the point where shortcuts still work.



So, how does our hypothetical Team 2 from the beginning of the book avoid this situation? **They refactor**. They clean up as they go. They often invest considerable amounts of time and money into **avoiding the risk** of becoming **much** slower at a later point.

[Blogs 28] https://www.devteams.at/2015/12/14/well-crafted-code-quality-speed-and-budget.html
[Blogs 29] https://en.wikipedia.org/wiki/Code_refactoring#Benefits

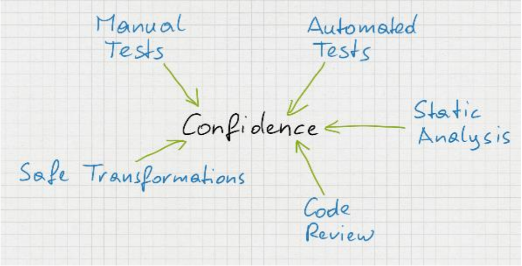
…Without Changing the Functionality

“While working on the refactoring, I also fixed that bug.”

No, you did **not**. Because what you did was not *refactoring*. If it changed the visible behavior of the code, what you did was something else. But does that matter? Does it matter what we call it?

> *“Words don’t mean what they don’t mean.”*
>
> *Dave Nicolette*

Refactoring has gotten a bad reputation in some companies and communities. “Developers are refactoring *again*? Nooo! They always *break* something when they do that. They take *ages to finish*. It is a complete waste of time!”



And that bad reputation comes from people calling code changes “refactoring” when that isn’t what they are. **Refactoring** is changing (and **improving**!) the structure of program code without changing its visible behavior in small, **safe steps**.

You can never prove the absence of bugs and you can never prove that your refactoring did not change the behavior, although some behavior changes might be less of a problem than others.

But you can be **more** or **less** confident that your refactoring did not change behavior. You should take steps to increase your confidence **before** and **while** refactoring.

Using only safe transformations and code reviews/pair programming can increase your confidence.

When you run your program and test it manually, you can be a little bit more confident that you did not break anything. You will also need a comprehensive, automated test suite and maybe even static code analysis, and you should deliberately use both of them.

Otherwise, what you do is not refactoring. It is not a safe process you can do in parallel with your normal work, which is the safe process that refactoring should be.

[Blogs 30] https://davenicolette.wordpress.com/2012/02/26/words-dont-mean-what-they-dont-mean/

Code Smells

If you have been in this industry for some time, you will have seen some bad source code. Some part of a program that was crap. Code that *had* to be re-written. But how do you decide whether a piece of code is bad? What do you look for when you search for bad code?

Before reading on,
think about it... What
makes code bad?

The phrases “code smell” and “design smell” describe an indication that *something* is not right in your code. They are heuristics. When you encounter them, when the code is “smelly”, there is often a problem. But that is not always the case. Sometimes, the smelly part is there on purpose.

A code smell is a hint the code gives you to make you think about your software design and architecture and to consider refactoring. A code smell might tell you that…

- Something about your code or design is too big. A class that has too many methods or dependencies, a method that is too long or has too many parameters, etc.
- Something about your code is unclear. A name that does not fully describe the concept, a name that is too short, a method or class that does too many things at once, etc.
- There is duplicated code or duplicated knowledge, your code is missing an abstraction or design concept, etc.
- There is something else wrong with your code or design.

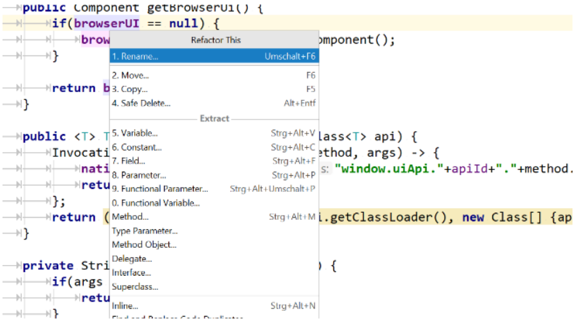
For a more complete list of code and design smells, search the terms on Wikipedia and take a look at the book *Refactoring* by Martin Fowler.

[Blogs 31] https://en.wikipedia.org/wiki/Code_smell
[Blogs 32] https://en.wikipedia.org/wiki/Design_smell
Book: *Refactoring – Improving the Design of Existing Code* by Martin Fowler

Transformations

When you refactor, you should know some transformations that are small, safe and often useful. You can do them by hand, but often, your IDE will provide them for you, making them even safer to use. When you change your code using only refactoring steps provided by your IDE, you can be very sure, but not entirely sure, that you did not change the behavior.

Some transformations you will use regularly are:



```
public Component getBrowserUI() {
    if(browserUI == null) {
        brow                                   omponent();
    }

    return b
}

public <T> T                                   lass<T> api) {
    Invocati                                   ethod, args) -> {
        nati                                   s: "window.uiApi."+apiId+"."+method.
        retu
    };
    return (                                   i.getClassLoader(), new Class[] {ap
}

private Stri                                   ) {
    if(args
        retu
    }
```

- **Extract** some code into a **variable or function**: Assigns a higher-level name to a concept in your code and can make your code more readable.
- **Extract** some code into a **function** because your current function violates the single level of abstraction.
- **Extract** some code into a **field or method parameter**: Makes your code configurable from the outside.
- **Inline** a method, function, parameter or variable because, after some other refactoring steps, you do not need it anymore.
- **Rename** a variable, parameter, function, class, etc. Often, the first name we think of for a concept is not the best name we can come up with. Change the names as you learn more about the problem and your solution.
- **Move** classes and packages/modules because your module structure is not cohesive anymore or modules are too tightly coupled.
- **Move** methods or functions to a new class or package because you found a problem with the cohesion of your code, a single responsibility principle or an open/closed principle violation. Maybe you even **extracted** the function in the previous step.

Often, you can break down even large, complex refactorings into a series of these small, safe transformations. Try hard to do that. When you can make your IDE or a refactoring tool do all the work, you can be more confident that your changes to the code did not change the functionality.

Smaller Steps

Jason Yip writes that his “agile doctrine” is:

- Reduce the distance between problems and problem-solvers
- Validate every step
- Take smaller steps
- Improve as you go

And this simple doctrine can be a great guiding principle for everything you do as an agile team. The last three are particularly important when you do refactoring, as you can change the structure of your code (with the goal of improving it) without changing any functionality.

Take smaller steps: Early in my career, I often refactored by “compile error”. I would move some code or rename something and not care whether I broke something else. Then, I would fix all the compiler errors, as well as the new compiler errors caused by my fixes. Then, I would test and fix all the regressions I introduced—or, at least, all the regressions I found!

Using this process, it would often take me days to get back to a green state. It was extremely risky. Sometimes I got to a point where continuing was extremely hard, but I had to continue because nothing worked. And I had already invested so much time that I could not revert.

Take smaller steps. Extract a method. Commit. Rename a variable. Commit. When a step gets too complicated, revert and start over again. You will only lose a few minutes when you work like that.

Validate every step: Before every step, think about what you are about to do and why you are sure you will succeed. Plan your next refactoring step.

Run your automated tests after every little step. Commit when the tests are green. Revert and start over when they are red. Use the Mikado method you learned in an earlier chapter to break down a large, complicated refactoring into small, safe steps using your tests and your version control.

Improve as you go: When you work like this, taking smaller steps and validating each step, you can always refactor on the side. You can even do large refactorings while you also work on the next features of your product.

[Blogs 33] https://medium.com/@jchyip/what-do-you-mean-when-you-say-agile-9a69201b1d9

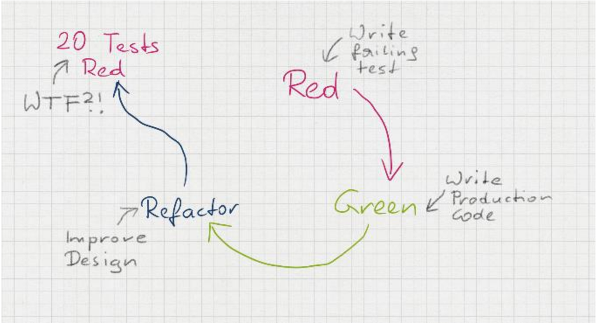
Broken Unit Tests

Sometimes a refactoring step will break some unit tests. Sometimes even a lot of them. What do you do now?

This should not happen. We are supposed to write robust tests; tests that survive refactoring. When the refactoring did not change the visible behavior of the code and the test only tests the visible behavior that is important to us, refactoring should never break a test.

But it's almost impossible to write tests like that.

Didn't some TDD expert say that tests, especially unit tests or microtests, are the safety net we need for refactoring? How does that fit together?



While it is hard to write a test that survives **any** refactoring, it is possible to write tests that survive **most** of them. While it is impossible to write a **perfectly robust** test, it is possible to write tests that are **more robust** than others.

When multiple tests turn red after a valid code change, you probably have a test suite that is too closely coupled to the internal workings of the production code:

- Maybe it created a lock on a concrete implementation/dependency
- Maybe it duplicated some code or logic from the production code
- Maybe it was too closely coupled to another test and turned red because of the same problem as the other test

When tests get in your way, are truly bad or have become obsolete, consider deleting them.

While deleting tests is a valid solution sometimes, there are often better solutions. Undo your changes, then refactor the test first so that it allows your change. After that, you can make the change. If you are already using the Mikado method, note down those two steps in your Mikado graph.

[Blogs 34] http://cdunn2001.blogspot.com/2014/04/the-evil-unit-test.html

Test Design and Refactoring

When tests get in your way as you refactor a piece of code, you will sometimes find an issue with your test design. Your tests may have created a lock on your current implementation or on your current design, i.e., how your functions, classes, modules, packages, etc. work together to achieve their goal.

While you cannot completely avoid writing tests that will get in your way later, try to learn from them and write better tests afterward. Think about why your test caused the problem.

- Was it too closely coupled to the implementation of the production code? Then think about how you could have asserted the correct behavior/result without creating close coupling.
- Did it duplicate some logic in the production code? Think about whether you can observe the correct behavior at a higher level of abstraction.
- Was it too closely coupled to the structure of the production code? Maybe you can re-structure your tests and group them by logical units, not by concrete classes or functions.
- Was it too closely coupled to the result of another test, thus becoming red for the same reason? Try to bail out with an `Assumption` early to avoid chains of failed tests.

But as is often the case with software design, those are trade-offs, and sometimes you cannot create the perfect test. Sometimes, when you improve on one axis, your tests get worse on another. The right combination of improvements will depend on your current situation, team and project.

For example, creating a test class for each production class couples the tests to the structure of your code. So, should you always avoid doing this?

Not necessarily, because having a test class per production class **and** well-named tests create very nice documentation for your production code. When you have that, you can find out what your classes implement simply by reading the test names. This documentation cannot become stale. When the code changes, thus invalidating the documentation, the test will fail.

So, learn the design trade-offs for writing tests and how they might get in your way when refactoring. Try to write tests that will not get in your way later. But if you get it wrong, don’t worry too much. You can learn something from it and change or delete the failing tests.

[Blogs 35] https://www.devteams.at/react_tdd/2019/11/18/react-tdd-2-value-and-cost-of-tests.html

Top-Down or Bottom-Up?

You encounter a piece of code where the design no longer fits the current requirements. You must do a larger refactoring. You know that you should break it down to small, safe steps, *maybe* using the Mikado method.

But still… where do you start? You could work:

- Top-down, making bigger design changes, like extracting classes and moving code first. Only then would you look inside the moved code and fix the little design problems, like bad names or unnamed concepts.
- Bottom-up, looking at the small design problems and code smells first. You would start at a particularly bad part of a function, fix it, and then work your way up to the function itself and the class or module containing it. You could extract new private functions, group them and then extract new classes or modules when you see groups that are cohesive.

Both approaches will work when you do them carefully and step by step. But which one is more cost-effective? Which one is a better use of your time? Well, it depends…

Are you trying to change code that you know very well? Do you already have an idea of how the new design should look? In this case, the **top-down** approach might be the easiest way to get from the current design/architecture to the new, desired design. But when you use this approach, breaking down the large refactoring into small, safe steps will be harder than with a bottom-up strategy. You need a structured way to break down the large change into small steps. Perhaps you could use a structuring method, like the Mikado method.

Or are you trying to understand some legacy code? Were you assigned to a new codebase that *obviously* has design problems, but there are too many to fix all of them right away? Then the **bottom-up** approach will often be the best way to get started. Begin with a nasty part of the code and clean it up a little. By refactoring the code, you learn more about it, and after learning more, you can move upwards and refactor larger parts.

Also, when you do TDD, you usually work **bottom-up** in the refactoring step. The tests are green, so you clean up a little. Then you notice a pattern or a new domain concept, so you extract a function. You see some more code that belongs to this concept, so you group it in a class. And so on…

Large-Scale Refactoring

The small refactoring that you should do daily, like in the “Refactor” phase of the TDD cycle, can be easily done in combination with the production code changes.

But what about large-scale refactoring? Like when we did not get a part of the architecture right on the first try or when our requirements changed so much that architecture that was once right now gets in our way? What about refactoring where we want to change large parts of our system?

Breaking those down into a *long* series of small, safe steps will considerably slow you down. You would *definitely* be faster if you did the refactoring of a branch in one, big step and then merged everything… right?

Before you create that branch and start with your big, single-step refactoring… Do you remember the chapter “Refactoring vs. Rewriting” (page 38)? Doing the refactoring on a specific branch is basically a re-write of a part of the system in parallel with your normal work.

You will have to support both branches for some time and merge all production changes to your refactoring branch. Otherwise, merging them once you are done will become harder with every day that passes.

You cannot easily estimate how long the refactoring will take you because you will be pessimistic about doing it in small steps (because you know the mess you are in), but you can be optimistic about doing refactoring on a branch.

You might even suffer the second system effect, where you want to improve on the existing situation **too much**, thus wasting time and money when you do a large refactoring on a branch.

For the small refactoring that you do multiple times per day, working on a branch does not make sense because you will be done very quickly. And for large refactoring efforts, working on a branch is probably more expensive and riskier than breaking down the refactoring into small, safe steps.

There might be situations where doing the refactoring in a separate branch would be faster or cheaper. But most of the time, breaking it down into smaller steps, e.g., with the Mikado method, is probably safer and more cost-effective.

What Did You Learn?

Do you sometimes refactor by compiler error? How does that feel? How do you break down large refactorings into smaller steps? How much do you use the refactoring transformations and your IDE?

Do you use tests as your safety net? Have you ever experienced that tests get in your way while you are refactoring? What do you do when that happens?

Write down how you
experienced this in the
past and what you did

Also write down what
you'll try to do
differently now...

Testing

“#TestAutomation requires both strong #testing skills and strong #programming skills. It's not enough to have a good test coded badly, and it's not enough to have good code that doesn't test well.

Don't be lazy. Draw from both disciplines. Test well. Code cleanly.”

Cassandra H. Leung – cassandrahl.com

Keep It Simple

In this chapter, you will read a lot about the different kinds of tests you will need, who will work on testing (spoiler: everyone) and when you will work on testing. But before we come to that: Please keep things simple!

“The Handbook for Writing a Testing Handbook”

I was consulting the software architecture department of a large organization with tens of projects, many more teams and several hundred developers.

One of the colleagues in my department who was responsible for testing asked me: “Can you review my testing handbook?”

He sent me a PDF with over 100 pages. And to my surprise, it was not a testing handbook. It was the “Handbook for how every team must write their team’s testing handbook”!

There were rules for everything, including how to define and enforce a unit testing strategy and how to document it, how to document the process for integration testing, UI tests and manual tests, how to decide which tests to perform, who should perform them, when to perform them and how to document this decision. And so on…

Nothing in there was factually wrong, but it just was too much. The handbook itself was too complicated. And it required every team to implement a complicated testing and documentation process. By trying to create synergies, they made life worse for everyone.

Testing is important, so do not neglect it. Spend time and effort to improve your team’s testing capabilities. Invest in testing and the people who do it. Value your tests, and value your testers even more.

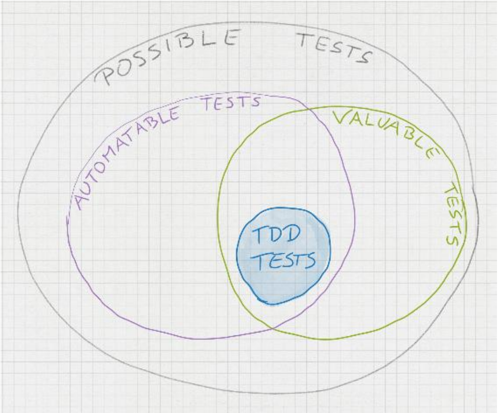
But take care to keep things simple. Like in coding and software architecture, keeping things simple is not easy. You must constantly invest time and money to keep your testing as minimal as possible while making it as comprehensive as necessary.

Also, every team and every piece of software is different. Let the people who work on a system decide and figure out how to test it.

Types of Tests

There was a whole chapter about TDD and writing unit tests/microtests earlier in this book. So, why another chapter about testing?

Well, when you do TDD, you only write very specific types of tests. You write the tests you need at that moment to drive the implementation. You rely on your personal judgment to decide which tests to write based on the value that the test can provide right then, while you are driving the production code.



With TDD, you will never write all the tests that might be useful or valuable to you. You will only write tests that show your production code is technically correct and that it behaves exactly as you, the developer, want it to behave.

However, those tests often do not check all possible edge cases. Sometimes, they cannot be easily understood by business people and do not show whether the technical behavior is correct from a business perspective.

They do not always show whether your system implements all the required features, how your system behaves under load, or how your code works together with other sub-systems when deployed in a distributed system.

And this is OK. Your TDD tests do provide value, but you will need more tests than that, so you should learn about them.

Of all the tests that you could ever perform, only some are automatable and only some are valuable (and valuable depends on your product, process and situation). As a developer, you should focus on the automatable **and** valuable tests. Learn about them and help automate them.

You must also learn about the valuable but not automatable tests. Somebody—a tester or expert or user—will have to do them, so you should help design a system where doing them is possible and easy.

Tester/Developer/BA/PO

In an agile development effort, a team—a *feature team*—should be able to bring a new feature from a vague idea all the way to production. This team needs all the skills required to get the job done.

You need front-end, back-end, database, software design and software architecture. You need graphic design and UX design. You need testing and test automation. You need business, system and domain expertise. You are responsible for your team's budget, the value created and the return on investment. You need people and process knowledge and coaching skills.

So, do you need a database expert, a front-end engineer, a professional coach, and so on? Do you need specialists?

Or should everyone know all of those things and have all of those skills? Do you need true generalists?

Neither. And both.

You do not need true specialists or true generalists, but it is also not a problem *per se* to have them.

In order to **become excellent at testing** as an agile team, a tester, a software developer, a business analyst and a product owner will have to work together. But this does not mean that a single person must have all four skills or that there must be four different people working together.

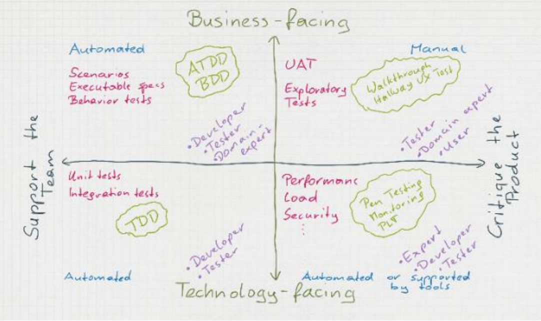
For example, there might be a great developer who also knows a lot about the business or an expert tester who is also good at software architecture and software development.

Wherever your main expertise or your main interest lies, in an agile team, it will pay off to learn other skills, too. If you are an expert business analyst, you can become even more valuable to your team if you also learn testing and development skills—especially in a situation where testing or development becomes a bottleneck.

[Blogs 36] https://www.agilealliance.org/glossary/three-amigos

Types of Tests by What They Support

Which tests do we need? And who (developers, business experts, testers, etc.) is responsible for creating them? Do they need help from other roles? How do we decide which tests we need and who works on them?



Wouldn’t it be great if we had a way to categorize the tests that we will need? One common way to categorize them is called “agile testing quadrants.”

I first learned about agile testing quadrants in the book *Agile Testing: A Practical Guide for Testers and Agile Teams* by Lisa Crispin and Janet Gregory. Although this book developed the quadrants into a comprehensive view of agile testing, the idea was originally coined by Brian Marick in a blog post that I will link at the end of this section.

From left to right, we sort the tests we perform into whether they support the team or whether they critique the product.

Tests that **support the team** are primarily there to help us do our day-to-day work, like programming, software design, architecture, documentation and communication. We write them to help us think about the problem, to prove that our software does not solve the problem yet and to show us later that the software *does* now solve the problem. We automate them to catch regressions later.

Tests that **critique the product** are primarily there to uncover missing features, reassure us that we created value, challenge our assumptions and find new ways to do things. Using these tests, we are searching for defects and opportunities that nobody has even thought about.

Along the other axis, **technology-facing** tests check whether we implemented an algorithm correctly, whether a piece of code behaves as we intended it to and whether our system integrates correctly with the outside world. This can happen on a very small scale (checking the outcome of a single function or even a single if statement) or on a larger scale (checking the security or performance of a database call, a micro-service or even the whole system).

Business-facing tests are on the other end of this spectrum. They make sense from a user's or a domain expert's point of view. They check whether our system implements a feature that the user or domain expert wants. They might even check whether our system provides value. These tests can also be on a very small scale (a single component in the checkout flow calculates the shipping cost and tax correctly) or on a larger scale (the whole system supports the checkout, shipping and payment flow).

In the picture on the previous page, you can see a graphic representation of this idea. You will learn:

- ① How to technically perform those tests by automating them, using specialized tools or testing manually.
- ② Who will (mainly) work on those tests (developers, testers, etc.). But remember, this does not mean that you need a specialized person for every one of these roles or that everyone on the team must have all these skills. Instead, split the work, work together and learn from each other.
- ③ What kinds of tests you will create (integration/integrated tests, unit tests, etc.).
- ④ Which (agile) engineering practices you can use to create and perform the tests.

The quadrants can help you identify which tests you need and who will mainly work on them.

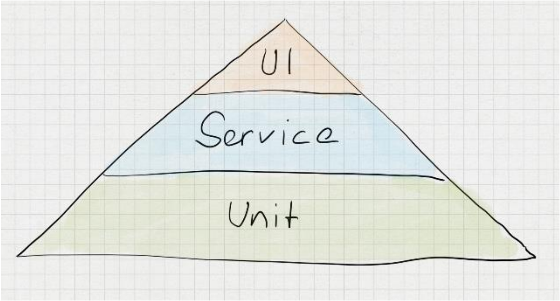
They do not define an order. Nothing in there says, "start with technology-facing tests that support the team, then continue clockwise." That would be a waterfall approach. You should work on all types of tests simultaneously.

Keep in mind that your focus might shift as your software matures. Some teams will start paying more attention to tests that support the team because they want to create a walking skeleton quickly. Other teams will first focus on tests that critique the product because they need to validate or invalidate an idea. But in both cases, the teams should not neglect the tests at the other end of the spectrum.

Book: *Agile Testing: A Practical Guide for Testers and Agile Teams* by Lisa Crispin and Janet Gregory
[Blogs 37] http://www.exampler.com/old-blog/2003/08/22/#agile-testing-project-2
[Blogs 38] https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/

Types of Tests by Entry Point

Given the different types of automated tests, how many tests of this type should you have? How much time, effort and money should you invest in creating and maintaining them?



The agile test pyramid says that you should have lots of unit tests, only some tests that test at the service level and very few that start with the UI.

Well, the most basic version of the pyramid says that. Some people have suggested that the labels within the pyramid should be different, that there should be more layers, or that the pyramid should look more like a house.

But, does that matter? Does it change the proposition of the pyramid?

Mike Cohn came up with this pyramid and described it in his book *Succeeding with Agile*. When you develop in an agile way, you must automate most of your tests and checks. The idea of the test pyramid is to divide those tests by their granularity and entry point. The tests on the lower levels are fine-grained, and they become more coarse-grained the higher up you go.

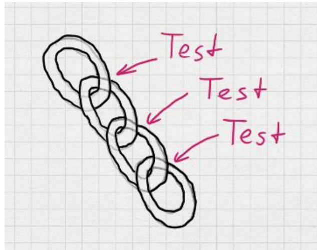
Also, tests on the lower levels provide more isolation, while tests on the higher levels provide more integration. Tests near the bottom will be cheaper to implement, easier to debug (for example, when a test that was automated through the UI fails, there are often tens, hundreds or even thousands of lines of code that might have caused the problem) and run faster.

The horizontal size of each layer tells us how many tests we should automate in that category, and how much time, effort and money we should spend on them.

We should spend most of our time at the unit level and create many tests in this category. This also implies that most of our confidence in our system will come from those tests.

But can unit tests give us confidence that our system works correctly? Can tests that run a very small piece of code or functionality in isolation tell us whether the whole system will behave correctly?

The chaining premise, which is one of the *Five Underplayed Premises of Test-Driven Development* by GeePaw Hill, tells us, "The way to test a chain is to test each individual link in that chain."



This means, when you have one piece of code, one part of the program that you can trust, you can build on that trust. You can automate a test that essentially says, "given that this lower part works correctly, the next piece of code will also work."

You can then also trust the next piece of code, and you can build on this trust to move further along the chain.

But what about things that will only break when you integrate the components? Like when your dependency injection container does not find a class that you forgot to bundle with your application? Or what about a slow connection or a network failure in a distributed system? A race condition that will only show when *both* components are executed on the same thread pool?

Well, sometimes you can even test for those problems on the lowest level. Sometimes, you'll move further up the pyramid. You will run the whole service to check whether all dependencies are configured correctly or to automate a business-facing test. And you will automate some key workflows through the UI to make sure that the whole system is up and running.

But when you find yourself automating more and more tests at the higher levels, pause and take a step back.

You have very likely found a problem with the design and software architecture of your system. You have designed and architected a system where the chaining premise does not apply.

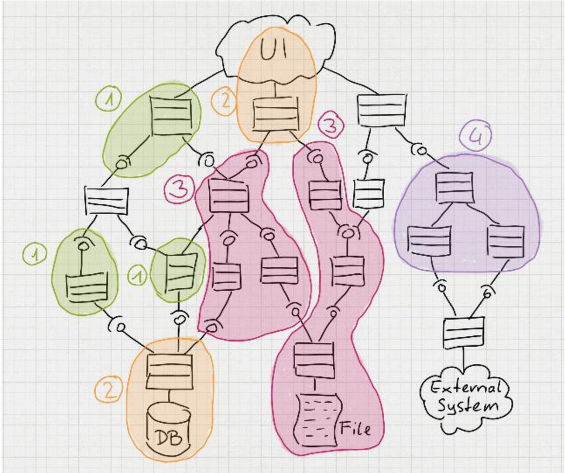
That most likely means that parts of your system are too closely coupled, that you did not separate the concerns and responsibilities correctly and that you have direct dependencies between your business concepts and the concrete, technical implementation.

Use the testing pyramid to remind yourself which level to spend most of your time on and to search for design and architecture problems when the ratios between the layers are off.

Book: *Succeeding with Agile: Software Development Using Scrum* by Mike Cohn
[Blogs 39] https://www.geepawhill.org/2018/01/18/five-underplayed-premises-of-tdd-2/
[Blogs 40] https://martinfowler.com/articles/practical-test-pyramid.html

Types of Tests by Reach

We can also classify tests, especially automated tests, by their reach. How much code do they test? How many components are involved?



This classification by reach overlaps with the agile testing pyramid, but it provides a different perspective. The focus here is not how much effort we should put into testing at a certain level, but at how those types of tests differ.

① Unit tests test a single unit of functionality. They only involve a small piece of code, often from just one class. They might test how a module uses a collaborator via an interface or how the module is used by others via an interface.

② Integration tests check whether your own system integrates with the outside world. They test a small part of your own system together with an external system (a database, a cloud service or your user interface).

③ Many developers also call the tests labeled “3” integration tests. However, they are different from the tests labeled “2”, so can we find a better name? J.B. Rainsberger calls them “integrated tests.” These are “tests whose result (pass or fail) depends on the correctness of the implementation of more than one piece of non-trivial behavior.”

④ And what about this test? Think about it… what could it be? Is it a different category? It looks like an integrated test, and it might be one. But it could also be a unit test. Maybe this test only tests a single, trivial unit of functionality. But why are there three classes involved? This might be the result of a refactoring (e.g., the developer saw a violation of the single responsibility principle and extracted a class). The point is, you cannot decide whether a test is a unit test or an integrated test simply by counting how many classes (or modules or functions) are involved.

Integrated Tests

Integrated tests are, according to J.B. Rainsberger, “Tests whose result (pass or fail) depends on the correctness of the implementation of more than one piece of non-trivial behavior.” They check whether a large(-ish) part of the application behaves as the developers expect it to behave. They often seem like a good idea. Your hypothetical co-worker or manager might say:

- “Testing at the unit level how those components work together is so hard. We need dozens of tests. We could replace all of them with a single, integrated test.”
- “Our unit tests are so small. Every one of them is almost meaningless. Can’t we write bigger, more meaningful tests?”
- “Our unit tests did not catch this defect that only occurred when the whole system is running. Better replace them with an integrated test.”

But what happens when an integrated test fails? What exactly was the problem that caused it to fail? You usually cannot tell because the test covers so much code. There could be dozens or hundreds of different problems that caused the test to fail. So, you fire up the debugger. Didn’t we agree before that writing tests would save you debugging time?

Also, integrated tests often “blink”. In other words, they sometimes fail without any reason. You re-run the test and it is green again. This often happens because they cover so much functionality. Additionally, as these tests are not repeatable, your team’s trust in their tests inevitably decreases.

And, your integrated tests probably still don’t catch all the defects. Think about it… To cover all possible combinations of problems that might occur, to even cover all possible paths through the system, the number of tests grows exponentially with the calls (i.e., layers) involved. You’ll need **more** integrated tests than unit tests to **thoroughly** test a system.

So, should you never test bigger parts of your system? Should you never run end-to-end tests?

Of course not! Testing bigger parts of the system, or even testing a system end-to-end, might be necessary for creating business-facing tests (scenarios, acceptance tests, etc.), but you should still try to do them at the lowest entry point you can find. Automate a test through the UI? Don’t do that if you can also reasonably automate it at the controller or even the service layer.

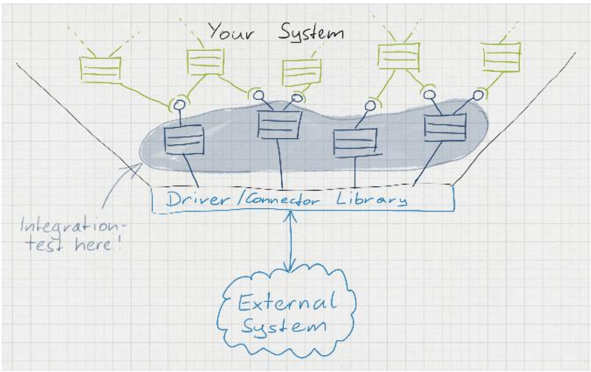
[Blogs 41] https://blog.thecodewhisperer.com/permalink/integrated-tests-are-a-scam

Integration Testing

An automated integration test checks whether your system or service works correctly with an external system or service.

Does your system use the database correctly, and can it handle different edge cases of stored data? Does your system communicate correctly with an external system, another micro-service or an enterprise service bus? Can it create and archive files correctly? Can it send and receive emails, if this is necessary for you? What happens when the network is down or slow, when messages get lost or when you receive duplicate messages?

When you use automated integration tests, you want to involve as little of your own system as possible. You want to test how your system interacts with the outside world. You do **not** want to test **everything** that happens **inside** your system before and after the interaction. Otherwise, you will simply face the same problems from the last chapter again—the problems caused by integrated tests.



When your system depends on an external system, you should have a small layer in place that abstracts the dependency. In domain-driven design (DDD), this is called an “anti-corruption layer.” In a ports-and-adapters architecture, this abstraction would be the adapters.

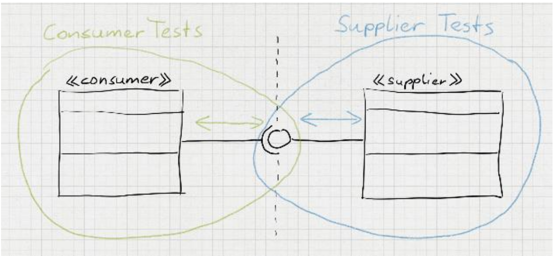
Automate your integration tests via this abstraction layer. Ideally, the integration tests should only touch a single class or function of your own system and use the external dependency.

Do not use a mock dependency or driver library! If you use a mock external dependency here, inside the test, the test does not check whether your system works well with the external system. You only test your own code and the mocks. And you must take care to ensure that your mocks are 100% in sync with the external system (even for defects!), which is almost impossible.

Book: *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric J. Evans
[Blogs 42] http://www.dossier-andreas.net/software_architecture/ports_and_adapters.html
[Blogs 43] https://wiki.c2.com/?PortsAndAdaptersArchitecture

The Two Sides of an Interface

When you have two components (functions, classes, sub-systems, micro-services) that interact via a defined interface (interface, protocol, REST service, etc.), you can test what happens at the interface from both sides.



By testing the interface from two sides, you avoid creating an integrated test and you rely on the chaining premise.

The two sides of the interface are the supplier side (the class or function or service that implements the interface and provides some functionality) and the consumer side (the class or function or service that needs the implemented functionality and accesses any supplier via a defined interface).

Supplier Tests

You can test the supplier side by calling the methods of the interface. In most situations, you should test the supplier in isolation (maybe using mock objects for the supplier's dependencies) because you only want to test the interactions through the interface.

These tests give you confidence that the supplier behaves correctly when the interface is used correctly and that it fails predictably when the interface is used incorrectly.

Sometimes, you will use the same tests for different suppliers, verifying that they all behave correctly according to the specification of the interface.

Consumer Tests

On the consumer side, you can test whether the consumer behaves correctly when they use the interface. So, you should test that the consumer can process all valid responses from the supplier and that it behaves predictably when the supplier fails.

For consumer tests, you will often mock the interface, creating a "testing supplier" to run your checks against.

The challenge you face here is to keep consumer and supplier tests synchronized. For every correct and every failing behavior of the supplier, you need a test on both sides of the interface.

Specification by Example

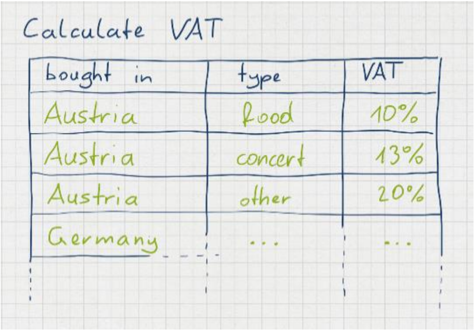
When you create business-facing tests that support your team, one of your goals should be to check whether the behavior of the system is the intended behavior. In other words, you should check whether the software works according to its specification.

On an agile team, you should not strive to create a detailed specification. That would prevent you from changing direction quickly when the world or your understanding of the world changes. A competitor created an innovative product. You've got to react **now**. A detailed specification will only slow you down.

Still, you need to define what you plan to build somehow. Agile teams usually use user stories, which are very short descriptions of the desired behavior from the perspective of a user, and acceptance criteria.

Wouldn't it be nice if we could write those acceptance criteria in such a way that they:

- Help us develop a shared understanding of what exactly we want to build next
- Help us to recognize whether we have implemented the functionality correctly
- Help us automate this correctness check to create an automated acceptance test
- Help serve as a documentation of the capabilities of our software later



Calculate VAT

bought in	type	VAT
Austria	food	10%
Austria	concert	13%
Austria	other	20%
Germany	...	...

When we use concrete examples of the desired behavior, we can achieve all four. Discussing examples will be easier than discussing abstract concepts, so we can develop a shared understanding. We can try to automate the examples directly when we check for correctness, and these examples can serve as documentation of the implemented features.

These concrete examples should be written in such a way that both developers and domain experts/users can understand them.

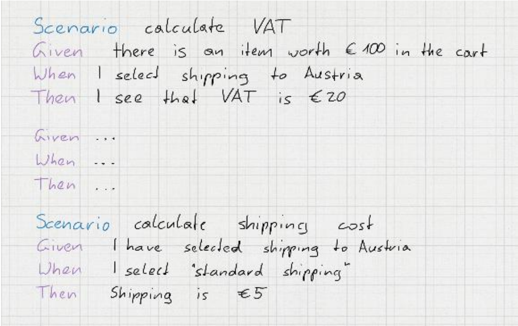
Book: *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing* by Gojko Adzic
Book: *Specification by Example: How Successful Teams Deliver the Right Software* by Gojko Adzic

Scenarios

You have captured a requirement by defining concrete examples of the desired behavior, and you have done that together, as a team. Business experts, your product owner, developers and testers were all present, and everyone who was present understands the examples you wrote down.

They understand the examples because you captured them in a format and a language that they can read. The examples are not automated tests yet. They are not code that can be executed directly by your test framework.

You used a format that your domain experts can understand, e.g., the table in the last chapter or plain English sentences.



Scenario calculate VAT
Given there is an item worth € 100 in the cart
When I select shipping to Austria
Then I see that VAT is € 20

Given ...
When ...
Then ...

Scenario calculate shipping cost
Given I have selected shipping to Austria
When I select "standard shipping"
Then Shipping is €5

And the examples use the language of the domain. They use known terms from the business domain with meanings that everyone agrees on. But what now? Shouldn't we automate those tests?

Yes, we should. And you can probably imagine some ways you could use the data from an excel file as input for your tests. That might be cumbersome. You'd have to either read the excel file directly or export it to a format that you can read more easily from your tests, but it seems possible.

But plain text? Well, that should also be possible if the text has some structure.

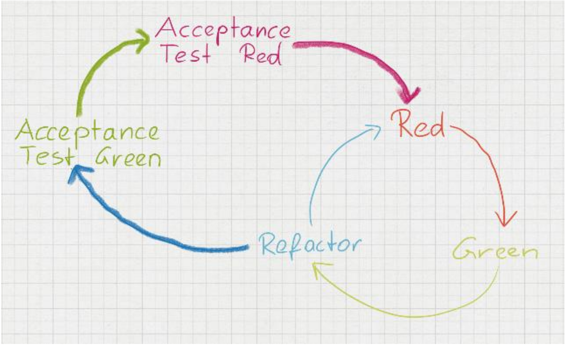
There are tools that can help you with both. *FitNesse*, for example, is optimized for specifications in tabular form. *Cucumber*, on the other hand, uses structured plain text as its input, like the sentences shown in the picture above.

So, you can **automate** your examples and scenarios **directly**. But even more important than that, you can capture the examples together. Make sure everyone understands them. Refine them to remove redundancies and ensure that they do not overlap too much. Only then should you automate them.

[Blogs 44] https://cucumber.io/docs/gherkin/reference/
[Blogs 45] http://docs.fitnesse.org/FrontPage

Acceptance-Test-Driven Development

If you have written down the specification of your desired behavior as examples, and if you have written them in such a way that you can directly automate them by only adding test automation code (but without modifying the examples themselves)…



Then why not also write that test automation code **before** the production code?

Before you implement anything else, automate an acceptance test based on the examples you created. This acceptance test will be red for now.

Then, develop the code required to make the acceptance test pass in a test-driven way. Add a unit test, make it green, refactor, repeat.

Then, at some point, you will realize that your acceptance test has turned from red to green. Now you can automate the next acceptance test, which should be red again.

Doing that creates two nested TDD cycles. But some rules are different when it comes to the outer acceptance test cycle.

In the TDD cycle, you should think about refactoring and committing your changes every time your unit tests are green. But what about the acceptance tests? Should you wait until they are green, too? Or can you refactor and commit while your acceptance tests are red?

For refactoring and committing locally, just do it. Refactor and commit while you have red acceptance tests. It will take you too long to make those green to postpone refactoring until then.

But can you push your code for your colleagues to use while an acceptance test is red? I think that often makes sense. You want to integrate your code continuously, even for unfinished features. However, whether your team does push on red acceptance tests or not should be a team decision.

When you push on red acceptance tests, they should not break your CI build when they are for features you are working on in the current iteration. However, they should break the build when they are for finished features. So, continuous integration becomes harder when you push on red.

Testing Legacy Code

> *"Legacy code is code without tests."*
>
> *Michael Feathers, Working Effectively with Legacy Code*

OK. Really? Any code without tests? On the one hand, this definition always sounded right to me. When there are no *automated* tests, we cannot reliably and repeatedly show that our code does what it is supposed to do, so we do not know what our code does.

But on the other hand, I always thought that something was lacking from this definition.

> *"Legacy code is valuable code that we feel afraid to change."*
>
> *J.B. Rainsberger*

Oh. Why are we afraid to change the code? It's because we don't know whether our change will break some part of the functionality, and a lack of good tests will contribute to that fear.

But an abundance of bad tests also contributes to that fear! When every change we make breaks dozens of tests, we become afraid of making changes. Our tests give us too many false positives.

We are afraid to change the code because we cannot be sure that we will not break anything *after* we make the change and because we don't understand it well enough to be sure we won't break anything *before* we make the change.

The reason for this is that the code is hard to understand. It is hard to find all the places where we must change the code to implement the desired functionality. It is hard to find all the places that might be affected by the change.

I will write more about the design principles that this code probably violates in the next part of this book series.

For now, it should be clear that we must refactor this code. We must improve the design without altering the functionality. And to be able to do this, we must first bring the code under test.

Book: *Working Effectively with Legacy Code* by Michael Feathers

Defects in Legacy Code

You can’t find defects in the functionality of legacy code.

OK, you can find them. But not easily. Especially when you are newly assigned to the project.

Yes, you will find many places where something is wrong. Wrong exceptions, missing logging, network connections, threads that leak, out of memory, etc. These are defects, and you should fix them eventually.

“We Know It Prints The Wrong Answers”

One of my clients did a partial re-write of an important business application. They completely re-wrote the Windows application and the mobile app, and they partially re-wrote and partially refactored the server.

It was a legacy application that had been running for over ten years. It was maintained and constantly enhanced for all that time before the re-write even started. So, the team was also trying to clean up the code and remove old, unused features.

We found a feature that generated some statistics. Even though our user base was quite big, this feature was very rarely used—only once or twice a year. We concluded that this must have been because the feature didn’t work. It produced the wrong numbers. We had a feature that didn’t work, and nobody was using it, so we removed it.

Half a year later, we got an angry call. “How can I generate the statistics with the new Windows application?”

Our product owner told them that we removed it. The person on the phone was quite angry. A small team needed the feature once a year. We tried to argue that the feature didn’t work. Their response was:

“We know it prints the wrong answers! We have a spreadsheet to correct them!”

When your code produces the “wrong” results, the answer is not that clear-cut. If the legacy code you are working on is valuable, it has probably been running for years or even decades. So, why don’t people complain about the “wrong” results?

- The wrong result might be there on purpose. Yes, it looks “obviously wrong” to every developer on the team, but the people who originally worked on the feature are all gone, so nobody knows whether or not the “obviously wrong” behavior is how our business experts wanted the system to behave.
- People have learned to live with the wrong result. They have learned to read the results in a certain way and intuitively pad them. When you fix the “defect” and give them better results, you break their mental model of the system.
- Another piece of software corrects the result. You are passing wrong information to an external system, but the system knows exactly how far off your results are and corrects them. When you fix the defects, you break the consumers of your APIs. When you re-write the legacy system, you must stay bug-compatible!
- The wrong result just doesn’t matter. It does not decrease the value of the system at all. Fixing the defect would be a complete waste of time.

Before you “fix” some “wrong” functionality, make sure to verify that this functionality is in fact a defect, preferably using multiple different sources (like key users, developers who were there when it was implemented, business experts, documentation, etc.).

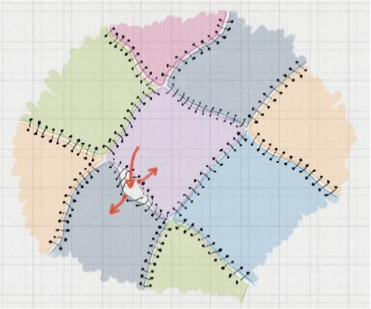
And even if you have verified that it is a defect, do not fix it while you are writing characterization tests or while you are refactoring!

[Blogs 46] https://www.davidtanzer.net/david%27s%20blog/2017/01/20/you-dont-find-bugs-in-legacy-code.html

Testing Seams

One of the first problems you often face with legacy code is that the code makes it hard to observe the internal state and to execute actions/commands in the system. It is a tightly sealed, monolithic piece of software that only exposes its state and actions via a user interface or a highly abstracted API.

Maybe you could automate all your tests using the GUI or the API, but then you would always be running the whole system, even when you only wanted to test and change a small sub-system. You would only be writing integrated tests, and those are often slow, brittle and hard to debug.



Wouldn’t it be nicer to tear the shell of the system open just a little bit, so you could look inside and execute actions?

You should try to find a testing seam—a place where you can open the system a little bit with very few changes or without changing it at all.

In the “BabystepsTimer”, a legacy code exercise I created for testing such a seam is the `timerPane`. It is a `JTextPane`; a UI component that contains almost all the relevant information from a user’s perspective, and it also allows us to execute commands inside the system. It is a testing seam because we only need to make one little adjustment to the production code: Change the visibility from `private` to package protected.

```
//...

~~private~~ static JTextPane timerPane;
private static boolean timerRunning;

//...
```

By removing this one little word, “`private`”, I can now automate the first few characterization tests. These tests will be slow and brittle, but when I have run them, I can start to refactor.

I can change the program to make it easier to test and to make the tests faster. During refactoring, I can create new and *better* testing seams, and then I can refactor my tests to use the new seams. I can now start to write better tests and refactor more, thus creating a virtuous cycle.

[Blogs 47] https://davidtanzer.net/david%27s%20blog/2017/01/13/legacy-code-refactoring-at-softwerkskammer-munich.html

Golden Master / Approval Testing

When you cannot find a testing seam to pry open the system and observe its state under the hood, you can often still ensure your changes to the code did not change the behavior using the *golden master* technique.

A golden master test runs the system under test with multiple different inputs in an automated way. It captures an output of the system, such as console messages, a log file, images of a rendered screen, or a combination of those.

On the first execution of the tests, only the results are recorded. On every subsequent execution, the tool compares the current results to the recorded results. When they are the same, your change was good. When they differ, the tool asks you to decide whether the change was bad or whether the new results are still good and should be stored as the new base for comparison.

Adding golden master tests to legacy code can be done easily and safely, but only if there is a way to execute the code in an automated way. Add some log statements and you are good to go.

But beware of using this technique. Golden master tests can be harder to maintain, slower and more brittle than tests under the hood (like unit tests, integration tests, well-written scenarios and even integrated tests using testing seams).

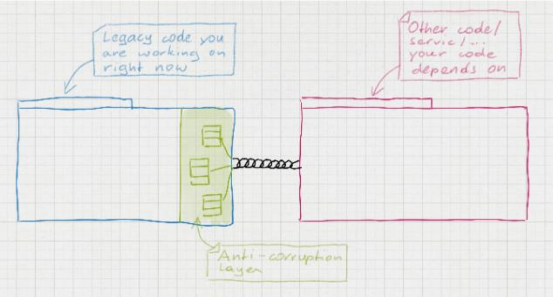
Maintaining golden master tests can become a major chore in the long run. You need results (e.g., log outputs) that are *stable*. If there is an output that occurs only occasionally, e.g., because of a cache miss, your test tool should ignore it. And getting this ignore list right so that you ignore enough, but not too much, output can be hard.

But when you get those tests right, they can also be more stable and easier to maintain. So, if you want to use this technique, make sure to invest some time into learning it properly and maintaining your tests.

[Blogs 48] https://github.com/SamirTalwar/smoke
[Blogs 49] https://craftedsw.blogspot.com/2012/11/testing-legacy-code-with-golden-master.html
[Blogs 50]https://stevenschwenke.de/whatIsTheGoldenMasterTechnique
[Blogs 51] http://coding-is-like-cooking.info/2013/09/approval-testing/

Dependencies of Legacy Code

So, you want to change a piece of legacy code. In this instance, you do not start with the whole system as that would be too big. You start with the smallest sub-system that still makes sense. Before you change it, you must write tests to ensure you do not accidentally change the visible behavior of the code.


Legacy code you are working on right now
Other code/ servic/... your code depends on
Anti-corruption layer

Now the dependencies of the code you are trying to test/change will get in your way. They will make both testing and changing harder.

The code you are working on is too closely coupled to the code on the other side. It's almost as if your code is chained to the other piece of software. You need to test something on your side, but the other side gets in your way. When you change anything on your side, something on the other side breaks.

In order to be able to bring your own code under test and to change it without having a ripple effect, without having to change larger and larger parts of the whole system, you must isolate your own code from the dependencies.

One way to do this is to duplicate your interface with the other code, module, sub-system or service, both for incoming and outgoing calls. This duplicated interface now serves as an "anti-corruption layer" for you. For the other code, it looks like it is still interacting with your old legacy code, but you can refactor freely behind this layer.

This anti-corruption layer allows you to treat the other side as if it were an external system, even though it might just be some classes or functions in the same process.

After creating that layer, you can use the techniques I described in the chapters "Types of Tests by Reach" (page 97) and "Testing Seams" (page 107) to proceed. Then, create integration tests for your anti-corruption layer and the other side.

Write integrated tests (but without the other side) behind that layer, then refactor the code to make it more testable. Now you should be able to write better, faster unit tests, which will enable you to do even more and even better refactoring, remove the integrated tests, and so on.

Mock Objects and Legacy Code

Those dependencies of our legacy code and the “other side” that our code seems to be chained to… Why should we write an anti-corruption layer and integration tests? Why should we treat it like an external system? Why don’t we just add stubs and mocks for it and be done?

Because using mock objects to decouple legacy code from its dependencies is a slippery slope. Yes, it is sometimes the only solution to your problem, but it often leads us into the mock objects trap, which is a situation where you use mock objects in such a way that they make your life harder, not easier.

While mock objects allow you to test a piece of code without knowing the exact dependencies, they also create a lock on those dependencies. When, in your test, you verify that an interaction has taken place, the test makes it harder for you to refactor and use a *better* interaction. When you stub all dependencies for your tests, your tests make it harder for you to *clean up* those dependencies.

All of that is not a big problem when your code is relatively clean. When your system is loosely coupled, has good abstractions in place and adheres to the dependency inversion principle, the single responsibility principle and other design principles, your tests and mock objects will probably not be a big impediment to refactoring.

When your code and design are relatively clean and you spot a minor problem that you want to refactor, you will probably also only change a small number of stubs and mocks.

But in legacy code with “chained” dependencies, your code and design are not relatively clean. You do not care about minor design problems yet because you must first get the big problems out of the way.

When you use stubs and mock objects to sweep those big problems under the rug at first, cleaning them up later will be even harder. Your tests and your mocks will prevent you from refactoring, which is exactly the opposite of what you wanted to achieve.

Mock objects have their place and can work well in certain situations, but testing legacy code is not one of those situations.

[Blogs 52] https://www.davidtanzer.net/david's%20blog/2016/06/03/the-mock-objects-trap.html

The *Other* Automation Trade-off

“We decide what to automate on a case-by-case basis,” a manager at a past client once told me. “If we expect the time to automate to be shorter than the time to do the task manually for one year, we automate. Otherwise, we postpone the automation.”

This makes sense, right? Automation is a trade-off. Automating tasks takes time, and if that time is longer than doing the task manually, it does not make sense to automate. At least, it *seems to make sense* at first, but…

There is another automation trade-off. The case-by-case decision at this past client meant that they optimized small parts of the system (the cost of each automation task). This does not necessarily mean that they optimized the whole system (the benefit of having automation).

In this case, it meant that some tests and some setup tasks were not automated. “We only deploy to production every two to three months, and we only run those tasks during production deploys. So, doing those tests and setup tasks by hand is cheaper.”

But this case-by-case optimization created a system that was not working well overall. The manual steps and tests prevented them from optimizing other parts of the system. Manual installation steps made hardware and software updates slow and expensive. And because of that, they had to support several clients with different and older hardware and software versions on the server-side, which was expensive.

The manual tests and verification steps prevented them from releasing more often. This meant they got slow feedback and often wasted money on features that their customers did not really need. They prevented them from implementing a continuous deployment pipeline, which would have helped them to automate more.

The company did save some money by doing fewer manual tasks, but because they decided not to automate *everything*, the company was deadlocked and unable to gain the true benefit of automation: Faster feedback and lower risk.

The *other* automation trade-off is: When you do not automate *everything*, you will not get the true benefits of automation. Instead, you pay a large part of the cost of automation and you still do a lot of manual work.

Wait… Automate *Everything*?!

Should you automate exploratory testing? Programming? Gathering requirements? Planning iterations and releases? Should you automate everything?

Of course not. There might come a time when we can automate those tasks. But for now, that’s impossible, so it would be a waste of time to work on it.

You should strive to automate all the repetitive tasks that your team performs. And while I want you to **automate all those repetitive tasks** (maybe even programming and gathering requirements) in the long run, **use common sense** when deciding what to automate first and how much time, money and resources to spend on the automation.

Some of those tasks will be easy to automate, but for others, automation will be difficult or even impossible to do now (because of pre-conditions to the automation). For some tasks, automating them will bring significant benefits to your team. For other tasks, the benefits will be minor. Automating a task can sometimes benefit you immediately, and other times, you will only benefit in the long run or when you finish automating a group of tasks.

Start automating tasks that **immediately provide a significant benefit** and where **automating** the task **is easy**. Then move on to the more challenging jobs.

And while you should thrive to automate *everything*, you should also know that you’ll never reach that goal.

What Did You Learn?

As a software developer on an agile team, you write automated tests, work together with testers to automate higher-level tests, work with your product owner and business people to define executable specifications and automate pipelines that make sure only working code gets to run in your production environment.

You must sometimes test code that is almost untestable (legacy code) and you will design tests on different levels and with different purposes.

How do you make sure that you check that your program is working at the right level? That you create both business-facing and technology-facing tests? That your tests not only support your own team, but also help you build a better product?

Write down some
interesting things you
have learned and
some further ideas

About You

‘‘Please don’t think career success means happiness. Please don’t put off being happy for one day years down the road. Please don’t take what you have now in the non-career world for granted.

Work on being happy with what you have, right now. I hope someone reading takes that to heart.’’

Stephanie Hurlburt – @sehurlburt

Life Is Not a Competition

Life is not a race. You do not win when you finish first. You do not win when you finish last.

You do not win when you finish with the most money in the bank or the biggest house. Yes, it would be nice to leave your relatives some legacy they can build upon, but you do not win when this is your only goal in life.

Life is more like a hike. It *is* nice when you arrive at your destination, but not *because* you reached the destination. It is nice because you can look back and say, "That was beautiful. Now I need some rest." The way may have been exhausting. There were times when you wanted to give up, when you were scared or when you needed help. But you made it. Now, it is time to rest.

Reaching the destination is not the goal. We hike to have a good time while we are getting there and to sometimes pause and enjoy the view. Often, we do it together, because doing nice things together is even more worthwhile.

"Why the **** are you writing about that in a book about writing software," you ask?

Because looking after yourself is important. It will increase your quality of life. No matter how much you like your work, when you live only for your work, you will miss out. And you will probably get health problems later.

There are also some abusive companies out there. Companies that demand overtime, companies where you never have enough time to learn or practice or even take a holiday. They promise a big exit or a promotion in return, but even if you get that promotion or exit, is it *really* worth working yourself to death for years?

Taking care of yourself will not only improve your quality of life, but it can also help you become a better programmer. That is why I have included this chapter in this book.

Rewarding, Challenging, Exhausting

Developing software—writing code, running it, debugging, solving problems—can be extremely rewarding, even though it is also often challenging. Sometimes it does not even feel like work at all. We *want* to work until late at night, until the problem is solved. We *do not want* to take a break until this one little feature is finished, and we're always sure that we are almost done.

For some of us, software development is also a hobby. But for some, it is not. For some, it is just a job to earn the money they need for other things, maybe for other hobbies. Hopefully, it is a job that they enjoy, but it is still just a job.

And both are OK. Both **must** be OK. You will sometimes read on the internet that people who do not learn during their free time, who don't work on open source and have a private GitHub account and who do not keep up with all the latest technologies without getting paid will never make it in this industry.

Do not believe it. There are companies that reward those ideas, but this is abusive behavior. Our industry should not be like that, and there are companies out there that are different. We should all work together to change the rest.

Yes, it **is important** to learn new technologies, to keep up with innovations and to always expand your knowledge. Your employer should be interested in you learning everything you need to do your job *and* a little bit more. But there are probably even more things you want to learn to stay employable, and you should learn those on your own time.

However, you must take care of yourself first. Even for those of us who consider creating software their hobby, it can still be very exhausting. There are companies and policies and regulations that can drain your mental energy even more. The co-worker who always criticizes you. The policy that prevents you from getting anything done. The company structure that prevents decisions from being made because you always need people from four departments to decide anything. The micro-manager or the manager who does not give you nearly enough guidance.

And there are worse things going on, like mobbing or sexual harassment. It is no wonder that people burn out or quit our industry for good. So, take care of yourself. No matter how much you enjoy programming, always remember that your *current* job is **just a job**.

Slow Down

It can be tempting to spend ten or more hours at the computer every day designing and writing software. Especially when you are passionate about software or when programming is also your hobby. On the other hand, there are some companies that demand long hours.

But this is not sustainable. You cannot run a marathon by sprinting all the time.

Working in this way is not good for you, not good for your professional development and not good for your team.

Working too much is not good for your physical and mental health. If you don't get enough sleep and distractions, you will begin to learn slower, so working too much is not good for your professional development either.

What makes it even worse for me is that when I work too much, I cannot even come down and relax anymore. I feel restless and nervous.

Yes, your current project or your current employer is important, but you are here for the long run. Most of us spend forty years or more at work. Spend those years in a way that also allows you to stay productive, learn **and** have fun in the next year, in ten years, and at the end of your career.

Also, when you do not rest enough and do not take enough breaks, you will make more mistakes and your productivity will suffer. You will not only become slower yourself, but you will also slow down your team members by making bad decisions.

Agile teams know that they should write code at a Sustainable Pace (page 16). Agile companies know that their teams need some slack to improve the code and the design, to research new ideas, to innovate and to react to emergencies. They know that putting less work into the queue will make the whole company go faster.

Developers on agile teams should know that this is true for themselves, too. Do less. Take care of yourself. Slow down.

Health

You only have one body. Look after it and treat it well. Especially since your work comes with some health risks. Yes, there are worse jobs when it comes to health, but as software developers, we sit too much, we do not move enough and the movements we make are too repetitive.

And your body does not like that. Many of us suffer **chronic pain**. I often feel pain in my neck, my lower back, and sometimes in my thumbs when I use my phone and mouse too much.

So, stand up a lot. Get a standing desk and incorporate movement into your day. Get good tools like an ergonomic keyboard and mouse, even though those are expensive. Learn keyboard shortcuts.

Some of us have problems with **getting enough sleep**. Being adequately rested is extremely important. You make fewer mistakes and learn faster when you are relaxed and awake. Relaxation exercises and meditation can help with this.

Sometimes, there is not enough time to prepare healthy, homemade food, so we eat fast food or ready-made meals. But since we also do not move nearly enough and work indoors, this **food can make us sick** and overweight as we do not burn a lot of calories during work.

We need vitamins to stay healthy. So, try to get some light, stay hydrated, and eat fresh food as often as you can. This may also help with poor sleeping and with chronic pain.

But even if you get a standing desk, move during your breaks and eat healthy, we still **don't move enough** in our job. If you are into sports, great! But even if you are not, try to incorporate movement and exercise into your daily routine. I bike from home to the train station and, if I can, from the other train station to my clients. If not, I try to walk. It's only 10-30 minutes of movement per day, but it is better than nothing.

[Blogs 11] https://www.davidtanzer.net/david's%20blog/2012/04/02/cheap-plastic-drills.html

Mental Health

I am not an expert when it comes to mental health. Fortunately, I have never had to face any severe issues myself.

But I have had that feeling of “This is too much. I have too much to do and too many responsibilities. I’ve taken on too many commitments. I cannot do this anymore. **I cannot do anything** anymore.”

I think I did get *pretty* close to my own limit more than once in the past. But so far, I have always been able to reduce the scope, cancel commitments and/or take a break.

But others are not that fortunate. I know that many in our industry fight with overwork, depression and burn out. Some have lost all their excitement for a job they once loved.

Stress can drain your energy quickly. When there is always too much to do, when we can never take a break and reflect, learn or come down, we can lose our ability to relax.

Some of us feel helpless. Feeling micro-managed at work or that your work and your ideas do not have any impact (maybe because your organization is too bureaucratic or too slow) can lead you down this dangerous path of getting close to your limit.

The opposite, too many commitments, can be a problem, too. When we find our job rewarding and exciting, we can say “yes” to too many things. At first, you were excited about all those opportunities. But when you take on too much, none of them are exciting anymore. They are just. Too. Much. Work. Learn to say “no” more often, and even cancel commitments you have already made. *Your* health is more important than any promise you made in the past.

And then there is often not enough time for your friends and family because of all the commitments and the stress. This can make us feel guilty and *add to our stress*.

Some in our industry are even harassed and discriminated against during their job, while searching for jobs and during the application and interview process. Some of them leave our industry forever.

When you are getting close to your limit, stop. Take a break. Try to break the vicious circle.

Seek help and help others.

Seek Help

When you want to learn something, seek help. When you need some feedback, seek help. When you are facing problems, when you want to hear a second opinion, when you are feeling sick or down, seek help.

Grow your professional network so you know people who can teach, mentor or support you when you have questions about software design.

And do not hesitate to ask people for help, even strangers. Many “internet-famous” people are very kind and helpful. Some have standing invitations allowing you to ask for help. If they have that, do it.

Sometimes, you won’t get an answer when you ask a stranger for a favor. Maybe they were too busy to answer, so you might want to ping them again with a very short message. And if they still don’t answer, move on. They probably don’t have the time or energy to reply right now.

If you get a rude or condescending reply, also move on. Do not respond. Just block that person on social media. Life is too short to deal with people like that.

Keep your friends and family close. Talk to them regularly when you need to discuss personal issues.

Seek professional help when you feel ill, both physically and mentally. Do not experiment or try to push through too much. Sometimes, it is better to let a trained professional take over.

Also, help others if you see that they are in need, but only after confirming that they:

- **Actually** need help
- Want **you** to help them

Only do so when you feel you have the energy to help. Do not help others when you yourself feel down and are in need of help. And be aware of people who always need you and never give back.

But do help others and do seek help when you feel that you need it.

What Did You Learn?

You are in a marathon, not a sprint. Take care of your body and your mind. What can you do to make sure that you are productive and having fun at work in the next year and in ten years?

What I already do

What I plan to change

The book is almost over now; there's only one chapter missing. If you like it, please do not forget to **recommend it to your friends**, followers and colleagues.

But do not send them the PDF. The e-book is free for now, so anyone can download it from the book's page in exchange for an email address:

https://www.quickglance.at/agile_developers_practices.html

You can use the following links to directly recommend the book:

- Recommend the book with a Tweet
- Recommend the book on Facebook
- Recommend the book on LinkedIn

Is That All?

“What if the most valuable human skill in the future of work is the capacity to ask the (contextually) right questions?”

Esko Kilpi – @EskoKilpi

That was a lot. And it still was not everything. Not even close. I am planning to write a second book, and maybe a third, and even then, I will not have written everything.

I hope you now have a lot more ideas to research. What were the most important topics for you? Where did you already know a lot? Where do you have some room for improvement?

How will you start improving?

How will you start?
Write down your
ideas, plan & impediments

Further Reading

Here you can find all the blog posts and books that I have linked in the previous chapters, in the order in which they appeared.

Blog Posts

1. https://www.devteams.at/2015/12/14/well-crafted-code-quality-speed-and-budget.html "Well Crafted Code, Quality, Speed and Budget" by David Tanzer
2. https://people.engr.ncsu.edu/gjin2/Classes/591/Spring2017/case-tdd-b.pdf "Realizing quality improvement through TDD" by Nachiappan Nagappan et al.
3. https://bowperson.com/wp-content/uploads/2014/11/SixTrumpsArticle220101.pdf "Six Trumps: The Brain Science That Makes Training Stick" by Sharon L. Bowman
4. https://hackernoon.com/software-complexity-naming-6e02e7e6c8cb "Software Complexity: Naming" by Alexandre Oliveira
5. https://tommikaikkonen.github.io/impure-to-pure/ "From Impure to Pure Code" by Tommi Kaikkonen
6. https://www.davidtanzer.net/david%27s%20blog/legacy_code/2018/05/21/legacy-code-mikado-method.html "Legacy Code: The Mikado Method" by David Tanzer
7. https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/ "Things You Should Never Do, Part I" by Joel Spolsky
8. https://stackoverflow.com/questions/2281619/tips-for-avoiding-second-system-syndrome Stack Overflow question "Tips for avoiding second system syndrome" asked by user Wally Lawless
9. https://www.geepawhill.org/2018/11/14/the-gold-of-microtests-the-intro/ "The Gold Of Microtests: The Intro" by GeePaw Hill
10. https://www.geepawhill.org/2018/04/16/the-technical-meaning-of-microtest/ "The Technical Meaning Of Microtest" by GeePaw Hill
11. https://www.davidtanzer.net/david's%20blog/2012/04/02/cheap-plastic-drills.html "Cheap plastic drills" by David Tanzer
12. https://www.davidtanzer.net/david%27s%20blog/2012/08/21/kitchen-knives-and-other-tools.html "Kitchen knives (and other tools)" by David Tanzer
13. https://en.wikipedia.org/wiki/Touch_typing Wikipedia page "Touch typing"

14. https://en.wikipedia.org/wiki/Dvorak_keyboard_layout Wikipedia page "Dvorak keyboard layout"
15. https://www.mercurial-scm.org/wiki/UnderstandingMercurial Tutorial "Understanding Mercurial"
16. https://en.wikipedia.org/wiki/Continuous_integration#Automate_the_build Wikipedia page "Continuous integration"
17. http://feelings-erased.blogspot.com/2013/03/the-two-main-techniques-in-test-driven.html "The Two Main Techniques in Test-Driven development, part 2" by Grzegorz Gałęzowski
18. https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html "The Cycles of TDD" by Robert C. Martin
19. http://pitest.org/ Tool: "Pitest"
20. https://www.devteams.at/red_green/2019/03/18/red_green_part_3_why_red.html "TDD: Red-Green Part 3: Why Red" by David Tanzer
21. https://8thlight.com/blog/georgina-mcfadyen/2016/06/27/inside-out-tdd-vs-outside-in.html "TDD - From the Inside Out or the Outside In?" by "Georgina McFadyen"
22. https://medium.com/@erik.sacre/clean-architecture-through-outside-in-tdd-64a31de17ccf "Clean architecture through Outside-in TDD" by Erik Sacré
23. https://martinfowler.com/bliki/TestDouble.html "TestDouble" by Martin Fowler
24. https://martinfowler.com/articles/mocksArentStubs.html "Mocks Aren't Stubs" by Martin Fowler
25. https://site.mockito.org/ Tool: "mockito"
26. https://jasonrudolph.com/blog/2008/07/01/testing-anti-patterns-overspecification/ "Testing anti-patterns: Overspecification" by Jason Rudolph
27. https://osherove.com/blog/2008/7/12/over-specification-in-tests.html "Over Specification in Tests" by Roy Osherove
28. https://www.devteams.at/2015/12/14/well-crafted-code-quality-speed-and-budget.html "Well Crafted Code, Quality, Speed and Budget" by David Tanzer
29. https://en.wikipedia.org/wiki/Code_refactoring#Benefits Wikipedia page "Code refactoring"
30. https://davenicolette.wordpress.com/2012/02/26/words-dont-mean-what-they-dont-mean/ "Words don't mean what they don't mean" by Dave Nicolette

31. https://en.wikipedia.org/wiki/Code_smell Wikipedia page: "Code smell"
32. https://en.wikipedia.org/wiki/Design_smell Wikipedia page: "Design smell"
33. https://medium.com/@jchyip/what-do-you-mean-when-you-say-agile-9a69201b1d9 "What do you mean when you say Agile?" by Jason Yip
34. http://cdunn2001.blogspot.com/2014/04/the-evil-unit-test.html "The Evil Unit Test" originally by Alberto Gutierrez
35. https://www.devteams.at/react_tdd/2019/11/18/react-tdd-2-value-and-cost-of-tests.html "React TDD 2: Value and Cost of Tests" by David Tanzer
36. https://www.agilealliance.org/glossary/three-amigos Glossary: "Three Amigos"
37. http://www.exampler.com/old-blog/2003/08/22/#agile-testing-project-2 "Agile testing directions: tests and examples" by Brian Marick
38. https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/ "Using the Agile Testing Quadrants" by Lisa Crispin
39. https://www.geepawhill.org/2018/01/18/five-underplayed-premises-of-tdd-2/ "Five Underplayed Premises Of TDD | Video" by GeePaw Hill
40. https://martinfowler.com/articles/practical-test-pyramid.html "The Practical Test Pyramid" by Ham Vocke
41. https://blog.thecodewhisperer.com/permalink/integrated-tests-are-a-scam "Integrated Tests Are A Scam" by J.B. Rainsberger
42. http://www.dossier-andreas.net/software_architecture/ports_and_adapters.html "Ports-And-Adapters / Hexagonal Architecture" by Unknown
43. https://wiki.c2.com/?PortsAndAdaptersArchitecture C2 Wiki page: "Ports And Adapters Architecture"
44. https://cucumber.io/docs/gherkin/reference/ Tool: "Cucumber", "Gherkin Reference"
45. http://docs.fitnesse.org/FrontPage Tool: "FitNesse"
46. https://www.davidtanzer.net/david%27s%20blog/2017/01/20/you-dont-find-bugs-in-legacy-code.html "You don't Find Bugs in Legacy Code" by David Tanzer
47. https://davidtanzer.net/david%27s%20blog/2017/01/13/legacy-code-refactoring-at-softwerkskammer-munich.html "Legacy Code Refactoring at Softwerkskammer Munich" by David Tanzer
48. https://github.com/SamirTalwar/smoke Tool: "Smoke"

49. https://craftedsw.blogspot.com/2012/11/testing-legacy-code-with-golden-master.html "Testing legacy code with Golden Master" by Sandro Mancuso
50. https://stevenschwenke.de/whatIsTheGoldenMasterTechnique "What is the Golden Master Technique?" by Steven Schwenke
51. http://coding-is-like-cooking.info/2013/09/approval-testing/ "Approval Testing" by Emily Bache
52. https://www.davidtanzer.net/david's%20blog/2016/06/03/the-mock-objects-trap.html "The Mock Objects Trap" by David Tanzer

Books

- Book: *Continuous Delivery* by Jez Humble and David Farley
- Book: *The Nature of Software Development* by Ron Jeffries
- Book: *The Passionate Programmer* by Chad Fowler
- Book: *Code Complete 2* by Steve McConnell
- Book: *Clean Code* by Robert C. Martin
- Book: *Working Effectively with Legacy Code* by Michael Feathers
- Book: *The Mikado Method* by Ola Ellnestam and Daniel Brolund
- Book: *The Pragmatic Programmer* by David Thomas and Andrew Hunt
- Book: *The Git Book* by Scott Chacon and Ben Straub - Free book at https://git-scm.com/book/en/v2
- Book: *Refactoring – Improving the Design of Existing Code* by Martin Fowler
- Book: *Agile Testing: A Practical Guide for Testers and Agile Teams* by Lisa Crispin and Janet Gregory
- Book: *Succeeding with Agile: Software Development Using Scrum* by Mike Cohn
- Book: *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric J. Evans
- Book: *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing* by Gojko Adzic
- Book: *Specification by Example: How Successful Teams Deliver the Right Software* by Gojko Adzic