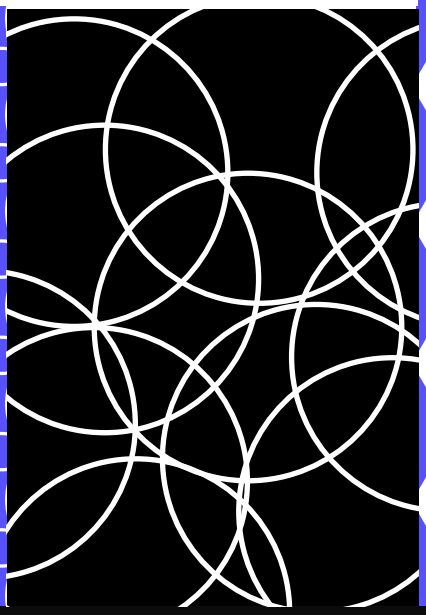
COMMON PATTERNS THAT MAKE TDD HARDER

An essay from practitioners



Matheus Marabesi

codurance

COMMON PATTERNS THAT MAKE TDD HARDER

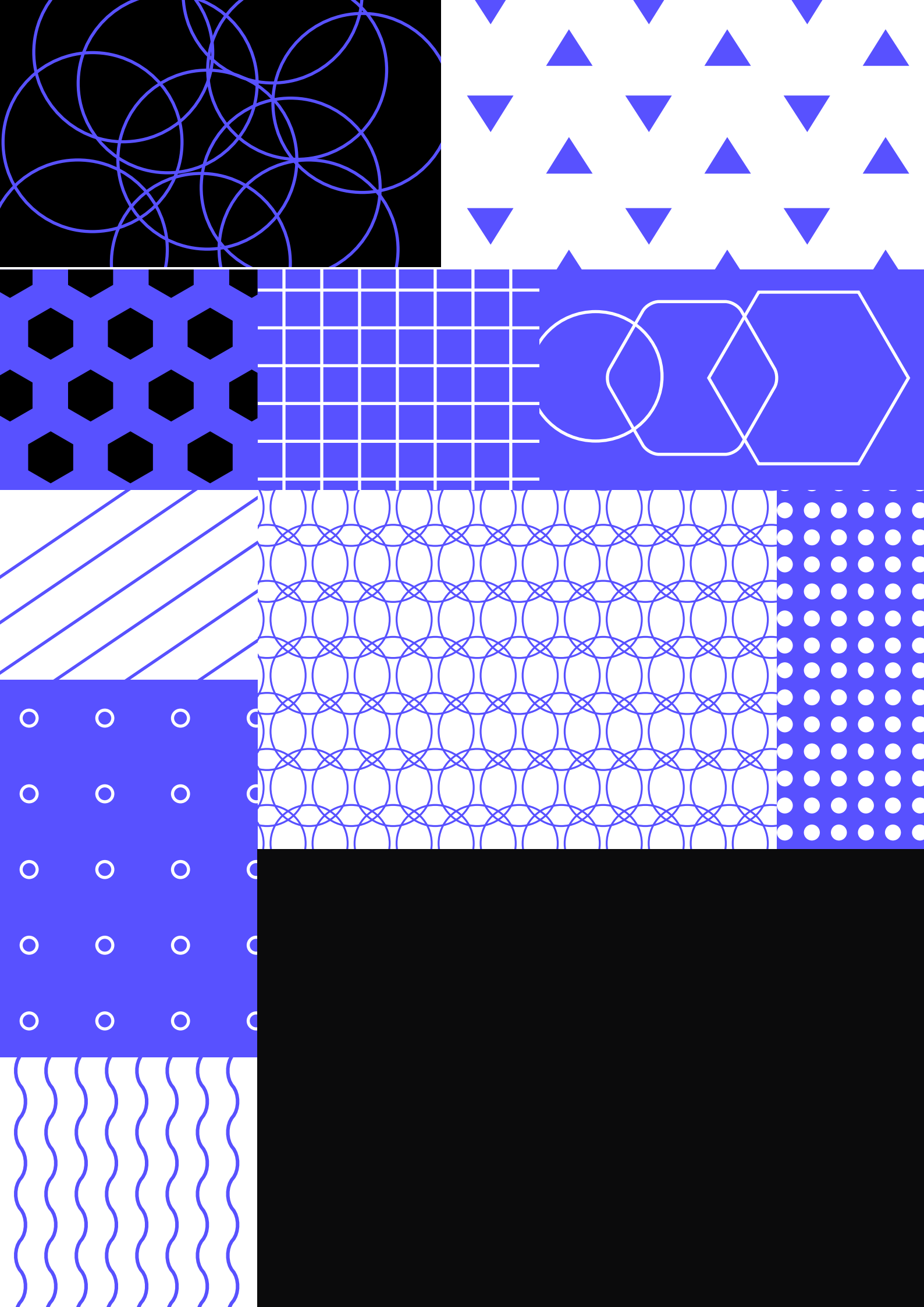
An essay from practitioners

Matheus Marabesi
2023

Every programmer knows they should write tests for their code. Few do.

Kent Beck and Erich Gamma

TABLE OF CONTENTS



Foreword

It has been almost 20 years since Kent Beck presented his “rediscovery” of Test Driven Development (or TDD as it is more commonly known) and published his subsequent book, Test Driven Development: By example.

My journey began while reading Growing Object-Orientated Software Guided By Tests by Nat Pryce and Steve Freeman. I could not help but think that having this book that Matheus Marabesi put together back then would have been very useful. Many of the patterns Matheus presents in this book I have experienced myself through working with different software development teams over the years.

Since Kent Beck’s “rediscovery,” there have been many publications, conferences, podcasts, and other media that extol the virtues of TDD. However, far less focus has been made on testing patterns that can slow down development over time and inhibit wider adoption. The patterns presented in this book by Matheus, and the accompanying code examples, can be used as guardrails for implementing TDD and maintaining a suite of automated tests. In that regard, this book is a useful resource to ensure you can maximize the benefits that TDD can bring.

Matt Belcher

Acknowledgments

Putting together a book like this was challenging. It involved gathering information from open-source projects and practitioners who work with test-driven development daily. Throughout the life of this work, many people have contributed to its content by providing feedback and sharing their time and wiliness to help.

I would like to start by especially thanking Codurance for making the space and the effort to put together an environment that fosters continuous learning and sharing back with the community.

This book is also an outcome of a series of talks Codurance and I gave regarding TDD anti-patterns which ignite this more formal approach. A big thanks to Helena Abellán for all her hard work behind the scenes, gathering all the information needed, and making the video series a success.

A special thanks to all those who participated in the video series and collaborated: Cameron Raw, Giulio Perrone, Juan Pablo Blanco, Javier Martínez Alcantara, Sofia Carballo, Ignacio Saporitti, and Pablo Díaz.

Matheus Marabesi
March 2023

Contact Us

If you found any issue with the content or want to reach out to provide feedback, please follow this google forms link.

License

Preface

The idea of writing test code before the production code (or as a some might refer to it "real code")[1] brings some mixed feelings for those experienced practitioners. This is brought about by the feeling that "it does not make sense to write code to test code." In extreme scenarios, practitioners might think they are clever enough or experienced enough not to waste time writing test code.

Once this first stage passes, slowly the buy-in happens. It isn't easy to change the way we work. It requires effort and the willingness to experiment with new ways of working. In the end, we have all been there. Writing some code and seeing it worked the first time it ran was considered a success.

After all the negation for a test-first approach and understanding that it might help in the development flow (by preventing regressions), the final stage is reached: accepting that writing tests help in the long run as it builds a safety net for practitioners to create software.

That acceptance is just the first step practitioners will take on the road to mastering TDD, a challenge in itself.

On that path, practitioners will face many scenarios in which testing is required and will face many challenges, trade-offs and potential pitfalls along the way.

For example, you may want to ensure that a suite of tests continues to run as fast as possible[2] to ensure a short feedback loop. Whilst, on the other hand, you may also want to avoid lengthy tests that aim to test everything in a single test case.

In this sense, at Codurance we are trying to give a new way of looking at different anti-patterns that practitioners might face when writing testable code (or trying to). This book is the result of that initiative.

---

1 Production code is used to point to the code that will be executed when the users interact with the application; practitioners could also refer to that as the "real" code.

2 Running 1000 tests in 1s by @marvinhagemeist: https://marvinh.dev/blog/running-1000-test-in-1s.

Who Should Read This Book?

This book is aimed at practitioners who want to explore particular testing patterns that, when followed, can make TDD harder and, consequently, perceive that the tests are not adding value to the development cycle. Throughout the book, an attempt is made to expose the pain points that can lead to practitioners feeling this way, with examples and how to avoid those when writing test code or production code. As we will see, both are highly connected.

It is likely that you have some experience in writing code in a test-first manner but along the way, have experienced issues with the following:

- Not perceiving the value of the tests
- Spending more time debugging tests than using them as a guide
- Waiting too long to receive feedback from tests
- Disparity between the test feedback and production code

However, even if you have limited prior experience in writing test code first, you will still find something valuable to take from this book. Hopefully, you will find patterns to avoid when writing test code in the future.

Whilst this book is not necessarily intended for practitioners who already know the TDD anti-patterns, such as how to approach difficult scenarios to progress while developing code-guided by tests, it may still be of interest to you.

Effort has been made to make the content as beginner friendly as possible, however, the content used shows different patterns, programming languages, and frameworks. Each chapter offers extra resources to go deeper into a particular subject if needed.

In order to get the most out of this book, it is recommended that you feel comfortable and understand the concepts and methods being presented.

Take time, if needed, to refresh the following topics, which will be covered and / or discussed in the book:

- Object-Oriented Programming
- Web development (HTML, JavaScript, and the respective frameworks such as VueJs and ReactJs)
- Testing frameworks such as Junit, PHPUnit, Jest

Last but not least, this book is best viewed as expert recommendations as opposed to definitive “best practices”.[3] This makes it easier to avoid the

---

3 Since “best practices” is context-dependent and usually, the context is difficult to share when using the term “best practices” it can be more of a hindrance than a help. In its place term “sensible defaults,” depicted in the Technology Podcast by Thoughtworks: Starting with sensible default practices available at https://www.thoughtworks.com/en-es/insights/podcasts/technology-podcasts/sensible-defaults.

"one solution fits all" ideology. It is also a much closer description of our recommendations based on our experiences practicing TDD in the industry combined with the survey we conducted to obtain insights from other real-world practitioners.

The formatted content here attempts to contribute and share back to the wider community. You may have experienced different scenarios that aren't included in this book, which we accept. As stated in the previous paragraph, this book is a culmination of our experiences and those who participated in the survey.

If you are looking for an exhaustive step-by-step, recipe-style guide of patterns, this book is not that.

Introduction

One of the biggest challenges for practitioners is to keep going with the test-first approach. Often, projects are not set up for testing, and teams lack a strong culture of testing, as a result: little or no automated tests (whether it is TDD[4], TLD,[5] or any kind of approach, really). Accelerate, Radziwill (2020) already shared that high-performance teams use automation, and TDD is important as a practice to enable that.

In the context of software development as a whole, the test-first approach is really just in its infancy compared to software that was built previously. The premise was that software would always work; that's why no test beforehand was needed. Write the production code, compile it and try it out. If it worked, great! Maybe a few more cases? And then we can move on. That was often the cycle that followed.

Some of those practitioners had to build the culture themselves, building the mindset from the ground up in their teams.[6] Often it is a good idea but can be difficult to maintain long-term. For example, if you are trying to push for an alternative working style and your peers fail to recognize the value in that. In this situation, without the support needed, the easiest approach is to just drop the cause and maintain the habit of writing code and completing manual tests. In the end, going back to the development style we all have been used to for developing applications.[7]

---

4 Test Driven Development.

5 Test Last Development.

6 Julio César Pérez shared his experience facing this scenario, where he reported that team members resisted adopting a test-first approach. You can see the blog post available at https://www.codurance.com/es/publications/una-historia-de-testing.

7 TDD is often mixed with bug-free code, which is not necessarily correct (Dijkstra and others 1970).

On the other hand, various practitioners and teams have adopted Test Driven Development (TDD) as a way to deliver a new feature, install shorter feedback loops and avoid regressions for existing functionality.

Kent Beck (2003) popularised this methodology, which went onto become an industry standard. Not only that, but others have since improved and built on this.

Starting with TDD is not easy and it requires constant maintenance to ensure, for example, that a test suite is able to run fast. Codebases that tackle business problems require a non-trivial amount of code, and with that, an accompanying set of non-trivial set of test cases.

To tackle business needs, different types of tests are required. The test pyramid[8] described in the book Succeeding with Agile by Mike Cohn (2009), and later on referenced by Vocke (2018), suggests the following:

1. Having a solid base of unit tests, which ideally run as fast as possible and provide fast feedback.
2. In the middle we have integration tests that can be slower than unit tests but provides feedback if smaller pieces are working as they should
3. Finally, we have the end-to-end tests (depicted as UI Tests), also references as tests that act as if they were a user (be it a human or another system/program)

Based on the data gathered in this book, it seems that this approach of having the test suite with a pyramid shape is not the case for professional projects.

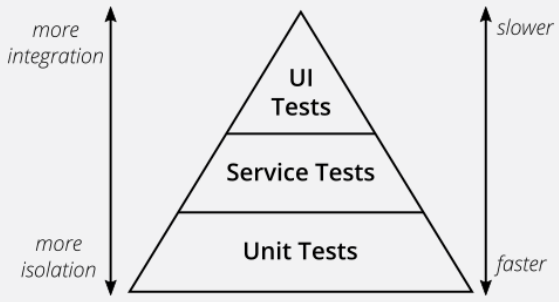


Figure 2: The Test Pyramid

The Test Pyramid from Mike Cohn (2009)

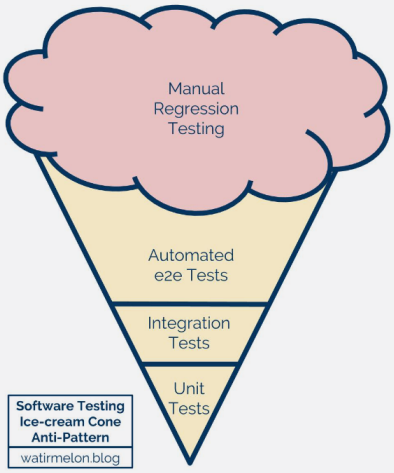
The patterns covered here suggest that there is a misconception about how best to split the type of tests and their responsibilities. For example, the unit test is

8 It is worth mentioning that this same pyramid was cited by Ham Vocke on Martin's Fowler blog https://martinfowler.com/articles/practical-test-pyramid.html.

often referred to as a one-to-one relationship between test and production code.[9]

However, what is often found in the industry is, in fact, the opposite. Usually (despite the age of the aforementioned Test Pyramid), automated test suites are typically comprised of a greater number of slower tests rather than faster tests. In effect, more integration and end-to-end tests than unit tests. Thus leading to the *Ice Cream Cone* pattern instead of the *Test Pyramid*.

The problem with having this Ice Cream test suite pattern instead of the Pyramid is the pain that practitioners feel in maintaining those tests while also maintaining the delivery rate.



The Ice Cream Cone – also referenced by https://alisterbscott.com/kb/testing-pyramids/#site-header

Besides that, Wang, Pyhäjärvi, and Mäntylä (2020) describe that the industry as a whole has an immature test automation process leading to slow feedback. While we recognise the importance and necessity of testing software, we also acknowledge that the industry does not always embrace effective or efficient test processes.

Another subject that is observed is the misconception of code coverage[10], which is a metric that managers often use to force practitioners to write automated tests.

However, this metric alone is not a good measure of success or, indeed, of good testing practice. For example, practitioners may decide to write automated tests in such a manner that the tests exercise the most lines of code through the least number of tests. Rather than following a TDD approach with a large number of smaller tests. Even though when asked, they do not agree with such a strategy.

---

9 It appears in test cases that are written with Oriented Object Programming in mind, one test class means one production class.

10 https://marabesi.com/thoughts/2021/05/29/on-100-percent-code-coverage.

In that sense, later on we will go over an anti-pattern that can occur through this form of chasing test coverage.

This doesn't imply that code coverage is entirely without merit, as it does have a role in the development process, as Mauricio Aniche (2022) suggests.

Looking at the subject of interest here, as recent as it sounds, more than ten years ago, some efforts were made to put together some of the pains practitioners suffer when trying to write code guided by tests. James Carr (2022) devised a list of anti-patterns to look at and keep under control to avoid the ice cream cone test suite pattern that extensive code bases might fall into.

Later on, a thread for voting was created at StackOverflow[11] to open the discussion and allow practitioners to contribute to the list.

More recently, Dave Farley also went through a few of them on his YouTube channel and elaborated on what he called *When TDD goes wrong* (Farley 2021) in an attempt to depict the pains that practitioners can find while doing TDD in a non-optimal fashion. Most examples used there are extracted from open-source projects and fictional examples based on real-world projects. This strategy was deliberate and was used to give the viewer a gist of those anti-patterns and depict that such a scenario happens daily.

Despite the great content delivered by Farley, the examples weren't a comprehensive list, nor were they based on what practitioners understand about the TDD anti-patterns. Instead, it was his interpretation of the issues found when not practicing TDD in the "correct way" (thus the title "When Test Driven Development Goes Wrong").

Yegor Bugayenko (2021) also tried to go over the TDD anti-patterns space, and he brought more topics to the table than what was presented by James Carr, adding a few more anti-patterns when compared with the original list.

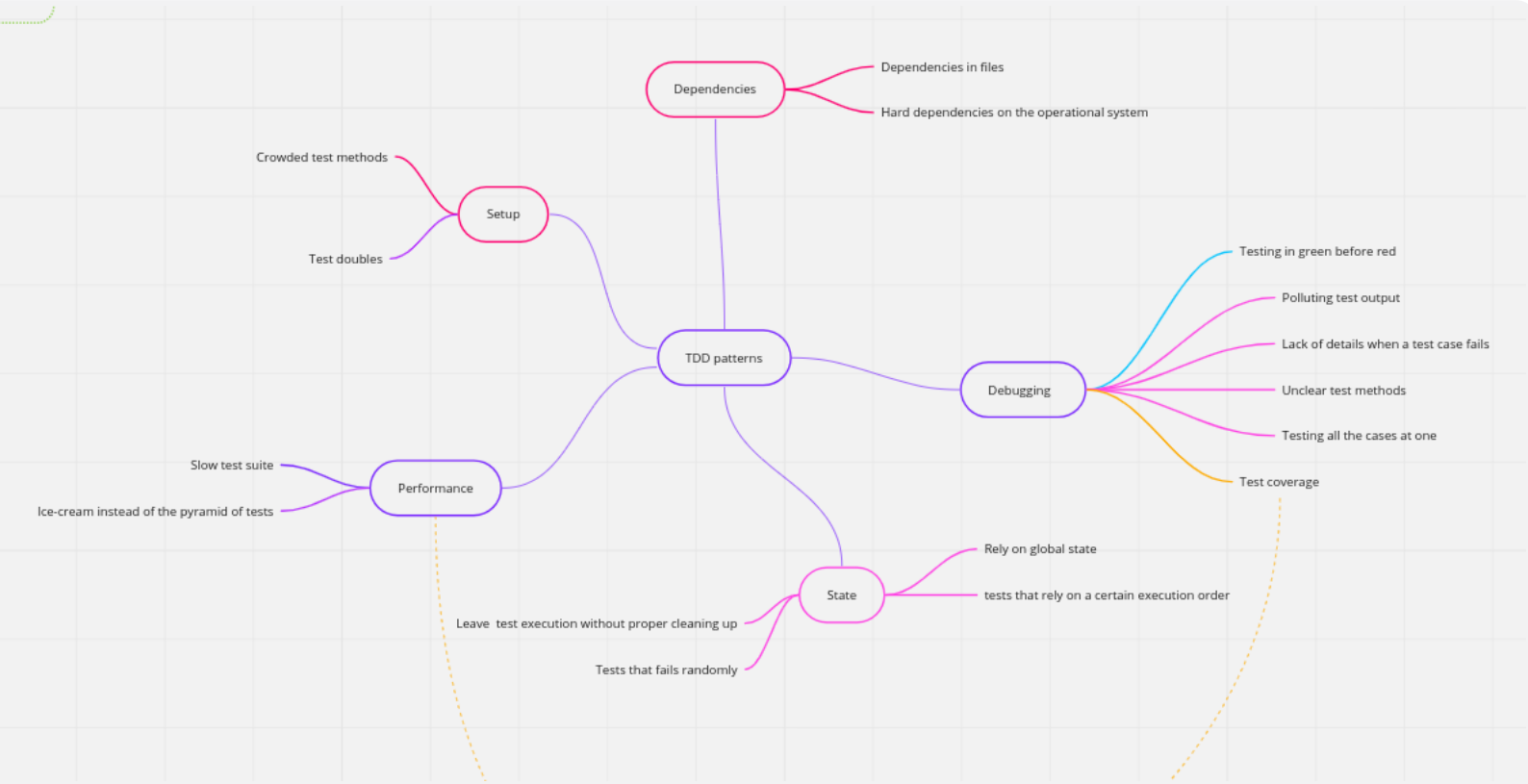
This paved the way for this book, which offers comprehensive examples for each anti-pattern in the list. Not only that, but this book also presents two additional anti-patterns based on the experience of observing practitioners using TDD daily.

Throughout this journey, the mind map depicted below was created to visualize the related subjects and covered here.[12] It might help to see the extent to which the anti-patterns depicted here related to other areas while practicing TDD.

---

11 https://stackoverflow.com/questions/333682/unit-testing-anti-patterns-catalogue.

12 You can see and interact with it by accessing the following link https://bit.ly/3yumaBI.



Mindmap around patterns that make TDD harder.

Book Structure

The book is divided into four levels in total (I – IV) that group the anti-patterns. Each level was designed to depict the progress of a practitioner starting to learn TDD today. This means that level I is more likely to present issues faced by those just starting on a TDD journey. Whereas IV covers some more advanced patterns as the practice of writing tests evolves.

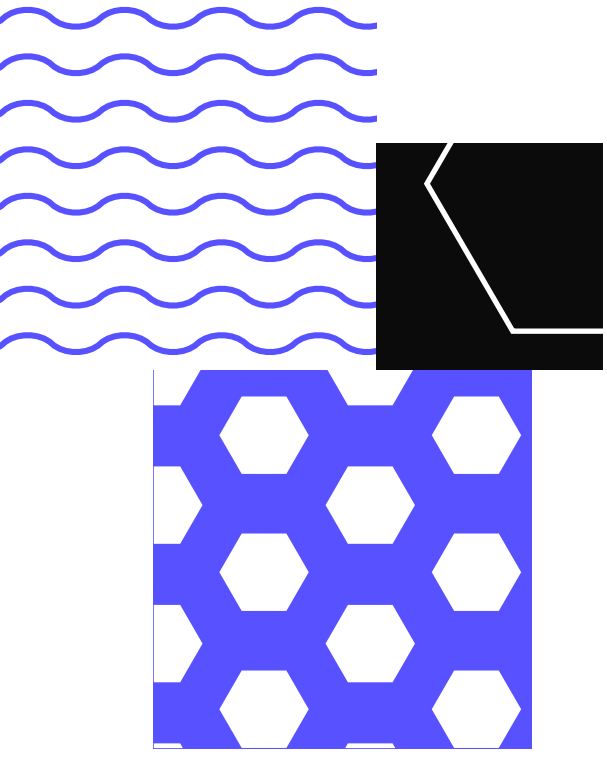
Of course, this can also bring some mixed feelings based on the context in which they happen; some practitioners might have faced issues at the level I later on when they already had some experience writing tests, which is fine as well.

The reason behind the levels is for presentation purposes only. Here, we formatted the content in a way that could be followed in a structured way that allows the readers to relate to their day-to-day practices easily.

A Note Before Starting

Before diving into the book, remember that the code and examples used here are for soley educational purposes and not intended to assign blame or illustrate what not to do. Having a codebase with more than one of the patterns listed here is more than common, and if you haven't seen any of those yet, the time will hopefully come.

The objective here is to shed light on what this can be and, through awareness, try to allow practitioners to manage such potential patterns that often make test driving code harder.



A Case for TDD Anti-Patterns

A Case for TDD Anti-Patterns

In this chapter, we will go over the data collected from practitioners that work daily on software projects across the globe.

The Survey

Before moving any further, it is important to make some clarifications about the survey that was undertaken and what it means for this book.

First, and foremost, is the recognition that the survey used has no scientific or statistical relevance for the subject being surveyed. In other words, despite trying to get answers from practitioners, the data collected is not meant to be used as a basis for any analysis or extrapolation..

As we will see in the next section, the methodology used needs a formal process even though it is reproducible.

Furthermore, while the data collected in its raw form is available on GitHub[13] as a gist, keep in mind that sensitive data such as email addresses are not included. As such, the question “If you want to be notified when the results are published, leave your email address in the box that follows” is empty for all entries.

Methodology

Before the start of the talks in the series about TDD anti-patterns, we asked practitioners if they would join us in participating in a survey that would focus on gathering data for the talks planned for the TDD anti-patterns journey at Codurance.[14]

From that question, 22 people answered the call positively to help us gather insights on what practitioners are doing and what they want to know regarding anti-patterns related to coding in a test-driven fashion.

Based on that, a survey was created containing the questions attached in the appendix. The form was created using google forms[15] and the survey was publicly

13 https://gist.github.com/marabesi/5f0eafd3ea948a5c1dcd25720299ac17.

14 Tweet sent asking if practitioners would join the survey https://twitter.com/MatheusMarabesi/status/1437525252121182214.

15 https://www.google.com/forms.

available from September 15, 2021 until October 5, 2021,[16] the recorded answers were computed in September 2021; the first being on the same day that the survey was available and the last being on September 27, 2021.

To share the survey with as many people as possible, Twitter and LinkedIn were used to disseminate it.[17]

Of the 21 people who said they would participate in the survey, 22 people completed the survey; in the end, we got an extra answer from what the audience intended.

The survey was structured into four sections aimed at collecting data in the following areas:

- Professional background
- TDD practices on the daily basis
- TDD practices of companies I worked at
- Anti-patterns
- Finishing up

In the next section, we will review the results and insights that this survey gave.

Results

In this section, we go over the results produced by the survey shared with practitioners. We will follow the same structure that the survey had, starting with the results from *Professional Background,* followed by the results in *TDD practices on a daily basis, TDD practices at companies I worked at and Anti-patterns.*

Professional Background

We surveyed to gather data on the TDD antipatterns in the industry and from practitioners worldwide. We got 22 answers, and the data shows that practitioners who answered the survey worked on projects in Latin America: Argentina, Brazil, and Mexico, and in Europe: France, Hungary, Ireland, Portugal, Spain, and Romania.

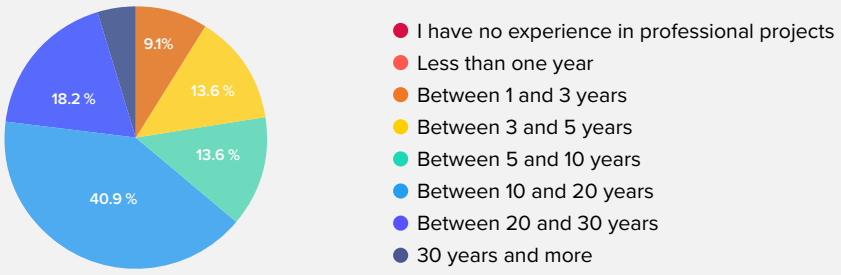
Most of the people who answered also work on professional projects in the

16 Those dates are not absolute, it might vary. The start date is the most trusted one (as it was shared on Twitter that the survey was opened to receive answers), on the other hand, the close date is the one that lacks a formal check; google forms does not offer the possibility to see when a given form was closed for answers. Therefore, Twitter was heavily used for diffusion. For more information on the start and end date, please refer to this twitter timeline https://bit.ly/3AhlBwt.

17 The use of only two social networks prevented the survey from reaching its full potential audience. This resulted in a limitation in the data that must be considered when analysing it.

industry; in total (without taking into account the years of experience) they are 60%. There were no answers to “I have no experience in professional projects”, therefore, 22.7% have between 1 to 5 years of working on professional projects.

The largest response was between 10 and 20 years of experience working on professional projects (40.9% in total).

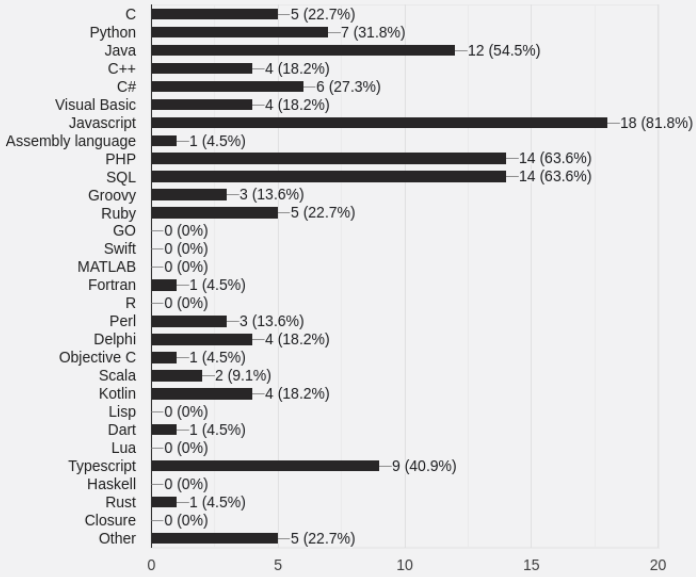


Answers to the question “I am a software developer working in the industry professionally for”

We also found that the most popular programming languages that practitioners work with professionally are:

- JavaScript
- PHP
- Java/Kotlin
- Typescript
- Python

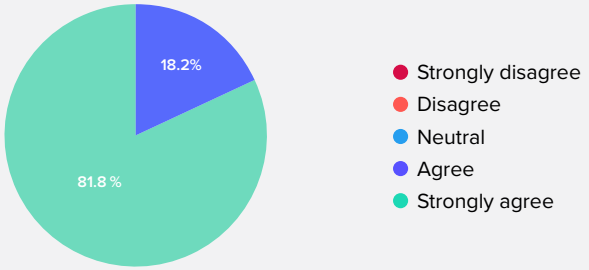
Despite we see a variety of languages appearing in this sample surveyed:



Answers to the question “I write/wrote code professionally in the following languages (programming languages listed are from https://www.tiobe.com/tiobe-index)”

You will find that most of the examples used in the following sections are also in JavaScript. We think this way will be easier to reach a broader audience and give back to those that answered the survey. On the other hand, we also see other languages showing up in the survey that were not as popular: **Ruby**, **Rust** and **Groovy**.

We also found that people that answered the survey are familiar with different testing tools such as **Junit**, **Jest**, **PHPUnit,** or any other framework that provides a common ground to write tests; it is worth sharing that all practitioners marked this question as “Agree” or “Strongly Agree.”



Answers to the question “I am familiar with testing tools such as Junit, Jest, PHPUnit, or any other framework that provides a common ground to write tests.”

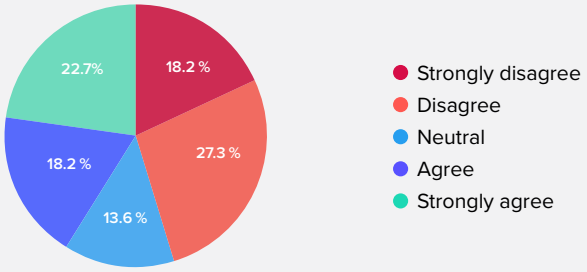
If we go back to the first question that was made in this survey, we have a few people that do not work professionally on projects, but still, they have some familiarity with the testing tools available in the open source.

TDD Practices on the Daily Basis

Next, we will look at the practices that the people who answered the survey follow when developing applications using the test-first approach.

Given that most practitioners work on professional projects, the next question addresses how those people learned TDD to apply to a professional project.

When asked if people that answered the survey learned TDD at work, most of them (45.5%) said that this is not the case; there is a split between Disagree 27.3% and Strongly Disagree 18.2%.



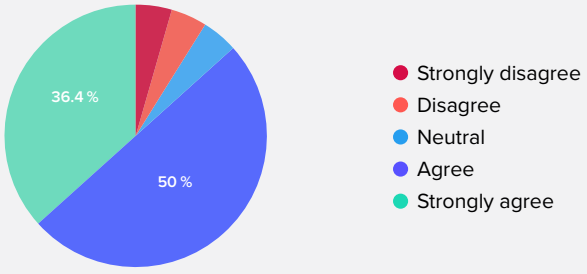
Answers to the question "I learned TDD at work."

The follow-up question focuses more on which sources practitioners learn TDD from (if not from the company they work for), and the majority (86.4%) said that they learned TDD through books, video courses or tutorials.

This can point to practitioners "informally" learning TDD, meaning that from the responses, more than 50% of them learned TDD alone through video courses, books, or tutorials.

This can point to practitioners "informally"[18] learning TDD, meaning that from the responses, more than 50% of them learned TDD alone through video courses, books, or tutorials. Following the same trend, only 50% of the companies understand the pros and cons of TDD and use it as a practice – it is important to mention that the answers come from practitioners. As such, this is a perception that they have regarding the topic.

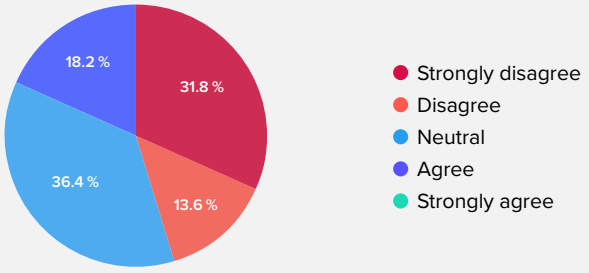
This question does not address the mix of all the options given; potentially that could be the case that practitioners mix different forms to learn TDD, for example, watching a video course and reading a book to fill in possible gaps.



Answers to the question, "I learned TDD myself, through books, video courses or tutorials."

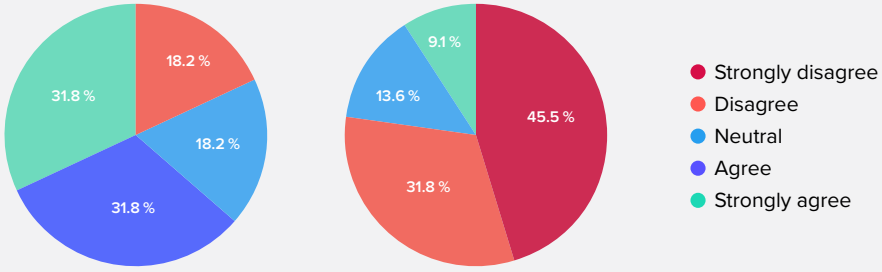
18 Informally here means not having a certification or institutional recognition for practicing TDD.

We also asked if the respondents peers knew TDD, and most of them (45.4%) answered that that was not the case, with 36.4% remaining neutral.



Answers to the question "People that I work/worked with already knew TDD."

We then asked if their peers practiced TDD, on a daily basis, and also if they, themselves, work on writing tests first.



On the left, answers to the question "The people I work with practice TDD as part of daily work," and on the right side, answers for the question "I practice TDD as part of my daily workflow."

Related to this, we also asked if adopting TDD had led them to perceive a slowdown in their development while coding. This is often the case when practitioners start to learn to test drive code. This is the case for hard skills. If we think about it for any new piece of technology, the start is the initial phase that takes longer to get used to.

77.3% disagree with this statement; in other words, most practitioners see value in practicing TDD and do not feel that the practice slows them down.[19]

Despite the survey, the well-known author Uncle Bob from the tech community,

19 It is important to mention that this survey was shared via Twitter and LinkedIn for people close to communities that already have such familiarity with TDD. This is a point of attention for the survey as a whole as it does not depict the state of the industry but rather a small number of practitioners that work in the industry.

who is the author of the book *Clean Architecture*, Martin (2017), shared his adoption of using TDD, stating that "the only way to go fast is to go well."

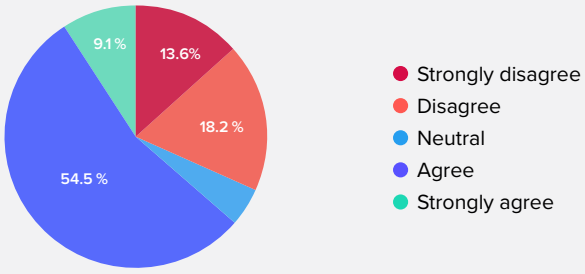
Such a statement is aimed at the perception that TDD slows the developer flow down; as the benefit for testing drive code will not show immediately after writing the test, but maybe in refactoring in which the tests catch possible failures that would have been noticed only in production by the end user.[20]

TDD Practices at Companies I Worked

Next up, we will go over more questions related to the daily environment in which the practitioners work.

We started with the question, "I am not allowed to push code for review without a test case with it", and for such, a premise was assumed in which the practitioners work on a gitflow[21] fashion instead of Trunk Base Development.[22] [23] The pull request is a popular method practitioners use to ask for code reviews.

In that sense, we found that most of the practitioners agree with that (63.6%), which we could explore a bit more why that is the case, are practitioners forced by some hierarchy? Why is there such a restriction? Further exploration is required to explore this topic more.



Answers to the question "I am not allowed to push code for review without a test case with it."

Related to that, we also asked if the companies that the practitioners work for required TDD to join them. This time we see that Neutral has 18.2%, which can lead to people that do not work for any company.

---

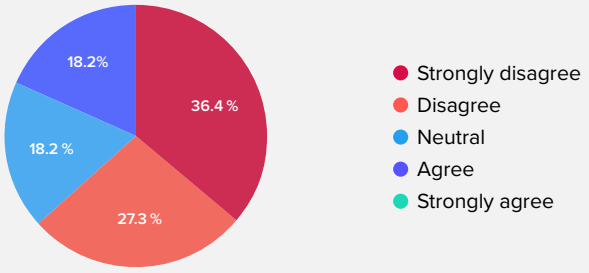
20 "Slow" and "fast" can be related to the codebase itself as well, but usually it is used in a way to avoid using TDD as a practice.

21 https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

22 Dave Farley also shared his thoughts on why git-flow might be a bad idea https://www.youtube.com/watch?v=_w6TwnLCFwA.

23 Trunk Base Development explained https://trunkbaseddevelopment.com.

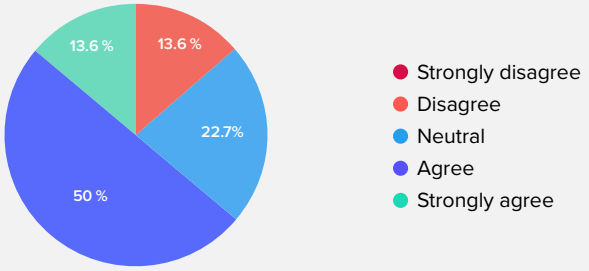
If we combine “Strongly disagree” and “Disagree”, we see that most of the companies (63.7%) did not require TDD beforehand.



Answers to the question “The companies I work/worked at required TDD to join them as part of the job description.”

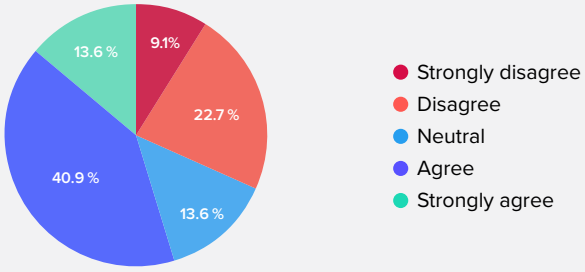
The next question tries to depict the work environment regarding practicing TDD, as the previous question already sheds some insights on that (as most companies do not require TDD as part of their job description); here, we see that we potentially have a follow-up on that trend.

For most of the practitioners, 63.6% said that the company they work for does not practice TDD.



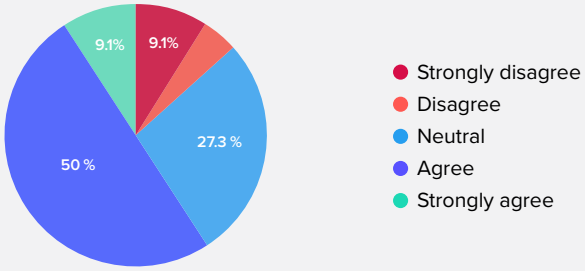
Answers to the question “Companies I work/worked at do not practice TDD.”

Practitioners (54.5%) also shared that the companies that they work for have the perception that tasks will take longer to complete if the team is using TDD.



Answers to the question "Companies I work/worked at argued that TDD requires more time to complete a task and the teams didn't have the required time to use it."

Connected to that, we also asked if the company that the practitioners work/ worked for understand/understood the pros and cons of using TDD, and 59.9% said that yes, it does.



Answers to the question "Companies I work/worked at value the TDD practice and acknowledge its pros and cons."

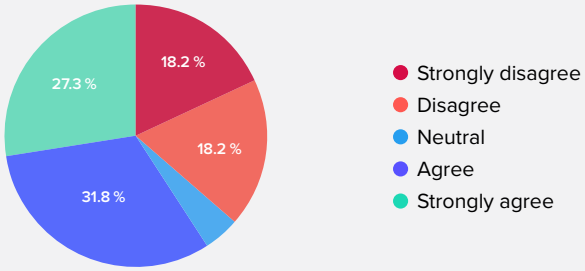
This can be related to the previous question as a deliberate decision to not use or use TDD in the software development process or as part of the recruitment process.

Next up, we will go over the section that asks practitioners about TDD anti-patterns. Even though there are four sections of the survey, This is the last section used to gather data related to the topic of this book; the last section named **Finishing up,** is aimed at collecting personal data.

Anti-Patterns

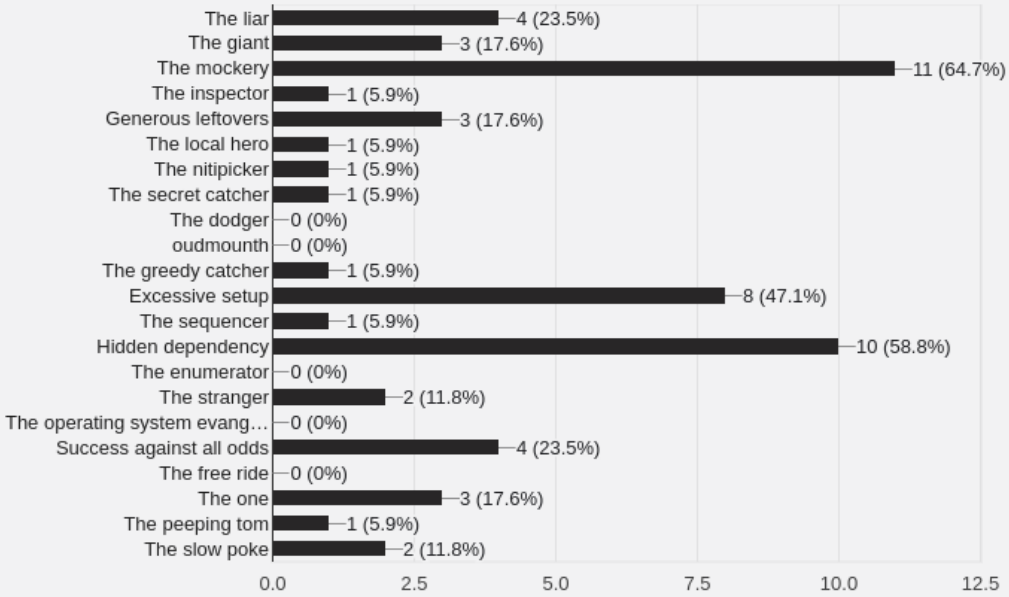
This section focuses on what is known as "TDD anti-patterns," which aims to depict what the practitioners knew about them and to what extent, given that TDD anti-patterns are not a popular subject among practitioners.

To get started, the first question tried to see if practitioners could recall at least one anti-pattern defined by James Carr (2022). As the names of the catalogued anti-patterns are not that familiar, this question shows that practitioners can recall a few of them regardless of that.



Answers to the question "I can recall in my mind at least one TDD anti-pattern"

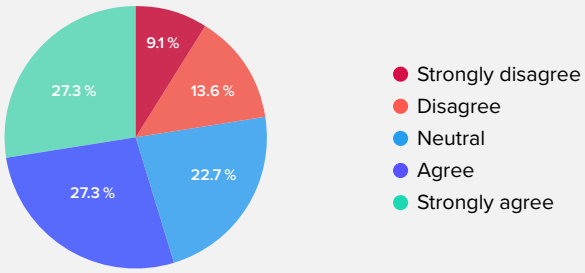
The follow-up question was created to depict which anti-pattern the practitioners who answered the survey could recall. The most popular was The Mockery.



Answers to the question "From the following list, check the anti-patterns you can recall."

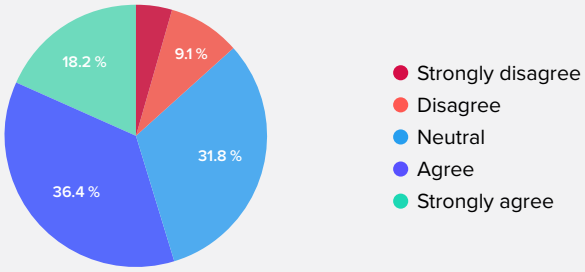
One of the pain points related to anti-patterns is usually in the code base that practitioners work on, but due to the lack of awareness, often those anti-patterns are not spotted on a daily basis

One of the pain points related to anti-patterns is usually in the code base that practitioners work on, but due to the lack of awareness, often those anti-patterns are not spotted on a daily basis. From the responses, 54.6% said that they could recall at least one anti-pattern.



Answers to the question "I could spot at least one TDD anti-pattern in the code base I work/worked on professionally."

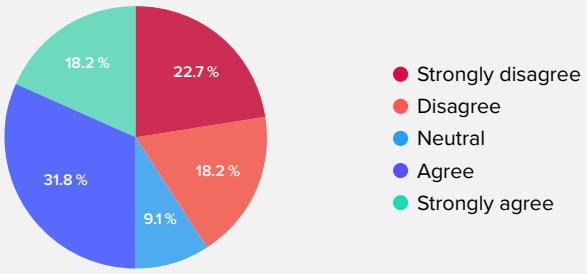
Regarding the question "I feel that TDD anti-patterns slow me down," 31.8% remained Neutral; this is potentially related to the question "I can recall in my mind at least one TDD anti-pattern" that had 22.7% of answers – If I cannot recall it is difficult to know if it slows me down or not.



Answers to the question "I feel that TDD anti-patterns slow me down."

The next question relates to a few anti-patterns that arise when the test is written after the production (if any at all), which sometimes is not written, leading to a code base without tests.

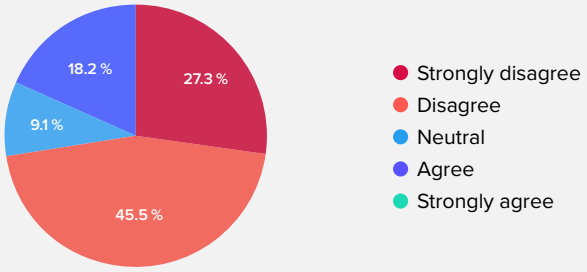
Thus, when practitioners try to add some tests, it slows them down in the process of getting something working with the test first. In total, 50% of practitioners agree with that.



Answers to the question "When I tried to practice TDD in a code base without tests, I felt that I was slowed down by writing the test first."

Test coverage is a subject that is usually used to explore the code base and see where the code lacks more test cases; as such, this is often a way.

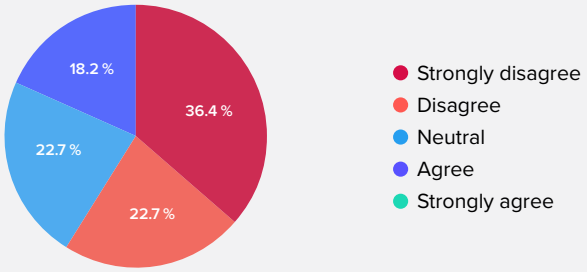
Out of the practitioners that answered 18.2% were indeed worried about coverage, against 72.8% that disagree (this is often a popular subject that divides practitioners as it can be used in an ineffective way).



Answers to the question "I write tests because I am worried about the code coverage."

Last but not least, in this section, we also asked if practitioners write tests because the project they work on demands it. We found that 59.1% disagreed with such a question.

This relates to the question, "The companies I work/worked at required TDD to join them as part of the job description" as companies do not require TDD to be in their recruitment process (see below), it seems feasible that most practitioners are not required to write tests for the project they are working on.



Answers to the question "I write tests, but without TDD because the project I am working on demands tests."

In this chapter, we went through the survey shared with practitioners to gather data on the state of anti-patterns, and we shared the results here and also opened a few questions that will require further investigation.

Now it is time to start diving into the anti-patterns one by one and see the impact they bring to practitioners while developing applications.

In the next chapter, we will start with the anti-patterns that practitioners who are starting with the practice of TDD might face.

Level I

Level I

The first level of this series relates to practitioners that just started to test-drive their code. As you might have expected, this is also the section that has the biggest number of anti-patterns compared to the others that follow.

One of the possible reasons for that is effectively how practitioners learn TDD (here, we can also add any approach related to testing drive applications).

The answers of the survey show one key result: that practitioners learn TDD informally.

Throughout this level, you will also see subjects related to:

- How depending on dependencies such as the operating system can harm testability.
- Creating dependencies in which the test runs beyond the operating system can also harm testability (for example, depending on the file system).
- Naming test cases are used to debug and quickly spot problems; naming them randomly harms understandability.
- Favour adding new test cases instead of polluting a single test case with many assertions.
- Avoid coupling test cases with the order in which they appear in a list (unless the order has a meaning).
- While building assertions focuses on the specific properties that the test needs instead of comparing an entire object.
- Focus on the desired behaviour instead of relatively simple actions such as testing a selection from the database.
- Pay a closer look at async-oriented or time-oriented tests to prevent false positives.
- Cluttering the test output with warnings or error messages (even when the test is green) might lead to miss understanding; try to avoid that whenever possible.

As there were many topics to go over, an effort was made to keep the sections split from each other so that each section is consumable individually.

The Operating System Evangelist

A unit test that relies on a specific operating system environment to be in place to work. A good example would be a test case that uses the newline sequence for Windows in an assertion, only to break when run on Linux, Carr (2022).

The Operating System Evangelist is covered in Episode 5 of the video[24] series covering the TDD anti-patterns hosted by Codurance.

The Lutris Project

The *Operating System Evangelist* is related to how coupled the testing code is to the operating system; the coupling can be on different aspects of the code, for example, using a specific path that exists only on Windows.

To depict such a case, the code snippet that follows was extracted from the open-source project Lutris.[25] Lutris aims to run games that are created for Windows on Linux. The premise of the project already gives some expected constraints in the codebase. The result is the following test case that launches a Linux process:

```
class LutrisWrapperTestCase(unittest.TestCase):
    def test_excluded_initial_process(self):
        "Test that an excluded process that starts a monitored process works"
        env = os.environ.copy()
        env['PYTHONPATH'] = ':'.join(sys.path)
        # run the lutris-wrapper with a bash subshell. bash is "excluded"
        wrapper_proc = subprocess.Popen(
            [
                sys.executable, lutris_wrapper_bin, 'title', '0', '1', 'bash',
                'bash',
                '-c',
                "echo Hello World; exec 1>&-; while sleep infinity; do true;
                done"
            ],
            stdin=subprocess.DEVNULL, stdout=subprocess.PIPE, env=env,
        )
```

The test case relies on a bash shell to execute the test case, and as a result, it would fail if we tried to execute it on a Windows environment, for example. Not to say that it is bad; rather, this is a trade-off between the focus of the project and the cost of having an abstraction on top of the underlying operating system.

In the end, for this specific scenario, we could argue that it is less likely that Lutris supports another operating system that justifies the cost of maintaining an abstraction.

---

24 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-5

25 https://github.com/lutris/lutris/blob/f5e8e007b3e492be1fd07ca695ad6e0e25fab1d5/tests/test_lutris_wrapper.py

In their 2020 book, *Cosmic python* (Chapter 3)[26] Harry Percival and Bob Gregory shared the idea behind coupling and abstraction. In their book, they discussed the idea of using the filesystem. Specifically, the file path and the implications of using a path directly without taking into account any abstraction leads to a coupled code (therefore harming testability as well).[27]

The Operating System Evangelist also appeared in the programming language Go lang, in an issue that was trying to mitigate the new line character differences between Linux and Windows operating systems. This issue is part of the definition of this anti-pattern "A good example would be a test case that uses the newline sequence for Windows in an assertion, only to break when run on Linux." Within that Github issue thread, a user shares the issues they are facing when needing to run the same tests on Windows. She states that most errors are due to the difference between the feed line code.

Issue from go lang repository, reporting frustration in handling feed lines in Windows and Linux.

Another anti-pattern related to The Operating System Evangelist is The Local Hero. The Local Hero is known for having everything in place locally to run an application, but as soon as you try to run it on another machine, it will fail.

We will discuss The Local Hero later on, but to reinforce how they are connected, here is an example of Jenkins source code:

```
1  @Test
2  public void testWithoutSOptionAndWithoutJENKINS_URL() throws Exception {
3      Assume.assumeThat(System.getenv("JENKINS_URL"), is(nullValue()));
4      // TODO instead remove it from the process env?
5      assertNotEquals(0, launch("java",
6              "-Duser.home=" + home,
7              "-jar", jar.getAbsolutePath(),
8              "who-am-i")
9      );
10 }
```

26 Chapter 3, A Brief Interlude On Coupling and Abstractions.

27 The example used can also be related to the strategy design pattern.

This snippet is particularly interesting because whoever wrote it already noticed some smell[28] going on with the comment: TODO instead remove it from the process env?

Interestingly enough, in the book *The Programmers Brain*, Felienne Hermans (2021) shared that adding TODOs notes while coding helps programmers to remember to come back and fix it; this is what she called to be a technique to help the prospective memory.

Therefore, as she also highlights, usually, that is not the case. This kind of comment tends to remain unsolved in a codebase for a long time.[29]

Lastly, coding katas are usually good places to catch these kinds of patterns early on and push for an abstraction during the refactoring phase.

For example, The WordWrap[30] is an example of a kata that aims to break into new lines if the content is greater than expected. For an explanation of the differences in feed lines and operating systems, check Baeldung.com post.[31]

Points of Attention

1. Avoid depending on the operating system; prefer to add an abstraction whenever possible based on the context.

The Local Hero

A test case that is dependent on something specific to the development environment it was written on in order to run. The result is the test passes on development boxes but fails when someone attempts to run it elsewhere, Carr (2022).

The Local Hero is covered in Episode 2 of the video[32] series covering the TDD anti-patterns hosted by Codurance.

The TDD anti-patterns precede the more recent rise of container usage in software development. Before, it was common to have differences between the machine on which the developer was working and the server on which the application would run. Often, these environments were not the same; configuration specific to the developer machine got in the way during the deployment process reaching the production server and, as a result, crashed the system.

---

28 Code smells are defined by the feeling that developers have while reading/writing code that something is not correct.

29 So, next time you face this situation, think twice instead of adding the TODO tag in the code!

30 https://codingdojo.org/kata/WordWrap.

31 https://www.baeldung.com/java-string-newline.

32 https://www.codurance.com/publications/tdd-anti-patterns-chapter-2.

PHP, for example, relies heavily on extensions that can or can't be enabled on the server. Extensions include threads, drivers for connecting to a database, and many more.

In this case, if the developer relied on a specific version for the given extension, the test would run successfully. Still, as soon as we try to run the suite on another machine (such as a Continuous Integration server), it would fail.

Not only that, environmental variables can get in the way of testing too. For example, the following code depicts a component that needs a URL to load a survey (some of the code has been removed/modified intentionally and adapted to fit the example – for more info, follow the GitHub link):

```
1  import { Component } from 'react';
2  import Button from '../../buttons/primary/Primary';
3
4  import '../../../../scss/shake-horizontal.scss';
5  import './survey.scss';
6
7  const config = {
8    surveyUrl: process.env.REACT_APP_SURVEY_URL || '',
9  }
10
11 const survey = config.surveyUrl;
12
13 const mapStateToProps = state => ({
14   user: state.userReducer.user,
15 });
16
17 export class Survey extends Component {
18   /* skipped code */
19
20   componentDidMount = () => { /* skipped code */}
21
22   onSurveyLoaded = () => { /* skipped code */}
23
24   skipSurvey = () => { /* skipped code */}
25
26   render() {
27     if (this.props.user.uid && survey) {
28       return (
29         <div className={`w-full ${this.props.className}`}>
```

```
30            {
31              this.state.loading &&
32              <div className="flex justify-center items-center text-white">
33                <h1>Carregando questionario</h1>
34              </div>
35            }
36
37            <iframe
38              src={this.state.surveyUrl}
39              title="survey form"
40              onLoad={this.onSurveyLoaded}
41            />
42
43            {
44              !this.state.loading && this.props.skip &&
45              <Button
46                className="block mt-5 m-auto"
47                description={this.state.buttonDescription}
48                onClick={this.skipSurvey}
49              />
50            }
51          </div>
52        );
53      }
54
55      return (
56        <div className="flex justify-center items-center text-white">
57          <h1 className="shake-horizontal">Ocorreu um erro ao carregar o
            questionario</h1>
58        </div>
59      );
60    }
61  }
62  /* skipped code */
63  And here goes the test case for these components:
64  import { mount } from 'enzyme';
65  import { Survey } from './Survey';
66  import { auth } from '../../../../pages/login/Auth';
67  import Button from '../../buttons/primary/Primary';
68
69  describe('Survey page', () => {
70
```

```
71   test('should show up message when survey url is not defined',() => {
72     const wrapper = mount(<Survey user={{}}/>);
73     const text = wrapper.find('h1').text();
74   });
75
76   test('should not load survey when user id is missing', () => {
77     const wrapper = mount(<Survey user={{}} />);
78     const text = wrapper.find('h1').text();
79   });
80
81   test('load survey passing user id as a parameter in the query string', () => {
82     const user = { uid: 'uhiuqwqw-k-woqk-wq--qw' };
83
84     const wrapper = mount(<Survey user={user} />);
85     const url = wrapper.find('iframe').prop('src');
86     expect(url.includes(auth.user.uid)).toBe(true);
87   });
88
89   test('should not up button when it is loading', () => {
90     const user = { uid: 'uhiuqwqw-k-woqk-wq--qw' };
91
92     const wrapper = mount(<Survey user={user} />);
93     expect(wrapper.find(Button).length).toBe(0);
94   });
95
96   test('should not up button when skip prop is not set', () => {
97     const user = { uid: 'uhiuqwqw-k-woqk-wq--qw' };
98
99     const wrapper = mount(<Survey user={user} />);
100    expect(wrapper.find(Button).length).toBe(0);
101  });
102
103  test('show up button when loading is done and skip prop is true', () => {
104    const user = { uid: 'uhiuqwqw-k-woqk-wq--qw' };
105
106    const wrapper = mount(<Survey user={user} skip={true} />);
107    wrapper.setState({
108      loading: false
109    });
110    expect(wrapper.find(Button).length).toBe(1);
111  });
112 });
```

Despite the code age (long-time class components in reactjs), it does the job well. Deriving the behaviour from the test cases, we understand that some loading is going on based on the survey URL and the user id. Unfortunately, the implementation details matter the most – if we run the test case for the current implementation, it will fail.

```
Test Suites:        1 failed, 62 passed, 63 total
Tests:      3 failed, 593 passed, 596 total
```

And the fix for such a run is to export an environment variable named REACT_APP_SURVEY_URL. Well, the easy fix would be to use the env variable. The long-term fix would be to avoid depending on the external definition and assume some defaults; here are some ideas that come to my mind to fix that properly:

- Assume a dummy variable as a default.
- Do not use any URL and build the tests around having it or not – if not, just skip the execution.

Another example would be relying on the underlying file system. This issue is also discussed in a Stack Overflow thread. The issue with the dependent test is: that the test would run only on a Windows machine. Ideally, external dependencies should be avoided using test doubles.

Points of Attention

1. File system
2. Dependencies on the operating system
3. External configuration management

The Enumerator

A unit test with each test case method name is only an enumeration, i.e., test1, test2, test3. As a result, the intention of the test case is unclear, and the only way to be sure is to read the test case code and pray for clarity, Carr (2022).

The *Enumerator* is covered in Episode 4 of the video[33] series covering the TDD anti-patterns hosted by Codurance.

Enumerating requirements in a brainstorming session usually is a good idea; it can be handy to create such a list for later consumption, and those can even become new features for a software project.

As in software, we are dealing with features; it seems to be a good idea to translate those in the same language and order, so verifying those becomes a checklist.

---

33 https://app.livestorm.co/codurance/testing-anti-patterns-episode-4

As good as it might sound for organization and feature handling, translating such numbered lists straight to code might bring undesired readability issues and even more for test code.

As weird as it might sound, enumerating test cases with numbers is common among starters. For some reason, at first, it seems a good idea for them to write down the same test description and add a number to identify it. The following code depicts an example:

```
from status_processor import StatusProcessor

def test_set_status():

    row_with_status_inactive_1 = dict(

    row_with__status_inactive_2 = dict(

    row_with_status_inactive_3 = dict(

    row_with_status_inactive_3b = dict(

    row_with_status_inactive_4 = dict(

    row_with_status_inactive_5 = dict(
```

The question for new practitioners inside the codebase is: *What does 1 means? What does 2 mean? Are those the same test case?* In a nutshell, the key point here is the space for being explicit about what is being tested. This is also explored by Martin (2009) in the section G25: Replace Magic Numbers with Named Constants.

The first example was in python, but this anti-pattern arises in different programming languages. The following example in typescript is another type of enumerating test cases, in this scenario, the test cases are file names that are used to run the tests.

Example of The Enumerator running in a GitHub Actions pipeline.

Enumerating test scenarios could hide some business patterns that are being replaced by numbers. The intention of what is being tested is not clear. Another issue that comes with that is the mitigation problem, if any of those tests fail, the error message will most likely give you a number, but not the root cause of the failure.

Points of Attention

1. Are we using 1, 2, 3?
2. The test that failed was easy to understand? And if so, why?

The Free Ride

Rather than write a new test case method to test another feature or functionality, a new assertion rides along in an existing test case (Carr 2022).

The Free Ride is covered in Episode 5 of the video[34] series covering the TDD anti-patterns hosted by Codurance.

The Puppeteer Project

The Free Ride is among the least popular anti-patterns in the survey. Perhaps this is because the name makes it difficult to recall its meaning.

---

34 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-5.

The Free Ride appears in test cases that usually require a new test case to test the desired behaviour. Still, another assertion is put in place, and sometimes even logic inside the test case is added to support this addition.

Let's have a look at the following example that was extracted from the Puppeteer project[35]:

```
it('Page.Events.RequestFailed', async () => {
    const { page, server, isChrome } = getTestState();

    await page.setRequestInterception(true);
    page.on('request', (request) => {
      if (request.url().endsWith('css')) request.abort();
      else request.continue();
    });
    const failedRequests = [];
    page.on('requestfailed', (request) => failedRequests.push(request));
    await page.goto(server.PREFIX + '/one-style.html');
    expect(failedRequests.length).toBe(1);
    expect(failedRequests[0].url()).toContain('one-style.css');
    expect(failedRequests[0].response()).toBe(null);
    expect(failedRequests[0].resourceType()).toBe('stylesheet');

    if (isChrome)
      expect(failedRequests[0].failure().errorText).toBe('net::ERR_FAILED');
    else
      expect(failedRequests[0].failure().errorText).toBe('NS_ERROR_FAILURE');
    expect(failedRequests[0].frame()).toBeTruthy();
  });
```

The Free Ride anti-pattern manifests itself in the above code in if/else statements at the end of the test. There are two test cases in this single test, but presumably, the idea was to reuse the same setup code and slide in an additional assertion within the same test case[36].

Another approach would be to split the test case to focus on a single scenario at a time. Puppeteer itself already mitigated this issue using a function of handling such a scenario. Using that to split the test cases, we would have the first test case focuses on the chrome browser:

---

35 https://github.com/puppeteer/puppeteer/blob/9ca57f190c85c4b6af5665a8cfe4703571e0edde/test/network.spec.ts#L497.

36 Using logic inside the test case relates to The Success Against All Odds that is discussed in section 9.1.

```
itChromeOnly('Page.Events.RequestFailed', async () => {
    const { page, server } = getTestState();

    await page.setRequestInterception(true);
    page.on('request', (request) => {
      if (request.url().endsWith('css')) request.abort();
      else request.continue();
    });
    const failedRequests = [];
    page.on('requestfailed', (request) => failedRequests.push(request));
    await page.goto(server.PREFIX + '/one-style.html');
    expect(failedRequests.length).toBe(1);
    expect(failedRequests[0].url()).toContain('one-style.css');
    expect(failedRequests[0].response()).toBe(null);
    expect(failedRequests[0].resourceType()).toBe('stylesheet');
    expect(failedRequests[0].failure().errorText).toBe('net::ERR_FAILED');
    expect(failedRequests[0].frame()).toBeTruthy();
  });
```

And then, the second case for Firefox:

```
itFirefoxOnly('Page.Events.RequestFailed', async () => {
    const { page, server } = getTestState();

    await page.setRequestInterception(true);
    page.on('request', (request) => {
      if (request.url().endsWith('css')) request.abort();
      else request.continue();
    });
    const failedRequests = [];
    page.on('requestfailed', (request) => failedRequests.push(request));
    await page.goto(server.PREFIX + '/one-style.html');
    expect(failedRequests.length).toBe(1);
    expect(failedRequests[0].url()).toContain('one-style.css');
    expect(failedRequests[0].response()).toBe(null);
    expect(failedRequests[0].resourceType()).toBe('stylesheet');
    expect(failedRequests[0].failure().errorText).toBe('NS_ERROR_FAILURE');
    expect(failedRequests[0].frame()).toBeTruthy();
  });
```

Logic inside the test case is already an indication that *The Free Ride* anti-pattern is playing a role. The Puppeteer example can be improved even further.[37]

Now that we shave split the logic into two separate test cases, there is some duplicated code (that could be an argument for adopting *The Free Ride*). If that is the case, the testing framework can help us here.

To avoid code duplication in this scenario, we could use the hook *beforeEach* and move the required setup there.

The Jenkins Project

Moving on from the Puppeteer Project, there are other ways in which *The Free Ride* can appear. Let's switch to another open-source project that also demonstrates *The Free Ride* anti-pattern.

The following code was extracted from the Jenkins project and it also shows the signs of *The Free Ride*. But before diving into that, let's have a look at the source code:

```
1  public class ToolLocationTest {
2        @Rule
3        public JenkinsRule j = new JenkinsRule();
4
5        @Test
6        public void toolCompatibility() {
7            Maven.MavenInstallation[] maven = j.jenkins.getDescriptorByType(Maven.
             DescriptorImpl.class).getInstallations();
8            assertEquals(1, maven.length);
9            assertEquals("bar", maven[0].getHome());
10           assertEquals("Maven 1", maven[0].getName());
11
12           Ant.AntInstallation[] ant = j.jenkins.getDescriptorByType(Ant.
             DescriptorImpl.class).getInstallations();
13           assertEquals(1, ant.length);
14           assertEquals("foo", ant[0].getHome());
15           assertEquals("Ant 1", ant[0].getName());
16     JDK[] jdk = j.jenkins.getDescriptorByType(JDK.DescriptorImpl.class).
       getInstallations();
17           assertEquals(Arrays.asList(jdk), j.jenkins.getJDKs());
```

37 It is important to highlight that the Puppeteer project also welcomed the pull request that fixed The Free Ride depicted in this section; for further details, please refer to the following pull request: https://github.com/puppeteer/puppeteer/pull/8095.

```
18          assertEquals(2, jdk.length); // JenkinsRule adds a 'default' JDK
19          assertEquals("default", jdk[1].getName()); // make sure it's really
            that we're seeing
20          assertEquals("FOOBAR", jdk[0].getHome());
21          assertEquals("FOOBAR", jdk[0].getJavaHome());
22          assertEquals("1.6", jdk[0].getName());
23      }
24  }
```

Another approach to avoid *The Free Ride,* in this case, would be once again to split the test cases:

```
1   public class ToolLocationTest {
2       @Test
3       @LocalData
4       public void shouldBeCompatibleWithMaven() {
5           Maven.MavenInstallation[] maven = j.jenkins.getDescriptorByType(Maven.
            DescriptorImpl.class).getInstallations();
6           assertEquals(1, maven.length);
7           assertEquals("bar", maven[0].getHome());
8           assertEquals("Maven 1", maven[0].getName());
9       }
10    @Test
11      @LocalData
12      public void shouldBeCompatibleWithAnt() {
13          Ant.AntInstallation[] ant = j.jenkins.getDescriptorByType(Ant.
            DescriptorImpl.class).getInstallations();
14          assertEquals(1, ant.length);
15          assertEquals("foo", ant[0].getHome());
16          assertEquals("Ant 1", ant[0].getName());
17      }
18      @Test
19      @LocalData
20      public void shouldBeCompatibleWithJdk() {
21          JDK[] jdk = j.jenkins.getDescriptorByType(JDK.DescriptorImpl.class).
            getInstallations();
22          assertEquals(Arrays.asList(jdk), j.jenkins.getJDKs());
23          assertEquals(2, jdk.length); // JenkinsRule adds a 'default' JDK
24          assertEquals("default", jdk[1].getName()); // make sure it's really
            that we're seeing
25          assertEquals("FOOBAR", jdk[0].getHome());
```

```
26            assertEquals("FOOBAR", jdk[0].getJavaHome());
27            assertEquals("1.6", jdk[0].getName());
28        }
29    }
```

The split would also bring the additional benefit of making it far easier to identify the causes of a test failure

Points of Attention

1. If a test has assertions that assert different behaviours, this is a candidate for splitting out into separate tests.
2. Starting with everything in a single test case is fine, but not refactoring the tests is something to watch for.

The Sequencer

A unit test that depends on items in an unordered list appearing in the same order during assertions (Carr 2022).

The Sequencer is covered in Episode 4 of the video[38] series covering the TDD anti-patterns hosted by Codurance.

The Sequencer brings light to a subject related to what was covered in the testing assertions blog post, Marabesi (2022), which depicts ways of improving the feedback of test cases based on the type of assertion used (in this case, using jest as a testing framework). More specifically, the section about **Array Containing** depicts what *The Sequencer* is.

In short, *The sequencer* anti-pattern appears when an unordered list is used to assert that it adheres to a given order – in other words, giving the idea that the items on the list are required to be ordered. Which often is the source of wasted time just to realize that everything was working as expected but not in the order expected.

The following example shows *The sequencer* in practice; the test case checks if the desired fruit is inside the list; the focus here is to know if the fruit is or is not on the list regardless of the position in which it might appear:

```
30 const expectedFruits = ['banana', 'mango', 'watermelon']
31
32 expect(expectedFruits[0]).toEqual('banana')
33 expect(expectedFruits[1]).toEqual('mango')
34 expect(expectedFruits[0]).toEqual('watermelon')
```

38 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-4

As we don't care about the position, using the utility *arrayContaining* might be a better fit and makes the intention explicit for further readers.

```
1  const expectedFruits = ['banana', 'mango', 'watermelon']
2
3  const actualFruits = () => ['banana', 'mango', 'watermelon']
4
5  expect(expectedFruits).toEqual(expect.arrayContaining(actualFruits))
```

It is important to note that *arrayContaining* also ignores the items' position and if there is an extra element. If the code under test cares about the exact number of items, it would be better to use a combination of assertions. This behaviour is described in the official Jest documentation.

The example outlined using Jest gives a hint on what to expect in codebases that have this anti-pattern. Still, the following illustration depicts a scenario in which the sequencer appears for a CSV file.[39]

```
1  def test_predictions_returns_a_dataframe_with_automatic_predictions(self,form):
2    order_id = "51a64e87-a768-41ed-b6a5-bf0633435e20"
3    order_info = pd.DataFrame({"order_id": [order_id], "form": [form],})
4    file_path = Path("tests/data/prediction_data.csv")
5    service = FileRepository(file_path)
6
7    result = get_predictions(main_service=service, order_info=order_info)
8
9    assert list(result.columns) == ["id", "quantity", "country", "form", "order_
     id"]
```

On line 4, the CSV file is loaded to be used during the test. Next, the result variable is what will be asserted against, and on line 9, we have the assertion against the columns found in the file.

CSV files use the first row as the file header separated by a comma; in the first row is where the name of the columns is defined, and the lines below follow the data each column should have. If the CSV happens to be changed with a different column order (in this case, switching country and form), we will see the following error:

```
1  tests/test_predictions.py::TestPredictions::test_predictions_returns_a_
   dataframe_with_automatic_predictions FAILED [100%]
2  tests/test_predictions.py:16 (TestPredictions.test_predictions_returns_a_
```

39 Thanks to Javier Martínez Alcantara for elaborating on this example and sharing it in Episode 4 of the anti-pattern video series at Codurance.

```
    dataframe_with_automatic_predictions)
    ['id', 'quantity', 'form', 'country', 'order_id'] != ['id', 'quantity',
    'country', 'form', 'order_id']
3
4   Expected :['id', 'quantity', 'country', 'form', 'order_id']
5   Actual   :['id', 'quantity', 'form', 'country', 'order_id']
```

In this test case, the hint is that we would like to assert that the columns exist regardless of the order. In the end, what matters the most is having the column and the data for each column regardless of its order.

A better approach would be to replace line 2 previously depicted by the following assertion:

```
1   assert set(result.columns) == {"id", "quantity", "country", "form", "order_id"}
```

The sequencer is an anti-pattern that is not that often caught due to its nature of being easy to write, and the test suite is often in green; such anti-pattern is unveiled when someone has a hard time debugging the failure that is supposed to be passing.

Points of Attention

1. Know your data structures
2. Think about the role the order plays in a collection

The Nitpicker

A unit test compares a complete output when it's only interested in small parts of it. Hence, the test must continually be kept in line with otherwise unimportant details. Endemic in web application testing, Carr (2022).

The Nitpicker is covered in Episode 3 of the video[40] series covering the TDD anti-patterns hosted by Codurance.

As the definition goes, *The Nitpicker* is noted in web applications where the need to assert the output is focused on an entire object rather than the specific property needed. This is common for JSON structures, as depicted in the first example.

Laravel Assertions

The following code asserts that an application has been deleted. In this context, an application is a regular entry in the database with the label "application."

---

40 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-3.

Note that this example in PHP is used to assert the exact output from the HTTP request, nothing more, nothing less.

```php
<?php
public function testDeleteApplication()
{
    $response = $this->postApplication();

    $this->assertFalse($response->error);

    $this->delete('api/application/' . $response->data)
        ->assertExactJson([                    // is this needed?
        'data' => (string) $response->data,
        'error' => false
    ]);
}
```

In this sense, this test is fragile for a specific reason: if we add another property to the response it will fail to complain that the JSON has changed. For removing those properties, such failure would be helpful; on the other hand, adding a new property should not be the case.

The "fix" would be to replace the "exact" idea in this assertion to be less strict, such as the following:

```php
<?php
public function testDeleteApplication()
{
    $response = $this->postApplication();

    $this->assertFalse($response->error);

    $this->delete('api/application/' . $response->data)
        ->assertJson([                         // <!--- changing this assertion
        'data' => (string) $response->data,
        'error' => false
    ]);
    }
```

The change here is to assert that the desired fragment is indeed in the output, no matter if there are other properties in the output, as long as the desired one is there. This simple change opens the door to move away from the fragile test we had in the first place.

The AWS CloudFront URL Signature Utility Project

Another way to not face *The Nitpicker* is to only look for the properties you are concerned with. The following code is from an open-source project that handles the signing process to access a resource in Amazon S3[41]:

```
[caption=]
describe('#getSignedCookies()', function() {
  it('should create cookies object', function(done) {
    var result = CloudfrontUtil.getSignedCookies(
      'http://foo.com', defaultParams);

    expect(result).to.have.property('CloudFront-Policy');
    expect(result).to.have.property('CloudFront-Signature');
    expect(result).to.have.property('CloudFront-Key-Pair-Id');
    done();
  });
});
```

The code has three assertions to assert that it has the desired property instead of checking all at once, regardless of the output.

The Metrik Project

Another example of how to approach such assertion is the code extracted from an open source project that aims to collect and process the Four Key Metrics, Radziwill (2020) named Metrik:[42]

```
@Test
fun `should calculate CFR correctly by monthly and the time split works well (
cross a calendar month)`() {
    val requestBody = """ { skipped code } """.trimIndent()
    RestAssured
        .given()
        .contentType(ContentType.JSON)
        .body(requestBody)
        .post("/api/pipeline/metrics")
        .then()
        .statusCode(200)
        .body("changeFailureRate.summary.value", equalTo(30.0F))
```

41 The source code can be accessed at https://github.com/jasonsims/aws-cloudfront-sign/blob/master/test/lib/cloudfrontUtil.test.js#L235.

42 https://github.com/thoughtworks/metrik.

```
12          .body("changeFailureRate.summary.level", equalTo("MEDIUM"))
13          .body("changeFailureRate.details[0].value", equalTo("NaN"))
14          .body("changeFailureRate.details[1].value", equalTo("NaN"))
15          .body("changeFailureRate.details[2].value", equalTo(30.0F))
16  }
```

Once again, the *RestAssured*[43] framework is used to look for individual properties, in turn, within the output rather than using the entire object for comparison, as depicted in the first example.

Testing frameworks usually offer such utility to help practitioners to test their code in this manner. In the first example, the PHP framework Laravel uses the syntax *assertJson/assertExactJson*.[44]

The second example uses the testing library Chai to demonstrate how to assert specific properties within an object.

Last but not least, *RestAssured* is the library used to demonstrate how to deal with *the Nitpicker* within the Kotlin ecosystem.

Points of Attention

1. Assert against only those properties and values you are interested in
2. It can be generalized to other applications (e.g., CLI)

The Dodger

A unit test with lots of tests for minor (and presumably easy to test) side effects, but which never tests the core desired behaviour. Sometimes you may find this in database access related tests, where a method is called, then the test selects from the database and runs assertions against the result, Carr (2022).

The Dodger is covered in Episode 3 of the video[45] series covering the TDD anti-patterns hosted by Codurance.

The Dodger anti-pattern, is the most common anti-pattern when starting with a test first approach to software development. Before diving into the code example, let's elaborate a bit more on why *The Dodger* might appear.

Writing code in a TDD manner implies writing the test first, for any code you write. The rule is: start with a failing test, make it pass, and then refactor the design. As simple as it gets, there are some specific moments while practicing this flow that

---

43 https://rest-assured.io

44 https://laravel.com/docs/9.x/http-tests#verifying-exact-match

45 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-3

the question “what should I test” might arise.

As the rule goes, the common approach is to start writing tests for one class and one production class, meaning that it will have a 1-1 relationship. Then, the following question comes: “how small should the test scope be?”. As this “small” is context-dependent, it is not obvious what the smallest acceptable scope for a test should be.

Those two questions, while starting to practice TDD are common, and they might lead to *The Dodger* anti-pattern, as it is focused on testing specific implementation code rather than the desired behaviour,[46] to depict that take the following production code:

```php
<?php

namespace Drupal\druki_author\Data;

use Drupal\Component\Utility\UrlHelper;
use Drupal\Core\Language\LanguageManager;
use Drupal\Core\Locale\CountryManager;

/**
 * Provides author value object.
 */
final class Author {

  /** skipped protected properties to fit code here  */

  /**
   * Builds an instance from an array.
   *
   * @param string $id
   *   The author ID.
   * @param array $values
   *   The author information.
   */
  public static function createFromArray(string $id, array $values): self {
    $instance = new self();
    if (!\preg_match('/^[a-zA-Z0-9_-]{1,64}$/', $id)) {
      throw new \InvalidArgumentException('Author ID contains not allowed
```

46 In this (https://www.youtube.com/watch?v=APFbb5MwLEU) talk Mario Cervera elaborates on what behaviour is and how it applies to Test Driven Development.

```
        characters, please fix it.');
28    }
29    $instance->id = $id;
30
31    if (!isset($values['name']) || !\is_array($values['name'])) {
32      throw new \InvalidArgumentException("The 'name' value is missing or
        incorrect.");
33    }
34    if (\array_diff(['given', 'family'], \array_keys($values['name']))) {
35      throw new \InvalidArgumentException("Author name should contains 'given'
        and 'family' values.");
36    }
37    $instance->nameGiven = $values['name']['given'];
38    $instance->nameFamily = $values['name']['family'];
39
40    if (!isset($values['country'])) {
41      throw new \InvalidArgumentException("Missing required value 'country'.");
42    }
43    $country_list = \array_keys(CountryManager::getStandardList());
44    if (!\in_array($values['country'], $country_list)) {
45      throw new \InvalidArgumentException('Country value is incorrect. It should
        be valid ISO 3166-1 alpha-2 value.');
46    }
47    $instance->country = $values['country'];
48
49    if (isset($values['org'])) {
50      if (!\is_array($values['org'])) {
51        throw new \InvalidArgumentException('Organization value should be an
          array.');
52      }
53      if (\array_diff(['name', 'unit'], \array_keys($values['org']))) {
54        throw new \InvalidArgumentException("Organization should contains 'name'
          and 'unit' values.");
55      }
56      $instance->orgName = $values['org']['name'];
57      $instance->orgUnit = $values['org']['unit'];
58    }
59
60    if (isset($values['homepage'])) {
61      if (!UrlHelper::isValid($values['homepage']) ||
        !UrlHelper::isExternal($values['homepage'])) {
62        throw new \InvalidArgumentException('Homepage must be valid external
```

```
          URL.');
63      }
64      $instance->homepage = $values['homepage'];
65    }
66
67    if (isset($values['description'])) {
68      if (!\is_array($values['description'])) {
69        throw new \InvalidArgumentException('The description should be an array
          with descriptions keyed by a language code.');
70      }
71      $allowed_languages = \array_
        keys(LanguageManager::getStandardLanguageList());
72      $provided_languages = \array_keys($values['description']);
73      if (\array_diff($provided_languages, $allowed_languages)) {
74        throw new \InvalidArgumentException('The descriptions should be keyed by
          a valid language code.');
75      }
76      foreach ($values['description'] as $langcode => $description) {
77        if (!\is_string($description)) {
78          throw new \InvalidArgumentException('Description should be a
            string.');
79        }
80        $instance->description[$langcode] = $description;
81      }
82    }
83
84    if (isset($values['image'])) {
85      if (!\file_exists($values['image'])) {
86        throw new \InvalidArgumentException('The image URI is incorrect.');
87      }
88      $instance->image = $values['image'];
89    }
90
91    if (isset($values['identification'])) {
92      if (isset($values['identification']['email'])) {
93        if (!\is_array($values['identification']['email'])) {
94          throw new \InvalidArgumentException('Identification email should be an
            array.');
95        }
96        $instance->identification['email'] = $values['identification']['email'];
97      }
98    }
```

```
99
100     return $instance;
101   }
102
103   public function getId(): string {
104     return $this->id;
105   }
106
107   public function getNameFamily(): string {
108     return $this->nameFamily;
109   }
110
111   public function getNameGiven(): string {
112     return $this->nameGiven;
113   }
114
115   public function getCountry(): string {
116     return $this->country;
117   }
118
119   public function getOrgName(): ?string {
120     return $this->orgName;
121   }
122
123   public function getOrgUnit(): ?string {
124     return $this->orgUnit;
125   }
126
127   public function getHomepage(): ?string {
128     return $this->homepage;
129   }
130
131   public function getDescription(): array {
132     return $this->description;
133   }
134
135   public function getImage(): ?string {
136     return $this->image;
137   }
138
139   public function checksum(): string {
140     return \md5(\serialize($this));
```

```
141   }
142
143   public function getIdentification(?string $type = NULL): array {
144     if ($type) {
145       if (!isset($this->identification[$type])) {
146         return [];
147       }
148       return $this->identification[$type];
149     }
150     return $this->identification;
151   }
152 }
```

The goal is to validate the Author object before creating it from an array. To be created, the given array should hold valid data; if it does not, an exception will be thrown. Then, next up is the testing code:

```
1   <?php
2   /**
3    * Tests that objects works as expected.
4    */
5   public function testObject(): void {
6     $author = Author::createFromArray($this->getSampleId(), $this-
      >getSampleValues());
7     $this->assertEquals($this->getSampleId(), $author->getId());
8     $this->assertEquals($this->getSampleValues()['name']['given'], $author-
      >getNameGiven());
9     $this->assertEquals($this->getSampleValues()['name']['family'], $author-
      >getNameFamily());
10    $this->assertEquals($this->getSampleValues()['country'], $author-
      >getCountry());
11    $this->assertEquals($this->getSampleValues()['org']['name'], $author-
      >getOrgName());
12    $this->assertEquals($this->getSampleValues()['org']['unit'], $author-
      >getOrgUnit());
13    $this->assertEquals($this->getSampleValues()['homepage'], $author-
      >getHomepage());
14    $this->assertEquals($this->getSampleValues()['description'], $author-
      >getDescription());
15    $this->assertEquals($this->getSampleValues()['image'], $author->getImage());
16    $this->assertEquals($this->getSampleValues()['identification'], $author-
      >getIdentification());
```

```
17    $this->assertEquals($this->getSampleValues()['identification']['email'],
      $author->getIdentification('email'));
18    $this->assertEquals([], $author->getIdentification('not exist'));
19    $this->assertEquals($author->checksum(), $author->checksum());
20  }
```

The first thing that is noticed, when skimming through the code, is that if you need to change how you get the author name (rename the method, for example) you also need to change the test code, even though the desired behaviour hasn't changed – the validation (the current behaviour) is still required.

An alternative approach would be to break out the single test case, into multiple test cases, catching the desired exception if an undesired value is passed, then encapsulate it in a validator class to prevent the coupling from the test and production code.

Points of Attention

1. 1-1 relationship between test class and one production class
2. Focus on testing behaviour rather than specific implementation details

The Liar

An entire unit test that passes all of the test cases it has and appears valid, but upon closer inspection, it is discovered that it doesn't really test the intended target at all, Carr (2022).

The Liar is covered in Episode 1 of the video[47] series covering the TDD anti-patterns hosted by Codurance.

The Liar is one of the most common anti-patterns due to its nature of being hidden in the source code. It reveals itself only on closer inspection. There are at least those two reasons to spot such issues among codebases:

1. Async-oriented test cases
2. Time-oriented test cases

The first one is well explained in the Jest official documentation.[48] Testing asynchronous code becomes tricky as it is based on a future value you may or may not receive (jestjs.io 2021). The following code is a reproduced example from jest official documentation (the docs state even in a code comment to not use the following test code in real projects).

---

47 https://www.codurance.com/publications/tdd-anti-patterns-chapter-1

48 The example used here is from jest but it can be found in other test frameworks.

Async Test with Jest

```
// Don't do this!
test('the data is peanut butter', () => {
  function callback(data) {
    expect(data).toBe('peanut butter');
  }

  fetchData(callback);
  });
```

Getting back to *The Liar* anti-pattern, this test would pass without complaint, even though that pass would actually be a lie, due to how the test is written. The correct approach is to wait for the async function to finish its execution and give Jest control over the flow execution again.

```
test('the data is peanut butter', done => {
  function callback(data) {
    try {
      expect(data).toBe('peanut butter');
      done(); // invokes jest flow again, saying: "look I am ready now!"
    } catch (error) {
      done(error);
    }
  }

  fetchData(callback);
});
```

In the second one, Martin Fowler elaborates on the reasons for that to be the case (Fowler 2011), and here let's share some opinions that go along with what he wrote.

Asynchronous is a source of non-determinism; we should be careful with that, as already depicted in the previous Jest example. Besides that, threads in test code deserve special care as well. If you need to handle them, ensure they are working as expected.

On the other hand, time-oriented tests sometimes can fail, seemingly for no obvious reason, if no proper handling is used to control that. Therefore, practitioners tend to adopt test doubles to handle dates in a way that they can control without being coupled to a real-time. This avoids the situation where on the day that the code was written, the test was passing, but on the following day, it broke.

Points of Attention

1. Async tests can mislead the test result, watch out for specific test runners.

The Loudmouth

A unit test (or test suite) that clutters up the console with diagnostic messages, logging messages, and other miscellaneous chatter, even when tests are passing. Sometimes during test creation there was a desire to manually see output, but even though it's no longer needed, it was left behind, Carr (2022).

The Loudmouth is covered in Episode 3 of the video[49] series covering the TDD anti-patterns hosted by Codurance.

When developing, it is common to add some temporary traces to the code to help a developer confirm whether or not the code is behaving as expected. This process is often referred to as debugging, often used when developers need to clarify their understanding of a piece of code.

TDD practitioners argue that once TDD is practiced, no debugging tool is needed, Freeman and Pryce (2009), be it a print statement or be it adding breakpoints into the code.

But, what happens if you don't have that much experience with TDD?

The Testable Project

Often the answer is a mix of both debugging and using the tests to guide you. For example, the following code depicts some test code that can handle an error if it receives an invalid piece of JavaScript code. Keep in mind that the code is used to parse JavaScript code and act upon its result:

```
1  test.each([['function']])(
2    'should not bubble up the error when an invalid source code is provided,
3    (code) => {
4      const strategy = jest.fn();
5
6      const result = Reason(code, strategy);
7      expect(strategy).toHaveBeenCalledTimes(0);
8      expect(result).toBeFalsy();
9    }
10 );
```

The check is straightforward. It ensures that the desired strategy was not called

49 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-3

as the code is an invalid piece of JavaScript code. It also checks whether the result from that was a Boolean false value. Let's see now what the implementation of this test looks like:

```
const Reason = function(code, strategy) {
  try {
    const ast = esprima.parseScript(code);

    if (ast.body.length > 0) {
      return strategy(ast);
    }
  } catch (error) {
    console.warn(error);           // < ---------- this is loud
    return false;
  }
};
```

Ideally, working in a TDD fashion, the console.log statement used would be mocked from the start. This is because it would require verification of when it was called and with which message. This first hint already points to an approach that is not tested first. The following image depicts what The Loudmouth anti-pattern causes. Even though the tests are green, there is a warning message – did the test pass? Did the change break something?

Freeman and Pryce (2009) gives an idea of why logging (such as this console.log) should be treated as a feature instead of a random log used for whatever reason.

The following snippet depicts a possible implementation mocking out the console.log and preventing the message from being displayed during test execution:

```
const originalConsole = globalThis.console;

beforeEach(() => {
  globalThis.console = {
    warn: jest.fn(),
    error: jest.fn(),
    log: jest.fn()
  };
});

afterEach(() => {
  globalThis.console = originalConsole;
});
```

With the mocked console, now it's possible to assert its usage of it instead of printing the output while running the tests, the version without *The Loudmouth* would be the following:

```
const originalConsole = globalThis.console;

beforeEach(() => {
  globalThis.console = {
    warn: jest.fn(),
    error: jest.fn(),
    log: jest.fn()
  };
});

afterEach(() => {
    globalThis.console = originalConsole;
});

test.each([['function']])(
  'should not bubble up the error when an invalid source code is provided',
  (code) => {
    const strategy = jest.fn();

    const result = Reason(code, strategy);
    expect(strategy).toHaveBeenCalledTimes(0);
    expect(result).toBeFalsy();
    expect(globalThis.console.warn).toHaveBeenCalled(); // < -- did it warn?
  }
);
```

The Loudmouth anti-pattern can potentially cause to the developer to question whether the test is passing for the right reason. This is because the additional logging output pollutes the testing output while the tests are being executed.

Points of Attention

1. Clean up!
2. Threat the logs as a feature, test drive them.

Level II

Level II

Level II targets more of an intermediate skill level but is still relevant to beginners, as we progress in the test-driven approach, things start to get blurry in the sense that we already know what to test and we might also feel comfortable setting up any kind of environment for testing, but due to the lack of confidence, some principles are lost.

Due to such progress, we might see test cases that should be red, but their result is green.

Throughout this level, you will also see subjects related to:

- Avoid writing a test that passes first.
- Avoid digging into other object implementations to set up a test case.
- When a test fails and it is difficult to spot the root cause, you might face a hidden dependency.
- Avoid catching exceptions just to make a test pass.
- Avoid sharing state between tests whenever possible.
- Avoid relying on exceptions to make the test pass instead, make assertions explicit.

The Success Against All Odds

A test that was written pass first rather than fail first. As an unfortunate side effect, the test case happens to always pass even though the test should fail, Carr (2022).

The Success Against All Odds is covered in Episode 5 of the video[50] series covering the TDD anti-patterns hosted by Codurance.

The Success Against All Odds is an anti-pattern that is related to the lack of a test-first approach, but instead, the practitioner follows the test first, and instead of failing first, it just makes the test pass from the start, except for the first test in the test class.

When this is the case, *The Success Against All Odds* is revealed. The practice of starting the test passing first leads to the test passing even when the failure is expected.

To depict such a scenario, the following snippet is an attempt to implement a repository from SpringBoot that will paginate and query based on a given query string.

---

50 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-5

Note: For the sake of the example, the teardown has been removed to keep it simple. The tear-down removes all the data inserted in the database used during the test.

```
@Repository
class ProductsRepositoryWithPostgres(
    private val entityManager: EntityManager
) : Repository {

    override fun listFilteredProducts(query: String?, input: PagingQueryInput?)
{
        val pageRequest: PageRequest = input.asPageRequest()
        val page: Page<Product> = if (query.isNullOrBlank()) {
            entityManager.findAll(pageRequest)
        } else {
            entityManager.findAllByName(query, pageRequest)
        }
        return page
    }
}
```

Once we look at the given code that performs the access to the database and apply the criteria, the following test code is used to test the repository.

Since the beginning, we have been doing some heavy lifting operations to populate the database with different data. Which could potentially be a code smell.

```
private fun setupBeforeAll() {
    productIds = (1..100).map { db().productWithDependencies().apply().
get<ProductId>() }
    productIdsContainingWood.addAll(
        (1..3).map { insertProductWithName("WoodyWoodOrange " + faker.
funnyName().name()) }
    )
    productIdsContainingWood.addAll(
        (1..3).map {
            insertProductWithName(
                faker.funnyName().name() + " WoodyWoodOrange " + faker.
funnyName().name()
            )
        }
    )
}
```

With the setup in place, let's look at the first test case in this class. The goal of the test case is to test that given a sort parameter, the parameter CREATED_AT_ASC (line 4) is the one we are looking for, once this has been given, the data should be ordered accordingly.

```
1  @Test
2  fun `list products sorted by creation at date ascending`() {
3      val pageQueryInput = PagingQueryInput(
4          size = 30, page = 0, sort = listOf(Sort.CREATED_AT_ASC)
5      )
6      val result = repository.listFilteredProducts("", pageQueryInput)
7
8      assertThat(result.currentPage).isEqualTo(0)
9      assertThat(result.totalPages).isEqualTo(4)
10     assertThat(result.totalElements).isEqualTo(112)
11
12     assertThat(result.content.size).isEqualTo(30)
13     assertThat(result.content).allSatisfy { productIds.subList(0, 29).
       contains(it.id) }
14 }
```

Let's dive a bit into what is going on in the code guided by the line numbers there:

1. Line 3, 4 and 5: The parameter that we send to the repository with the order we want and pagination
2. Line 6: The execution of the code we want to test
3. Line 8: We verify that the page returned from the repository is the first one
4. Line 9: We verify that there are 4 pages in total
5. Line 10: We verify that there are 112 in total
6. Line 12: We verify that the list of items returned is the same as the one asked in the pagination
7. Line 13: We verify that the list returned is the same as in the list created in the setup before all

The next text case depicts a variant of what we might want to test, which is the reverse order. Instead of ascending order, we will now test in descending order. Note that most of the assertions are the same if we compare them with the ones from the previous test case (from lines 6 through 14).

```
1  @Test
2  fun `list products sorted by creation at date ascending`() {
3      val pageQueryInput = PagingQueryInput(
4          size = 30, page = 0, sort = listOf(Sort.CREATED_AT_DESC)
5      )
```

```
6       val result = repository.listFilteredProducts("", pageQueryInput)
7
8       assertThat(result.currentPage).isEqualTo(0)
9       assertThat(result.totalPages).isEqualTo(4)
10      assertThat(result.totalElements).isEqualTo(112)
11
12      assertThat(result.content.size).isEqualTo(30)
13      assertThat(result.content).allSatisfy { productIds.subList(0, 29).
        contains(it.id) }
14  }
```

Let's avoid repeating the previous bullet point list and focus on the important items.

The first important aspect is the number of assertions we might not need for each test case. For example, from 8 through 12, some assertions verify the pagination and the numbers related to the list, reading the test name. Our goal is to test the sorting first and not the pagination functionality. In other words, we could have used just the last assertion for this test.

Moving on, let's dive into line 13 a bit more. Having an assertion such as the one here is one of the possible causes of *The Success Against All Odds*, and actually, in the test code, is one of them, as it asserts on a subset of the list that will always be true.

In the xUnit Test Patterns book, a way to avoid such false/positive behaviour is to have the code as simple as possible, with no logic in it.[51] This is called the robust test Meszaros (2007).

Refactoring Success Against All Odds

The question here is, what could we do, as an alternative, in order to avoid such a thing? A possible solution for this test case and source code is related to splitting responsibilities in the test case. We could focus on sorting only and test the pagination in a subsequent later test.

The first example here would be ordering the list in ascending order, it is worth mentioning that with this approach, we could potentially remove the big setup shown previously in the hook *setupBeforeAll*. For this approach, we instead set up the data that is required for the test inside it. No more shared state between tests.

---

51 The Free Ride depicted in section 8.4 also uses logic inside test cases.

```
@Test
fun `list products sorted by ascending creation date`() {
    db().productWithDependencies("created_at" to "2022-04-03T00:00:00.00Z").
    apply() // 1
    db().productWithDependencies("created_at" to "2022-04-02T00:00:00.00Z").
    apply() // 2
    db().productWithDependencies("created_at" to "2022-04-01T00:00:00.00Z").
    apply() // 3

    val pageQueryInput = PagingQueryInput(sort = listOf(SortOrder.CREATED_AT_
    ASC))

    val result = repository.listFilteredProducts("", pageQueryInput)

    assertThat(result.content[0].createdAt).isEqualTo("2022-04-01T00:00:00.00Z")
    assertThat(result.content[1].createdAt).isEqualTo("2022-04-02T00:00:00.00Z")
    assertThat(result.content[2].createdAt).isEqualTo("2022-04-03T00:00:00.00Z")
}
```

Once that is in place, we then move to the descending-order test case, which is the same, but the assertion and setup changed:

```
@Test
  fun `list products sorted by creation at date descending`() {
      db().productWithDependencies("created_at" to "2022-04-01T00:00:00.00Z").
      apply()
      db().productWithDependencies("created_at" to "2022-04-02T00:00:00.00Z").
      apply()
      db().productWithDependencies("created_at" to "2022-04-03T00:00:00.00Z").
      apply()

      val pageQueryInput = PagingQueryInput(sort = listOf(SortOrder.CREATED_AT_
      DESC))

      val result = repository.listFilteredProducts("", pageQueryInput)

      assertThat(result.content[0].createdAt).isEqualTo("2022-04-
      03T00:00:00.00Z")
      assertThat(result.content[1].createdAt).isEqualTo("2022-04-
      02T00:00:00.00Z")
      assertThat(result.content[2].createdAt).isEqualTo("2022-04-
      01T00:00:00.00Z")
  }
```

Next up, is the pagination, now we can start to focus on the pagination and the aspects it brings.

Once we have the sorting in place, we can start to have a look at the pagination, and of course, try to test a specific thing at a time. The following example depicts how we could assert that we got the desired number of pages when paging the result.

```
@Test
  fun `should have one page when the list is ten`() {
      insertTenProducts()
      val page = PagingQueryInput(size = 10)

      val result = repository.listFilteredProducts(
          null,
          null,
          Page
      )

      assertThat(result.totalPages).isEqualTo(1)
  }
```

The approach to decomposing the tests into smaller “units”[52] would help the communication between the team members dealing with this code later on and make these tests more robust.

Points of Attention

1. Start with the test in red whenever possible.
2. Avoid repeating the same assertions from previous test cases.
3. Avoid sharing state between test cases.

The Stranger

A test case that doesn’t even belong in the unit test, it is part of. It is really testing a separate object, most likely an object that is used by the object under test, but the test case has gone and tested that object directly without relying on the output from the object under test, making use of that object for its own behaviour. Also known as *TheDistantRelative,* Carr (2022).

The Stranger is covered in Episode 5 of the video[53] series covering the TDD anti-patterns hosted by Codurance.

---

52 Note that here the unit does not refer to a function or method but rather to behaviour (or responsibility if you will).

53 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-5

Let's start with an introduction, in this blog post from java revisited[54], the way that the law of Demeter is explained gives us a hint on why *The Stranger* is an anti-pattern. We can also relate this to the book Clean Code, which recommends "talk to friends, not to strangers" in designing code. In the example given in the book, the method chain is the one that exposes more to *The Stranger.* The example is being used in production code.

Carlos Caballero, in his blog post, *Demeter's Law: Don't talk to strangers!*[55] also uses production code to depict an example violation of the law. He provides a code snippet that ideally would need to be tested. It is at this point that we shall expand further on and specifically, implement the supporting test code.

To start with here is the code that depicts the law of Demeter violation within the production code:

```
person
    .getHouse() // return an House's object
    .getAddress() // return an Address's object
    .getZipCode() // return a ZipCode Object
```

Such code could potentially lead to *The Stranger* anti-pattern within the test code. For example, to test if the given person has a valid zip code, we could potentially write something like this:

```
describe('Person', () => {
    it('should have valid zip code', () => {
        const person = FakeObject.createAPerson({ zipCode: '56565656' });
        person
            .getHouse()
            .getAddress()
            .getZipCode()
        expect('56565656').toEqual(person.house.address.zipCode);
    });
});
```

Note that if we want to access the zip code, we need to go all the way down to the ZipCode object. This provides a hint that in fact what we want to test is the Address object itself and not Person.[56]

---

54 https://javarevisited.blogspot.com/2014/05/law-of-demeter-example-in-java.html

55 https://betterprogramming.pub/demeters-law-don-t-talk-to-strangers-87bb4af11694

56 The idea here is not to bring the topic around 1-1 relationship between production code and test code, but rather to depict a specific scenario in which we are breaking encapsulation as well.

```
describe('Address', () => {
    it('should have valid zip code', () => {
        const address = new Address(
            '56565656',
            '1456',
            'Street X',
            'My city',
            'Great state',
            'The best country'
        );
        expect('56565656').toEqual(address.getZipCode());
    });
});
```

The test itself has something here that could be improved to avoid this. For example, the interaction between the Person object, Address and Zip code could be "hidden" within an implementation behind an abstraction. In this case, we would then test the output of that, rather than directly navigating all the way down the object graph.

Before moving on to the next anti-pattern, remember that *The Stranger* could also be categorized as test smell. Here are some signs that could lead to *The Stranger*:

1. It is related to the xUnit (Meszaros 2007) pattern in the section "Test smells"
2. The usage of mocking

The Hidden Dependency

A close cousin of *The Local Hero*, a unit test that requires some existing data to have been populated somewhere before the test runs. If that data wasn't populated, the test would fail and leave little indication to the developer what it wanted or why... forcing them to dig through acres of code to find out where the data it was using was supposed to come from (Carr 2022).

The Hidden Dependency is covered in Episode 4 of the video[57] series covering the TDD anti-patterns hosted by Codurance.

The Hidden Dependency is an anti-pattern which is popular among practitioners. In particular, *The Hidden Dependency* annoys and makes practitioners unhappy about testing in general. It can be the source of hours debugging test code in

57 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-4

an attempt to understand why a test is failing. Sometimes it gives little to no information about the root cause. This issue is related to the following:

- Databases (embedded databases to run tests)
- Builders (complex logic to build data to set up a test case)

In the next section, we will go over the Vuex state management library that hides the complexity to handle data for frontend applications. If you are unfamiliar with Vuex[58] or the Flux pattern, it is recommended to check it out first.

The Vuex Dependency

The example in this section is related to Vue and Vuex, in this test case, the goal is to list users in a dropdown. Vuex is used as a source of truth for the data.

```
1  export const Store = () => ({
2    modules: {
3      user: {
4        namespaced: true,
5        state: {
6          currentAdmin: {
7            email: 'fake@fake.com',
8          },
9        },
10       getters: userGetters,
11     },
12     admin: adminStore().modules.admin,
13   },
14 });
```

On line 2, the structure needed for Vuex is defined, and on line 12 the admin store is created. Once the stubbed store is in place, we can start to write the test itself. As a hint for the next piece of code, note that the store has no parameters.

```
1  it('should list admins in the administrator field to be able to pick one up',
   async () => {
2    const store = Store();
3
4    const { findByTestId, getByText } = render(AdminPage as any, {
5      store,
6      mocks: {
7        $route: {
```

58 https://vuex.vuejs.org

```
8           query: {},
9         },
10      },
11    });
12    await fireEvent.click(await findByTestId('admin-list'));
13    await waitFor(() => {
14      expect(getByText('Admin')).toBeInTheDocument();
15    });
16  });
```

On line 2 we create the store to use in the code under test, and on line 14, we try to search the text Admin. We assume the list is working if it is in the text.

The catch here is that if the test fails to find the Admin, we will need to dive into the code inside the store to see what is going on, as you might have noticed, if we look at only the test case, it is not clear where the text Admin comes from.

The next code example shows a better approach to explicitly using the data needed when setting up the test. This time on line 2, the Admin is expected to exist beforehand.

```
1   it('should list admins in the administrator field to be able to pick one up',
    async () => {
2     const store = Store({ admin: { name: 'Admin' } });
3
4     const { findByTestId, getByText } = render(AdminPage as any, {
5       store,
6       mocks: {
7         $route: {
8           query: {},
9         },
10      },
11    });
12
13    await fireEvent.click(await findByTestId('admin-list'));
14
15    await waitFor(() => {
16      expect(getByText('Admin')).toBeInTheDocument();
17    });
18  });
```

In general, *The Hidden Dependency* can appear in different ways and different styles of tests. The next section depicts a hidden issue that comes from testing

the integration with a database.

The Database Dependency

Here we are trying to fetch manual purchases from the database based on a given criterion that is implemented behind the method **get_manual_purchases.** Then we compare the output from the method with the desired outcome that is stored in a CSV file.

```
1  def test_dbdatasource_is_able_to_load_products_related_only_to_manual_purchase(
2     self, db_resource
3  ):
4     config_file_path = Path("./tests/data/configs/docker_config.json")
5     expected_result = pd.read_csv("./tests/data/manual_product_info.csv")
6
7     datasource = DBDataSource(config_file_path=config_file_path)
8
9     result = datasource.get_manual_purchases()
10
11    assert result.equals(expected_result)
```

On lines 4 and 5 the setup is done via configuration files, line 5 is important as the result from the test should match its content. Then on line 9, the code under test is exercised.

Inside this method, there is a query that is executed in order to fetch the manual purchases and assert that it is the same as the expected result:

```
1  query: str = """
2     select
3   product.id
4         po.order_id,
5         po.quantity,
6         product.country
7     from product
8     join purchased as pur on pur.product_id = product.id
9     join purchased_order as po on po.purchase_id = cur.id
10    where product.completed is true and
11    pur.type = 'MANUAL' and
12    product.is_test is true
13    ;
14 """
```

This query has a particular where clause that is hidden from the test case, thus making the test fail. By default, the data generated from the expected result set the flag test to false, leading to no results returning in the test case.

Points of Attention

1. Test data integration as soon as possible.
2. If possible, avoid using data from external sources within tests.

The Greedy Catcher

A unit test which catches exceptions and swallows the stack trace, sometimes replacing it with a less informative failure message, but sometimes even just logging (c.f. Loudmouth) and letting the test pass, Carr (2022).

The Greedy Catcher is covered in Episode 4 of the video[59] series covering the TDD anti-patterns hosted by Codurance.

Handling exceptions (or even using them) can be tricky. Some practitioners advocate for not using exceptions at all[60]; others use them as a mechanism to inform that something went wrong during the execution of the program.

Despite the kind of developer you are, testing for exceptions can unveil some patterns that hurt the test-first approach.[61]

The Greedy Catcher appears when the subject under test handles the exception and hides useful information regarding the kind of exception, message or stack trace. Such information is helpful to mitigate possible undesirable exceptions being thrown.

Next, we have an example from a possible candidate for *The Greedy Catcher*. This piece of code was extracted from the project *Laravel/Cashier stripe – Laravel* it is written in PHP and is one of the most popular projects within the PHP ecosystem.

The following code is a package that wraps the Stripe SDK[62] into a Laravel package that offers an easier approach to integrating stripe into Laravel applications.

---

59 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-4

60 Here is a StackOverflow thread that discusses the subject in depth https://stackoverflow.com/questions/1736146/why-is-exception-handling-bad

61 The Greedy Catcher and The Secret Catcher are both similar but cover different aspects of exceptions. In section 9.6 we cover in detail The Secret Catcher.

62 Stripe Software Development Kit – https://stripe.com/docs/development/quickstart/php

The Laravel/Cashier Stripe Project

Despite having a try/catch handler inside the test case(that could potentially point to further improvements) when the exception is being thrown, the test case catches it and asserts some logic:

```
1  public function test_retrieve_the_latest_payment_for_a_subscription()
2  {
3      $user = $this->createCustomer('retrieve_the_latest_payment_for_a_
       subscription');
4
5      try {
6          $user->newSubscription('main', static::$priceId)
7            ->create('pm_card_threeDSecure2Required');
8
9          $this->fail('Expected exception '.IncompletePayment::class.' was not
           thrown.');
10     } catch (IncompletePayment $e) {
11         $subscription = $user
12           ->refresh()
13           ->subscription('main');
14
15         $this->assertInstanceOf(
16           Payment::class,
17           $payment = $subscription->latestPayment()
18         );
19         $this->assertTrue($payment->requiresAction());
20     }
21 }
```

The Greedy Catcher arises not only in the test code but also in the production code, hiding useful information for tracing back an exception is a source of time spent that could have been saved if the code had been written differently.

In the next section, we will see an example of how *The Greedy Catcher* anti-pattern can also be found within production code.

Parsing the JTW token with JavaScript

The following example is a representation of a JavaScript middleware component that parses a JWT token and redirects the user if the token is empty. The code uses the libraries Jest and Nuxtjs.

Line 3 decoded the JWT token via jwt-decode package, in the case of success,

the middleware follows the flow. If the token is false for any reason, it invokes the logout function (on lines 8 and 11).

As far as the code looks, it is difficult to recognize that under the catch block, the exception is being ignored and if something happens, the result will be what the logout function returns (line 11).

```
1  export default function(context: Context) {
2    try {
3      const token = jwt_decode(req?.cookies['token']);
4
5      if (token) {
6        return null;
7      } else {
8        return await logout($auth, redirect);
9      }
10   } catch (e) {
11     return await logout($auth, redirect);
12   }
13 }
```

The test code uses some Nuxtjs context to create the request that is going to be processed by the middleware. The single test case depicts an approach to verify if the user is being logged out or if the token is invalid. Note that the cookie is behind the *serverParameters* variable.

```
1  it('should log out when token is invalid', async () => {
2    const redirect = jest.fn();
3    const serverParameters: Partial<IContextCookie> = {
4      route: currentRoute as Route, $auth, redirect, req: { cookies: null },
5    };
6
7    await actions.nuxtServerInit(
8      actionContext as ActionContext,
9      serverParameters as IContextCookie
10   );
11
12   expect($auth.logout).toHaveBeenCalled();
13 });
```

The tricky part is that the test above passes as it should, but not for the expected reason. *serverParameters* holds the req object that has cookies set to null (line 3). When that is the case, JavaScript will throw an error as it will not be possible to

access a token of null.[63]

Such behaviour executes the catch block, which calls the desired logout function (line 12). The stack trace for this error will not show up in any place, as the catch block ignores the exception in the production code.

Points of Attention

1. Hiding information in try/catch blocks makes it harder to spot issues in production code.
2. Spending time to understand why the test is not giving the desired behaviour.

The Peeping Tom

A test that, due to shared resources, can see the result data of another test, and may cause the test to fail even though the system under test is perfectly valid. This has been seen commonly in fitness, where the use of static member variables to hold collections aren't properly cleaned after test execution, often popping up unexpectedly in other test runs. Also known as *TheUninvitedGuests* (Carr 2022).

The Peeping Tom is covered in Episode 6 of the video[64] series covering the TDD anti-patterns hosted by Codurance.

Having to deal with any global state within a test case is something that brings an extra layer of complexity. It requires explicit cleanup steps before each test and even after each test case is executed to avoid side effects.

The Peeping Tom depicts the issue faced when using any form of global state during test execution. Within the popular question-and-answer website Stackoverflow, there is a thread dedicated to this subject that has a few comments which help in understanding this better. Christian Posta also blogged about static methods being code smells.[65]

In there, there is a snippet that was extracted from this blog post that depicts how the use of *Singleton* (Gamma et al. 1994) and static properties can harm the test case and keep the state between tests. Here, we are going to use the same example with minor changes to make the code compile.

The idea behind the *Singleton* is to create and reuse a single instance from any

---

63 JavaScript will behave like that for an attempt to access any property from a variable that is null or undefined.

64 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-6

65 The blog post can be found at https://blog.christianposta.com/testing/java-static-methods-can-be-a-code-smell.

kind of object.[66] So to achieve that, we can create a class (in this example called *MySingleton*) and block the creation of an object through its constructor and allow only the creation inside the class, controlled by the method getInstance:

```
1  public class MySingleton {
2      private static MySingleton instance;
3      private String property;
4
5      private MySingleton(String property) {
6          this.property = property;
7      }
8
9      public static synchronized MySingleton getInstance() {
10         if (instance == null) {
11             instance = new MySingleton(System.getProperty("com.example"));
12         }
13         return instance;
14     }
15
16     public Object getSomething() {
17         return this.property;
18     }
19 }
```

When it comes to testing, there is not much in the way of a public interface for us to interact with. However, the method exposed in the MySingleton called *getSomething* can be invoked and asserted against a value as shown in the following snippet:

```
1  import org.junit.jupiter.api.Test;
2
3  import static org.assertj.core.api.Assertions.assertThat;
4
5  class MySingletonTest {
6      @Test
7      public void somethingIsDoneWithAbcIsSetAsASystemProperty(){
8          System.setProperty("com.example", "abc");
9          MySingleton singleton = MySingleton.getInstance();
10         assertThat(singleton.getSomething()).isEqualTo("abc");
11     }
12
13 }
```

66 Rainer Grimm listed the advantages and disadvantages of the Singleton, in there, he listed global access as an advantage, therefore, he also mentioned the issue regarding testability which he relates to The Hidden Dependency that we discuss in Chapter 9.3.

A single test case will pass without any problem, the test case creates the *Singleton* instance and invokes the *getSomething* to retrieve the property value defined when the test was defined. The issue arises when we try to test the same behaviour but with different values in the *System.setProperty.*

```
1  import org.junit.jupiter.api.Test;
2
3  import static org.assertj.core.api.Assertions.assertThat;
4
5  class MySingletonTest {
6      @Test
7      public void somethingIsDoneWithAbcIsSetAsASystemProperty(){
8          System.setProperty("com.example", "abc");
9          MySingleton singleton = MySingleton.getInstance();
10         assertThat(singleton.getSomething()).isEqualTo("abc");
11     }
12
13     @Test
14     public void somethingElseIsDoneWithXyzIsSetAsASystemProperty(){
15         System.setProperty("com.example", "xyz");
16         MySingleton singleton = MySingleton.getInstance();
17         assertThat(singleton.getSomething()).isEqualTo("xyz");
18     }
19 }
```

Given the nature of the code, the second test case will fail and shows that it still holds the value abc.

As the *Singleton* guarantees only one instance from a given object, during the test execution, the first test that is executed creates the instance and, for the following executions, reuses the same instance previously created.

The problem here is "easy" to see as one test case is executed after the other. But, for testing frameworks that execute tests in parallel or that do not guarantee the test order (which is often the default behaviour), it can be much for difficult to identify test failures caused by a dependency on a global state that has been mutated by other tests.

It can confuse practitioners because the test will pass when it is run alone and fail when running together with others.

As the *Singleton* class has a private property that controls the instance created, it is not possible to clean it without changing the code itself (which would be a

change just for testing purposes). Therefore, another approach would be to use reflection to reset the property and always start with a fresh instance, as the following code depicts:

```
class MySingletonTest {

    @BeforeEach
    public void resetSingleton() throws SecurityException, NoSuchFieldException,
    IllegalArgumentException, IllegalAccessException {
        Field instance = MySingleton.class.getDeclaredField("instance");
        instance.setAccessible(true);
        instance.set(null, null);
    }
}
```

Making use of reflection, it is possible to reset the instance before each test case is executed (using the annotation @BeforeEach). Although this approach is possible, it should bring attention to the extra code and possible side effects while testing the application using such a pattern.

Depicting *The Peeping Tom* like that, besides having to use reflection to reset a property, might not seem harmful to test drive code, but it can become even harder when a piece of code that we want to test depends on a *Singleton*.

As shared by Rodaney Glitzel (2022), the problem is not Singleton itself but a code that depends on that, doing so, the code that depends on that becomes harder to test.

The Secret Catcher

A test that at first glance appears to be doing no testing due to the absence of assertions, but as they say, "the devil's in the details." The test relies on an exception to be thrown when a mishap occurs and expects the testing framework to capture the exception and report it to the user as a failure, Carr (2022).

The Secret Catcher is covered in Episode 3 of the video[67] series covering the TDD anti-patterns hosted by Codurance.

The Secret Catcher is a practitioner's old friend. It can be spotted in different codebases. This anti-pattern often relates to the "hurry" in which features need to be released or even the desire to achieve X% of test coverage.

---

67 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-3.

This anti-pattern has *secret* within its name because the code doesn't make it clear that it is supposed to throw an exception. Instead of handling (or catching) said exception, the test case instead simply ignores it. *The Secret Catcher* is also related to *The Greedy Catcher*.

The following example written in Vue.js with an Apollo client is an attempt to depict such a scenario. At first glance, the method seems fine and does what it is supposed to. In other words, it sends a mutation operation to the server to remove the payment type from the associated user, and, in the end, it updates the UI to reflect that:

```
async removePaymentMethod(paymentMethodId: string) {
  this.isSavingCard = true;

  const { data } = await this.$apolloProvider.clients.defaultClient.mutate({
    mutation: DetachPaymentMethodMutation,
    variables: { input: { stripePaymentMethodId: paymentMethodId } },
  });

  if (this.selectedCreditCard === paymentMethodId) {
    this.selectedCreditCard = null;
  }

  this.isSavingCard = false;
}
```

The JavaScript test is written using jest and the testing library, and here is a challenge for you, before going any further in the text, can you spot what is missing from the test case?

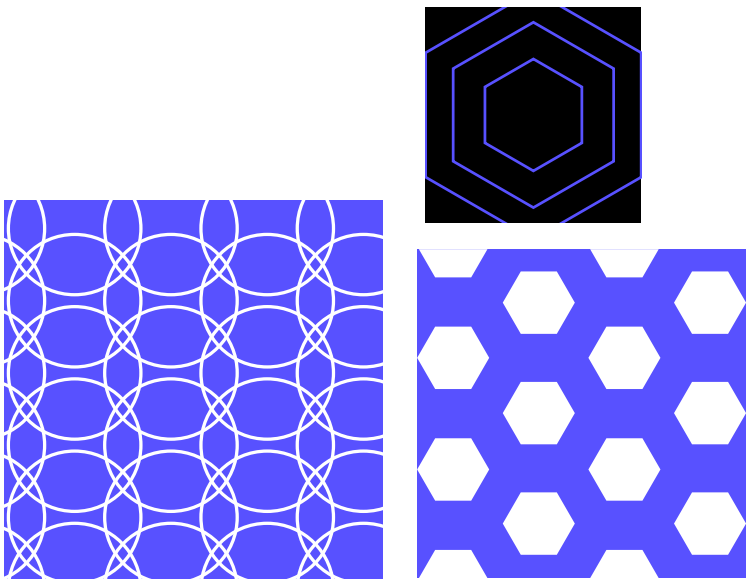
```
test('it handles error when removing credit card ', async () => {
  const data = await Payment.asyncData(asyncDataContext);
  data.paymentMethod = PaymentMethod.CREDIT_CARD;

  const { getAllByText } = render(Payment, {
    mocks,
    data() {
      return { ...data };
    },
  });

  const [removeButton] = getAllByText(Remove');
  await fireEvent.click(removeButton);
});
```

Let's start with the missing assertion at the end of the test case. If you haven't noticed, the last step that the test does is to wait for the click event. For some feature that removes a payment method from a user, asserting that a message is shown would be a good idea.

Besides, the test case relies on a specific setup that assumes that the Graphql mutation will always work. However, it is entirely possible that the Graphql mutation could fail and throw an exception within the production code. But the test is designed to cater to that situation. In this scenario, the test case relies on the jest testing framework to report the error, if any, rather than the error being handled within the test itself.

Points of attention

1. Avoid relying on errors from the test runner.
2. Make explicit the desired assertion in the test case.



Level III

Level III

In this chapter, we will go over the practices that one might face when practicing TDD for a while and, in some cases, even trying to apply TDD in code bases that are not ready for testability.

- Avoid having a test case that does everything simultaneously, leading to many lines in a single test case.
- Spending too much time setting up the test case points to a code that is not designed for testability, this relates to *The Mockery* covered later on.
- If possible, avoid violating encapsulation to achieve 100% of code coverage.

The Giant

A unit test that, although it is validly testing the object under test, can span thousands of lines and contain many, many test cases. This can be an indicator that the system under tests is a God Object, Carr (2022).

The Giant is covered in Episode 1 of the video[68] series covering the TDD anti-patterns hosted by Codurance.

The Giant anti-pattern is also a sign that something is lacking in the design of the codebase. Code design code is often a topic of discussion among TDD practitioners, Mancuso (2018). Similar to *The Excessive Setup*, this anti-pattern can also happen while developing in a TDD fashion. *The Giant* is often related to the God class code design. This is an "anti-pattern for Object-Oriented Programming" and goes against SOLID (Single responsibility, Open-Close, Liskov substitution, Interface segregation and Dependency Inversion), Martin (2017); Stemmler (2022) principles as well.

The Nuxtjs Project

In TDD, *The Giant* anti-pattern often shows itself with many assertions in a single test case. Dave Farley demonstrates this in his video. The same test file used from Nuxtjs[69] [70] *The Excessive Setup* shows signs of *The Giant*. Inspecting the code closer, we see a piece of code followed by assertions, some more code, then followed by further assertions all within the same test case:

---

68 https://www.codurance.com/publications/tdd-anti-patterns-chapter-1

69 https://nuxtjs.org

70 https://github.com/nuxt/nuxt.js/blob/d4b9e4b0553bcd617ecbc0b8b76871070b347fcb/packages/server/test/server.test.js#L166

```
test('should setup middleware', async () => {
  const nuxt = createNuxt()
  const server = new Server(nuxt)
  server.useMiddleware = jest.fn()
  server.serverContext = { id: 'test-server-context' }

  await server.setupMiddleware()

  expect(server.nuxt.callHook).toBeCalledTimes(2)
  expect(server.nuxt.callHook).nthCalledWith(1, 'render:setupMiddleware', server.app)
  expect(server.nuxt.callHook).nthCalledWith(2, 'render:errorMiddleware', server.app)

  expect(server.useMiddleware).toBeCalledTimes(4)
  expect(serveStatic).toBeCalledTimes(2)
  expect(serveStatic).nthCalledWith(1, 'resolve(/var/nuxt/src, var/nuxt/static)', server.options.render.static)
  expect(server.useMiddleware).nthCalledWith(1, {
    dir: 'resolve(/var/nuxt/src, var/nuxt/static)',
    id: 'test-serve-static',
    prefix: 'test-render-static-prefix'
  })
  expect(serveStatic).nthCalledWith(2, 'resolve(/var/nuxt/build, dist, client)', server.options.render.dist)
  expect(server.useMiddleware).nthCalledWith(2, {
    handler: {
      dir: 'resolve(/var/nuxt/build, dist, client)',
      id: 'test-serve-static'
    },
    path: '__nuxt_test'
  })

  const nuxtMiddlewareOpts = {
    options: server.options,
    nuxt: server.nuxt,
    renderRoute: expect.any(Function),
    resources: server.resources
  }
  expect(nuxtMiddleware).toBeCalledTimes(1)
  expect(nuxtMiddleware).toBeCalledWith(nuxtMiddlewareOpts)
  expect(server.useMiddleware).nthCalledWith(3, {
```

```
39      id: 'test-nuxt-middleware',
40      ...nuxtMiddlewareOpts
41    })
42
43    const errorMiddlewareOpts = {
44      resources: server.resources,
45      options: server.options
46    }
47    expect(errorMiddleware).toBeCalledTimes(1)
48    expect(errorMiddleware).toBeCalledWith(errorMiddlewareOpts)
49    expect(server.useMiddleware).nthCalledWith(4, {
50      id: 'test-error-middleware',
51      ...errorMiddlewareOpts
52    })
53  })
```

The point of attention here is to reflect on whether it makes sense to break out each block of code and assertion to its own individual test case. The issue here is having a test case that is structured so that it contains a block of test code assertions followed by further test code and then more subsequent assertions.

It would require further inspection to double-check if it is possible to break the code as suggested above. However, this scenario is a good example of how The Giant anti-pattern can manifest itself. As Dave Farley says in his video, this practice is not recommended.

Brian Okken in *Python Testing with pytest* (Okken 2022) also outlines that having a test setup with an *Arrange-Assert-Act-Assert-Act-Assert* structure is an anti-pattern. He argues that such a style works until the test fails. This is because when it does indeed fail, any of the previous actions could have caused the failure, making it difficult detect the cause of the failure quickly.

Points of Attention

1. Test after, instead of test first.

The Excessive Setup

A test that requires a lot of work setting up in order to even begin testing. Sometimes several hundred lines of code are used to set up the environment for one test, with several objects involved, making it difficult to really ascertain what is tested due to the “noise” of all the setups going on, Carr (2022).

The Excessive Setup is covered in Episode 1 of the video[71] series covering the TDD anti-patterns hosted by Codurance,

Practitioners can relate to *The Excessive Setup* anti-pattern to the non-practice of TDD from the start and the lack of practicing object calisthenics.[72]

The classic approach for *The Excessive Setup* anti-pattern is when you want to test a specific behaviour within your code. Still, it becomes difficult due to the many dependencies that you have to set up first. When the amount of these dependencies start to hurt testability, it is a code smell.

The Nuxtjs Project

The following code demonstrates a test case from the Nuxtjs framework which shows this anti-pattern. The test file for the server starts with a few mocks, and then it continues, until the method beforeEach which contains more setup work (setting up the mocks).

```
1  jest.mock('compression')
2  jest.mock('connect')
3  jest.mock('serve-static')
4  jest.mock('serve-placeholder')
5  jest.mock('launch-editor-middleware')
6  jest.mock('@nuxt/utils')
7  jest.mock('@nuxt/vue-renderer')
8  jest.mock('../src/listener')
9  jest.mock('../src/context')
10 jest.mock('../src/jsdom')
11 jest.mock('../src/middleware/nuxt')
12 jest.mock('../src/middleware/error')
13 jest.mock('../src/middleware/timing')
14
15 describe('server: server', () => {
16   const createNuxt = () => ({
17     options: {
18       dir: {
19         static: 'var/nuxt/static'
20       },
21       srcDir: '/var/nuxt/src',
22       buildDir: '/var/nuxt/build',
```

71 https://www.codurance.com/publications/tdd-anti-patterns-chapter-1

72 Object Calisthenics was introduced in the book The ThoughtWorks Anthology: Essays on Software Technology and Innovation (Steinberg 2008), in there nine steps to better software design were introduced.

```
23        globalName: 'test-global-name',
24        globals: { id: 'test-globals' },
25        build: {
26          publicPath: '__nuxt_test'
27        },
28        router: {
29          base: '/foo/'
30        },
31        render: {
32          id: 'test-render',
33          dist: {
34            id: 'test-render-dist'
35          },
36          static: {
37            id: 'test-render-static',
38            prefix: 'test-render-static-prefix'
39          }
40        },
41        server: {},
42        serverMiddleware: []
43      },
44      hook: jest.fn(),
45      ready: jest.fn(),
46      callHook: jest.fn(),
47      resolver: {
48        requireModule: jest.fn(),
49        resolvePath: jest.fn().mockImplementation(p => p)
50      }
51    })
52
53    beforeAll(() => {
54      jest.spyOn(path, 'join').mockImplementation((...args) => `join(${args.
        join(', ')})`)
55      jest.spyOn(path, 'resolve').mockImplementation((...args) => `resolve(${args.
        join(', ')})`)
56      connect.mockReturnValue({ use: jest.fn() })
57      serveStatic.mockImplementation(dir => ({ id: 'test-serve-static', dir }))
58      nuxtMiddleware.mockImplementation(options => ({
59        id: 'test-nuxt-middleware',
60        ...options
61      }))
62      errorMiddleware.mockImplementation(options => ({
```

```
63        id: 'test-error-middleware',
64        ...options
65      }))
66      createTimingMiddleware.mockImplementation(options => ({
67        id: 'test-timing-middleware',
68        ...options
69      }))
70      launchMiddleware.mockImplementation(options => ({
71        id: 'test-open-in-editor-middleware',
72        ...options
73      }))
74      servePlaceholder.mockImplementation(options => ({
75        key: 'test-serve-placeholder',
76        ...options
77      }))
78    })
79  })
```

Reading the test from the beginning gives an idea that to start with, there are 13 *jest.mock* invocations. Besides that, there is more setup on the *beforeEach*, around 9 spies and stub setups. Probably, if we wanted to create a new test case from scratch or move tests across different files, we would need to keep the same excessive setup as it is.

The Testable Project

The Excessive Setup anti-pattern is a common trap. The following code is from a research project called Testable, Marabesi and Silveira (2020) and depicts a single function that also suffers from many dependencies, leading to an excessive setup to get the function to execute:

wThe number of parameters to test the function are many such that if anyone were to begin writing a new test case, they will likely forget what to pass in order to receive the desired result.

Farley (2021) shows another example from Jenkins, an open-source CI/CD project. He depicts a particular test case that builds a web browser to assert the URL being used. In this example, it could have achieved the same outcome in a much simpler way but using a regular object calling a method to assert.

Points of Attention

1. Revisit the SOLID for the test case and arrange it accordingly (do not fear refactor tests)

2. Are you adding a test case that requires changing too many items in the setup? Rearrange the tests.

The Inspector

A unit test that violates encapsulation in an effort to achieve 100% code coverage but knows so much about what is going on in the object that any attempt to refactor will break the existing test and require any change to be reflected in the unit test Carr (2022).

The Inspector is covered in Episode 2 of the video[73] series covering the TDD anti-patterns hosted by Codurance.

Often, the purpose of testing is combined with "inspection", which can be a result of couples the test case to implementation details such as reflection or exposing internal behaviour to inspect the output.

Ideally, the way to think about a given scenario is to test for behaviour instead.

The Git Release Bot Project – Exposing Details

Sometimes while testing, it gets a bit trickier as often we find ourselves in situations where we would like to verify that if given an input X we will receive the outcome Y. In the following code snippet, there is the method *getFilesToWriteRelease* (line 17) and *setFilesToWriteRelease* (line 22), if we think about the context of the class, why do such methods exist?

```
1  class Assembly
2  {
3      /* skipped code */
4
5      public function __construct(
6          FindVersion $findVersion,
7          FileRepository $fileRepository,
8          string $branchName,
9          FilesToReleaseRepository $filesToReleaseRepository
10     ) {
11         $this->findVersion = $findVersion;
12         $this->fileRepository = $fileRepository;
13         $this->branchName = $branchName;
14         $this->filesToReleaseRepository = $filesToReleaseRepository;
15     }
16
```

73 https://www.codurance.com/publications/tdd-anti-patterns-chapter-2

```
17      public function getFilesToWriteRelease(): array
18      {
19          return $this->filesToWriteRelease;
20      }
21
22      public function setFilesToWriteRelease(array $filesToWriteRelease)
23      {
24          $this->filesToWriteRelease = $filesToWriteRelease;
25          return $this;
26      }
27
28      public function packVersion(): Release
29      {
30          $filesToRelease = $this->getFilesToWriteRelease();
31
32          if (count($filesToRelease) === 0) {
33              throw new NoFilesToRelease();
34          }
35
36          $files = [];
37
38          /** @var File $file */
39          foreach ($filesToRelease as $file) {
40              $files[] = $this->fileRepository->findFile(
41                  $this->findVersion->getProjectId(),
42                  sprintf('%s%s', $file->getPath(), $file->getName()),
43                  $this->branchName
44              );
45          }
46
47          $versionToRelease = $this->findVersion->versionToRelease();
48
49          $release = new Release();
50          $release->setProjectId($this->findVersion->getProjectId());
51          $release->setBranch($this->branchName);
52          $release->setVersion($versionToRelease);
53
54          $fileUpdater = new FilesUpdater($files, $release, $this-
            >filesToReleaseRepository);
55          $filesToRelease = $fileUpdater->makeRelease();
56
57          $release->setFiles($filesToRelease);
```

```
58
59          return $release;
60      }
61  }
```

In this sense, if we think about *Object-Orientated Programming,* McLaughlin, Pollice, and West (2007), we talk about invoking methods in objects. Such invocations allow us to interact and receive the result. But, the get/set methods break the encapsulation and exist solely for the purpose of testing.

Inspecting Code with Reflection

Another possible reason for experiencing *The inspector* is to add complexity to achieve inspection for a specific part of the code with reflection. Before proceeding, let's recap what actual reflection is and why it is used in the first place.

Baeldung[74] gives a few examples of why you might need reflection, the main usage is to get an understanding of a given part of the code, which methods, which properties and act on those. The following example is used on his site:

```
1   public class Employee {
2       private Integer id;
3       private String name;
4   }
5
6   @Test
7   public void whenNonPublicField_thenReflectionTestUtilsSetField() {
8       Employee employee = new Employee();
9       ReflectionTestUtils.setField(employee, “id”, 1);
10
11      assertTrue(employee.getId().equals(1));
12  }
```

Now that we understand the reflection's superpowers, why might it hurt testability? Luckily, we had this question asked at StackOverflow a few years ago.[75]

The thread is long and shares different opinions developers posted there. But in short, the consensus is that it is a bad practice to go to that level of detail when testing code.

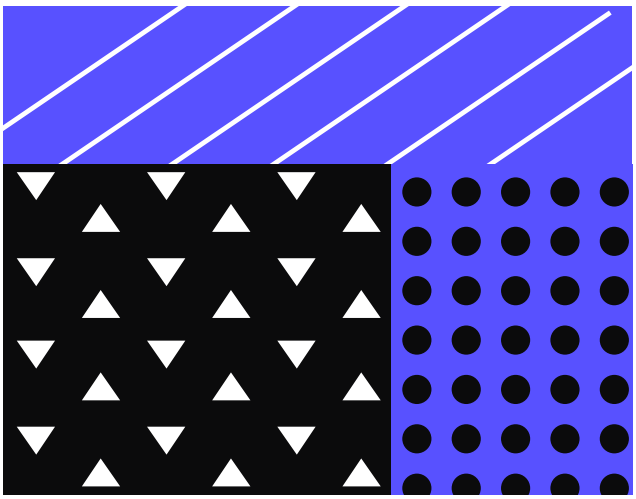
---

74 Guide to ReflectionTestUtils for Unit Testing.

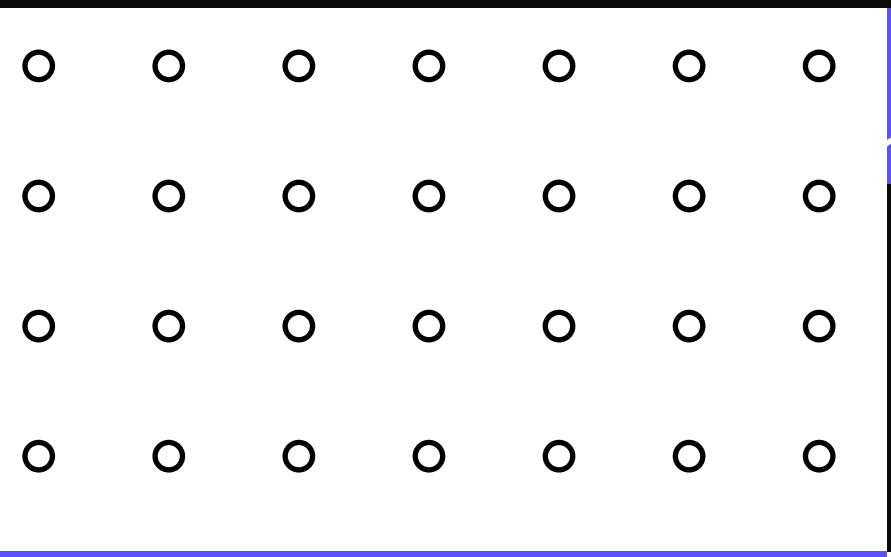
75 Is it bad practice to use Reflection in Unit testing?

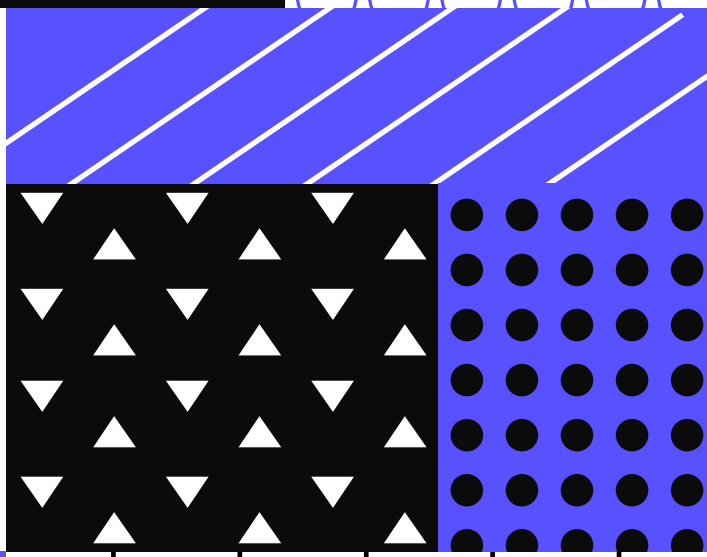
Points of Attention

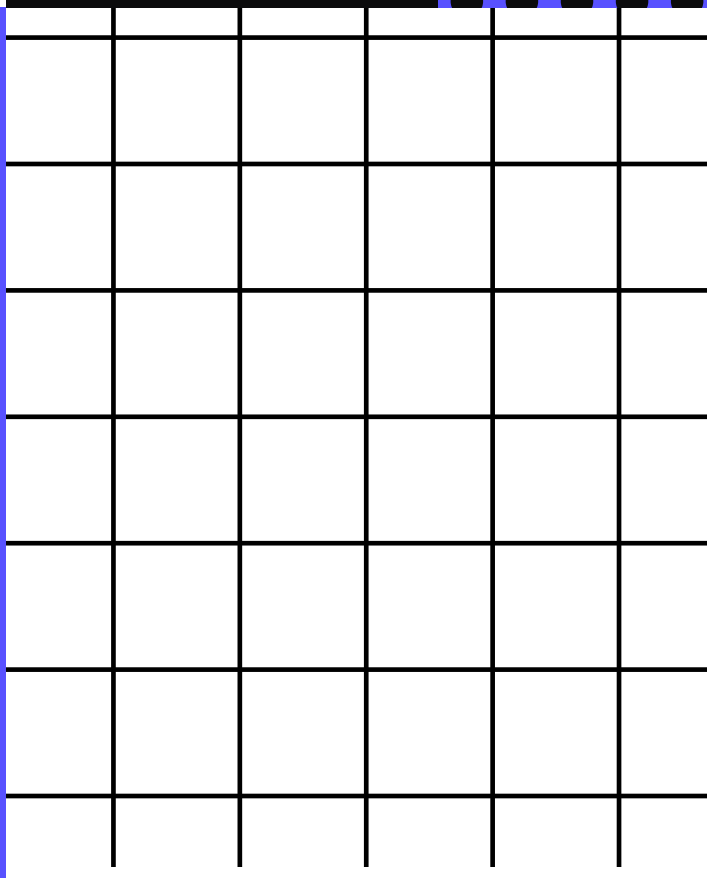
1. Avoid exposing methods purely for the purpose of inspecting output or defining inputs within a test case.
2. Modifying production code for the sake of testing is a *smell* of something wrong
   in the design.



Level VI







Level IV

If you made it this far, this level shows one of the most difficult scenarios that the test-first approach brings. At this level, we will go through *The Mockery*, the most popular anti-pattern that came out of the survey.

- Watch out for testing the test double instead of the production code.
- A single test case can have multiple anti-patterns at once.
- Avoid not cleaning up the created data for a specific test case, it is commended to avoid sharing data across tests. Also relates to *The Peeping Tom*.
- Avoid having a test suite that takes a long time to run whenever possible.

The Mockery

Sometimes mocking can be good and handy. But, at other times, practitioners can lose themselves trying to mock out what isn't being tested. In this case, a unit test contains so many mocks, stubs, and/or fakes that the system under test isn't even being tested at all, instead, data returned from mocks is what is being tested Carr (2022).

The Mockery is covered in Episode 2 of the video[76] series covering the TDD anti-patterns hosted by Codurance.

The Mockery is one of the most popular anti-patterns experienced by practitioners. It seems that everyone has had some experience mocking code to allow them to test some behaviour. The first idea of mocking is simple: avoid external code to focus on what you want to test. However, mocking can be tricky to implement correctly.

Mocking can also take many different approaches. For example, Fowler (2022b) and Bob (2022) classified the various types of mocking as follows: dummies, spies, fakes and stubs. Although Uncle Bob refers to "True mocks," whereas Martin Fowler refers to "Test double".

Uncle Bob explains that "mocks" got spread as it is easier to say: "I will mock that," or "can you mock this"?

The difference between those is important because we, as practitioners, have the habit of calling everything a mock, thus causing confusion around the true intention. For example, Spring.io goes even further, avoiding the debate about if we should use mocks or not (referring to the Classical and the London School of

76 https://www.codurance.com/publications/tdd-anti-patterns-chapter-2

TDD)[77] – I will do the same, this is a discussion that deserves a blog post itself.

Martin Fowler also wrote about this very same subject, Fowler (2022) and agrees that for a while, he also thought about mocks and stubs being the same.[78]

- DUMMIES – You pass in something, often resulting in the object not being utilised at all
- STUB – Unlike dummies, stubs are objects created so that you do care how they are used. For example, to tricky an authorization to test if the user can/can't do certain actions in the system.
- SPIES – To assert that a method was called by the system under test, as the post by Martin (2014): "You can use Spies to see inside the workings of the algorithms you are testing".
- TRUE MOCKS – Is interested in the behaviour instead of the return of functions. It cares about which functions were invoked, with what arguments, and how often.
- FAKES – Fakes have business logic, so they can drive the system under test with different sets of data.

The Mockery anti-pattern refers to the same definition stated by Uncle Bob, referring to all mocks. To make things clear, let's split the anti-pattern into two parts: the first is the excessive number of mocks needed to test a class. For example – it's also related to *The Excessive Setup*.

The second part is testing the mock instead of its interaction with the code under testing. For example, if we have the following code sample showing an interaction with a payment gateway.

```
/**
 * Two constructor dependencies, both need to be
 * mocked in order to test the process method.
 */
class PaymentService(
    private val userRepository: UserRepository,
    private val paymentGateway: PaymentGateway
) {

    fun process(
      user: User,
      paymentDetails: PaymentDetails
    ): Boolean {
```

77 Are You Chicago Or London When It Comes To TDD? – https://www.youtube.com/watch?v=_S5iUf0ANyQ

78 In practice, practitioners still treat different test doubles as being the same. The communication falls back to "mock this" or "mock that".

```
        if (userRepository.exists(user)) {
            return paymentGateway.pay(paymentDetails)
        }

        return false
    }
}
```

With the given test:

```
class TestPaymentService {
    private val userRepository: UserRepository = mockk()
    private val paymentGateway: PaymentGateway = mockk()
    private val paymentService = PaymentService(
      userRepository,
      paymentGateway
    )

    @Test
    fun paymentServiceProcessPaymentForUser() {
        val user: User = User()
        every { userRepository.exists(any()) } returns true
        every { paymentGateway.pay(any()) } returns true          // setting up
        the return for the mock

        assertTrue(paymentService.process(user, PaymentDetails())) // asserting
        the mock
    }
}
```

Points of Attention

1. Historic reasons/TDD styles?
2. As simple as the code is, the easiest is to overuse it
3. Also, no experience in TDD

The One

A combination of several patterns, particularly *The Free Ride* and *The Giant*, a unit test that contains only one test method which tests the entire set of functionality an object has. A common indicator is that the test method is often the same as the unit test name, and contains multiple lines of setup and assertions Carr (2022).

The One is covered in Episode 6 of the video[79] series covering the TDD anti-patterns hosted by Codurance.

Despite the name, *The One* is the anti-pattern that combines different anti-patterns. By definition, it is related to *The Giant* and *The Free Ride*.

As we have already seen, *The Giant* appears when a test case tries to do everything at once within a single test case.

There are some variants of that. The following snippet is extracted from the book *xUnit Test Patterns*, Meszaros (2007) and depicts the Giant anti-pattern (even though the number of lines is not as many as in the first episode). Despite being shorter in length though, the following is still an example of *The Giant* anti-pattern because the test case is trying to exercise all of the methods within the Flight object within a single test:

```
1  public void testFlightMileage_asKm2() throws Exception {
2      // set up fixture
3      // exercise constructor
4      Flight newFlight = new Flight(validFlightNumber);
5      // verify constructed object
6      assertEquals(validFlightNumber, newFlight.number);
7      assertEquals("", newFlight.airlineCode);
8      assertNull(newFlight.airline);
9      // set up mileage
10     newFlight.setMileage(1122);
11     // exercise mileage translator
12     int actualKilometres = newFlight.getMileageAsKm();
13     // verify results
14     int expectedKilometres = 1810;
15     assertEquals( expectedKilometres, actualKilometres);
16     // now try it with a canceled flight
17     newFlight.cancel();
18
19     try {
20         newFlight.getMileageAsKm();
21         fail("Expected exception");
22     } catch (InvalidRequestException e) {
23         assertEquals( "Cannot get cancelled flight mileage",
24         e.getMessage());
25     }
```

79 https://www.codurance.com/publications/tdd-and-anti-patterns-chapter-6.

The comments even give us a hint on how to split the single test case into multiple tests. Likewise, *The Free Ride* also can be noted in this example, as for each setup, some assertions follow.

The Jenkins Project

The next code example may be simpler to spot when *The Free Ride* anti-pattern appears. As previously depicted, the example that follows was extracted from the Jenkins repository. In this case, the distinction between *The Free Ride* and *The Giant* is somewhat blurred, but still, it is easy to prevent a single test case from doing too much.

```
1  public class ToolLocationTest {
2      @Rule
3      public JenkinsRule j = new JenkinsRule();
4
5      @Test
6      public void toolCompatibility() {
7          Maven.MavenInstallation[] maven = j.jenkins.getDescriptorByType(Maven.
           DescriptorImpl.class).getInstallations();
8          assertEquals(1, maven.length);
9          assertEquals("bar", maven[0].getHome());
10         assertEquals("Maven 1", maven[0].getName());
11
12         Ant.AntInstallation[] ant = j.jenkins.getDescriptorByType(Ant.
           DescriptorImpl.class).getInstallations();
13         assertEquals(1, ant.length);
14         assertEquals("foo", ant[0].getHome());
15         assertEquals("Ant 1", ant[0].getName());
16
17         JDK[] jdk = j.jenkins.getDescriptorByType(JDK.DescriptorImpl.class).
           getInstallations();
18         assertEquals(Arrays.asList(jdk), j.jenkins.getJDKs());
19         assertEquals(2, jdk.length); // JenkinsRule adds a 'default' JDK
20         assertEquals("default", jdk[1].getName()); // make sure it's really that
           we're seeing
21         assertEquals("FOOBAR", jdk[0].getHome());
22         assertEquals("FOOBAR", jdk[0].getJavaHome());
23         assertEquals("1.6", jdk[0].getName());
24     }
25 }
```

The Generous Leftovers

An instance where one unit test generates data that is stored somewhere, and another test then employs that data for its own purposes. If the "generator" is not executed, or run after the second test, the test using that data will fail outright, Carr (2022).

The Generous Leftovers is covered in Episode 2 of the video[80] series covering the TDD anti-patterns hosted by Codurance.

While practicing TDD, tasks such as setting up the state in which the test will run are part of the basic fundamentals, whether that be setting up fake data, listeners, authentication or any other dependent state.

We define them because they are crucial for the test, but sometimes we forget to reset the state to be what it was before the test was run. In short, the phases that a test case should have, as Fowler (2022) and Meszaros (2007), are: "setup, exercise, verify, teardown." The resetting of the state should occur within the teardown stage to avoid affecting other test cases.

Not doing this state reset can cause different issues. The first one is causing the next test to fail, where it would have otherwise passed without problems. The following list tries to depict a few scenarios in which this may occur.

1. Setting up listeners and forgetting to remove them might also cause memory leaks.
2. Populating data without removing them – things like files, databases, or even cache.
3. Last but not least, depending on one test creating the data needed and using it in another test.
4. Cleaning up test doubles.[81]

If we think about the third scenario in the above list, such behaviour could be something that gets tricky when testing. For example, using persistent data is a must for end-to-end testing. On the other hand, the last scenario in the list is a common source of errors while developing guided by tests. Often, as the mock is usually used to collect calls in the object (and verify it later), it is common to forget to restore its state.

Codingwithhugo[82] demonstrates this behaviour in a code snippet using Jest, giving the following test case (and assume they are in the same scope):

---

80 https://www.codurance.com/publications/tdd-anti-patterns-chapter-2

81 Also relates to keeping state between tests as discussed previously.

82 Jest set, clear and reset mock/spy/stub implementation.

```
const mockFn = jest.fn(); // setting up the mock

function fnUnderTest(args1) {
  mockFn(args1);
}

test('Testing once', () => {
  fnUnderTest('first-call');
  expect(mockFn).toHaveBeenCalledWith('first-call');
  expect(mockFn).toHaveBeenCalledTimes(1);
});

test('Testing twice', () => {
  fnUnderTest('second-call');
  expect(mockFn).toHaveBeenCalledWith('second-call');
  expect(mockFn).toHaveBeenCalledTimes(1);
});
```

The first test that calls the function under test will pass, but the second will fail. The reason behind this is the lack of a resetting of the mock. The test fails to say that the *mockFn* was called twice. Correcting this is as easy as:

```
test('Testing twice', () => {
  mockFn.mockClear();              // clears the previous execution

  fnUnderTest('second-call');
  expect(mockFn).toHaveBeenCalledWith('second-call');
  expect(mockFn).toHaveBeenCalledTimes(1);
});
```

Such behaviour is not as often faced by JavaScript practitioners, as the JavaScript functional scope prevents that out of the box within the language constructs. However, being mindful of these details can still make a difference while writing tests.[83]

Points of Attention

1. Lack of practice on TDD.
2. Persistent fixtures can become a source of errors.
3. Your tests are coupled on a specific sequence to be executed.
4. Mixing different types of tests, integration/unit/end-to-end.

---

83 More recently Marvin Hagemeister wrote a blog post entitled Running 1000 tests in 1s and much of the performance achieved was due to disabling some default behaviour of cleaning up resources that Jest has. When such a level of optimization is needed, having control over the state becomes crucial.

The Slow Poke

A unit test that runs incredibly slow. When practitioners kick it off, they have time to go to the bathroom, grab a smoke, or worse, kick the test off before they go home at the end of the day, Carr (2022).

The Slow Poke reminds me of the Japanese video game Pokémon. The game contains a creature with this name who is able to make its opponents less efficient when attacking. It is this characteristic that it shares with the anti-pattern of the same name. Here though, the Slow Poke anti-pattern reduces the efficiency of a test suite run. Usually, this anti-pattern causes an automated test suite to take a longer time to run and as a result, lengthen the feedback cycle.

Time-related code is usually difficult to handle under a test case and is often a key reason why *The Slow Poke* anti-pattern surfaces. Time-related code requires us to manipulate time in different ways to allow us to simulate time-based scenarios. For example, we might wish to test that an automated payment run is triggered at the end of a month within a payment system.

In order to do that, we would need a way to handle time and check for a specific date (the last day of the month, in this case). Alongside the date, we would likely also need to simulate the time (perhaps somewhere around the morning or the evening). In other words, we need a way to handle the time and deal with it without the need to wait until the end of the month to run the test.

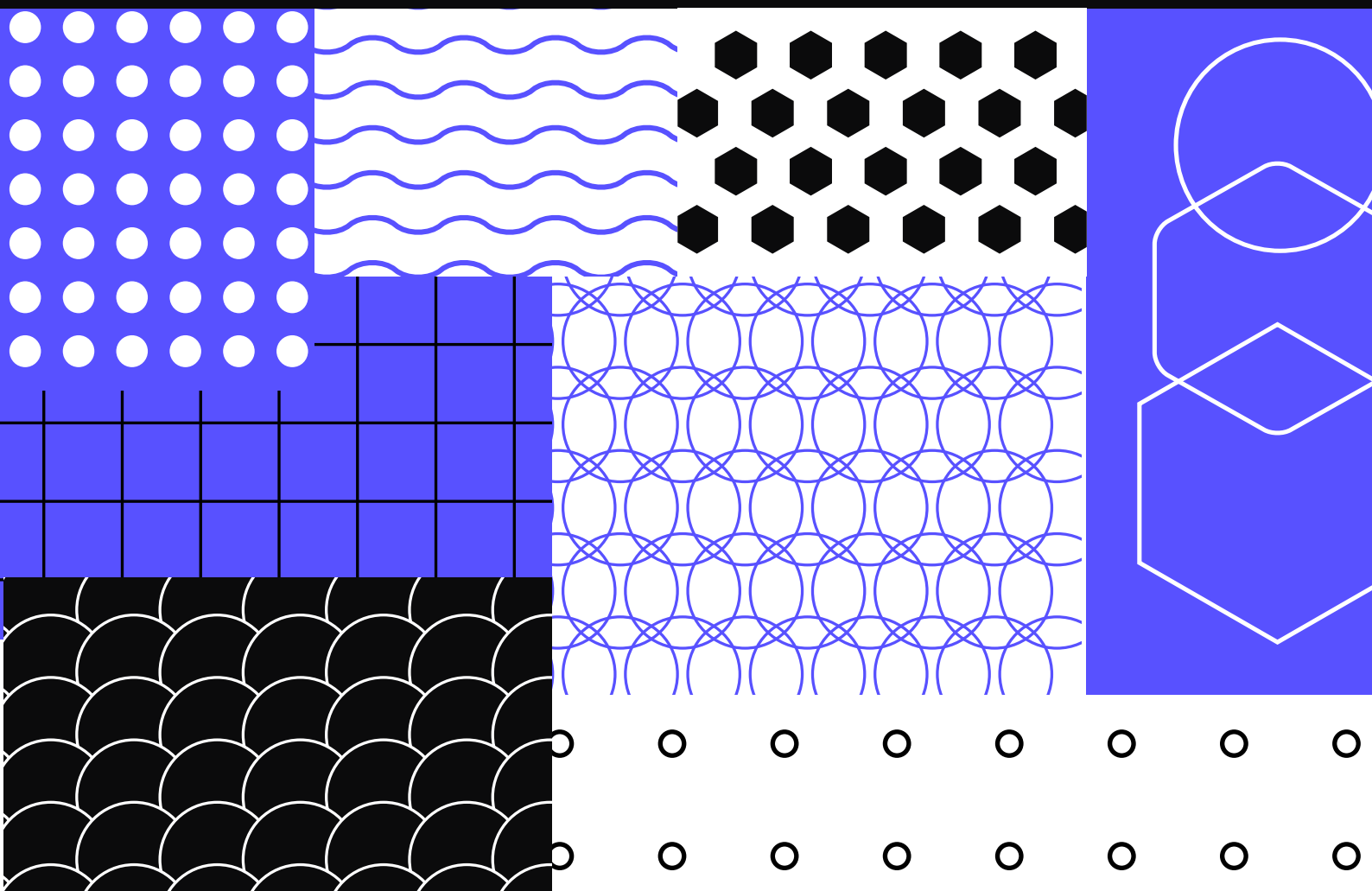
Time is asynchronous and if depended upon directly, leads to non-determinism, as mentioned by Fowler (2011). Fortunately, there is a way to overcome this situation but utilizing mocking within our tests.

Pushing towards more integration tests or end-to-end tests can also transform the test suite into a pain to run, taking long hours or even days to complete. This approach is also related to the ice cream cone test strategy. In an ideal scenario, you would follow the pyramid testing strategy where you would have as the base: more unit tests, a few integration tests, and even fewer end-to-end tests, Vocke (2018).

Points of Attention

1. Monolithic code bases where compile time takes more time than executing the tests
2. Testing with strategies that the framework already gives out of the box
3. Having too many integrations or e2e tests instead of unit (ice cream code as seen previously)

Conclusion – Patterns That Make TDD Harder



Conclusion – Patterns That Make TDD Harder

Having reached the end of this book, you should now have a better awareness and understanding of some of the patterns that can make test driving code difficult.

We started this journey by understanding the origins of TDD anti-patterns. Previously we discussed some of the history behind the creation of this book, beginning with the challenges faced by practitioners, on a daily basis, to test applications. We also covered the Test Pyramid which is commonly followed in the software development industry today. We saw that different authors gave awareness to the anti-patterns listed in different forms.

We also explored what is said about anti-patterns nowadays.

The survey shared with practitioners to understand what they know about TDD anti-patterns and what were the possible impact that those anti-patterns caused in the day-to-day coding. The data gathered from practitioners were covered by four different categories:

- Professional background – get to know better who is answering the survey.
- TDD practices on the daily basis – the practices that are practiced daily.
- TDD practices of companies I worked at – the practices that companies have daily regarding TDD.
- Anti-patterns – a section dedicated to getting information if practitioners can recall the anti-patterns.
- Finishing up – Share an email to share results from the data gathered.

The survey data triggered discussions about how TDD is applied at companies.

With the introduction and the data collected in place, we moved on to dive deeper into the anti-patterns list.

The method used in this book to present the list of anti-patterns was different to the one that is usually found on the internet. Here we adopted the idea of levels that represent the journey a practitioner will go on whilst learning TDD and the anti-patterns that might pop up during that journey.

In total, we divided the anti-patterns into four levels:

- Level I – This chapter covered most of the anti-patterns as they spot early on when practitioners are learning how to test-driving code.
- Level II – This chapter was marked as a beginner moving towards an intermediate level which led to more challenges during the TDD practice. Here we saw that anti-patterns can also be related to the lack of using principles of software development such as Single Responsibility in test cases.
- Level III – This chapter is an extension of Level II as it goes deeper into software design and relates object calisthenics with the anti-patterns presented.
- Level IV – Last but not least, here, anti-patterns such as *The Mockery* were covered, which introduces a new way of thinking about how to test drive code using test doubles and things that make the test suite slow, also known as *The Slow Poke*.

The levels were created to give this book a structure and a way to format the anti-patterns in such a way that could be matched with different subjects while practicing TDD, which does not mean that only beginners will face scenarios depicted in the level II or that only experienced developers will face scenarios in level IV.

... with this content, hopefully, we can reach an agreement that what we shared here are patterns that make testing harder.

Last but not least, with this content, hopefully, we can reach an agreement that what we shared here are patterns that make testing harder.

Even though they are named “anti-patterns,” the attempt here is to rather make them “patterns” that make test driving code more difficult and remove the negative tone from them. As most code bases can have at least one or more of them, and practitioners will deal with at least one of them in their journey.

What the Experience Has to Say

I don’t recall who introduced me to it, but I remember reading the iconic book by Kent Beck, TDD by Example on my commute time. Every page that I turned was a discovery. Those page turns were also tinged with frustration. The baby steps approach was in conflict with the confidence I had that code would work. “It will pass the test, it is hard coded”. One of the listed misconceptions by Olena Borzenko in her talk in 2021 at the TDD conference related to that type of thinking.

On the other hand, thinking about when I started out, and the context where I was, did not help me to improve my technical skills. Specifically, TDD, which also connects to the results we saw in the survey (more than 50% of companies do not practice TDD).

One of the biggest challenges for practitioners is to keep going with the test-first approach, regardless of the environment, regardless of the team we are working

One of the biggest challenges for practitioners is to keep going with the test-first approach, regardless of the environment, regardless of the team we are working with. We are still learning and the adoption of practices that are taken as good defaults for projects are not common practice for the majority of us.

with. We are still learning and the adoption of practices that are taken as good defaults for projects are not common practice for the majority of us. I recall joining projects that didn't have a test-first culture. The approach was to build it from the ground up, and in that, I am not alone, Pérez (2022).

Where To Go From Here

Despite the journey covered in this book, you might also want to go a bit further in the practice of TDD and what other authors have to say, for that, I compiled the following list of resources that could help you.

- YouTube has many talks around TDD, to keep track of what I found interesting I created a playlist on YouTube that might be worth checking. The focus was to build a progressive playlist, starting from the basics and incrementing the complexity step-by-step.
- Effective Software Testing by Mauricio Aniche – This is one of the books that is a must-read for practitioners, as he shares a detailed point of view on how to write tests and at the same time delivers a guide for developers to follow.
- We haven't touched on the London school x Chicago school of the TDD, it might be worth checking this out as well, as it might impact which patterns practitioners will face more based on the chosen approach.

Last but not least, there are many books about TDD covering different aspects of the practice and we already cited some of them throughout the book. But we haven't touched yet upon how to keep up to date with the practice beyond the literature.

As much as books cover a specific subject in great detail, I often find that engaging with the various software development communities around me is the best mechanism for becoming aware of new things and the number of insights it gives me.

There are a vast number of software development communities right around the world today. The vast majority can be found on meetup.com. For example, Codurance hosts events regularly; Tech Excellence also hosts events focused on technical and core skills.

Getting in touch with people in the community helps not only you as an individual to meet with people and discuss what they are doing, but it can also be an opportunity to share what you learned and what you want to talk about. Despite the small list of groups listed here, I am sure you will find a community nearby that talks about your favourite subject.

Appendix

Here you will find resources that are related to the book.

Google form – Survey

Aniche, Mauricio. 2022. “Mauricio Aniche: How Code Coverage Can Be Used and Abused to Guide Testing.”
https://www.youtube.com/watch?v=f1DueZcSxRse.com/watch?v=f1DueZcSxRs

Beck, Kent. 2003. *Test-Driven Development: By Example.* Addison-Wesley Professional.

Bob), Robert C. Martin (Uncle. 2022. “The Little Mocker.”
https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html

Borzenko, Olena. 2021. “TDD Misconceptions.”
https://www.youtube.com/watch?v=TbkBMeAt4K0

Bugayenko, Yegor. 2021. “SSD 14/16: Test Patterns and Anti-Patterns.”
https://www.youtube.com/watch?v=KiUb6eCGHEY

Carr, James. 2022. “TDD Anti-Patterns.”
https://web.archive.org/web/20100105084725/http://blog.james-carr.org/2006/11/03/tdd-anti-patterns

Cohn, Mike. 2009. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional.

Dijkstra, Edsger Wybe, and others. 1970. “Notes on Structured Programming.” Technological University, Department of Mathematics.

Farley, Dave. 2021. “When Test Driven Development Goes Wrong.”
https://www.youtube.com/watch?v=UWtEVKVPBQ0&feature

Fowler, Martin. 2011. “Eradicating Non-Determinism in Tests.”
https://martinfowler.com/articles/nonDeterminism.html

———. 2022a. “Mocks Aren’t Stubs.”
https://martinfowler.com/articles/mocksArentStubs.html

———. 2022b. “TestDouble.”
https://martinfowler.com/bliki/TestDouble.html

Freeman, Steve, and Nat Pryce. 2009. *Growing Object-Oriented Software, Guided by Tests*. Pearson Education.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education.
https://books.google.es/books?id=6oHuKQe3TjQC

Glitzel, Rodaney. 2022. “Singleton and Unit Testing.”
https://stackoverflow.com/questions/8256989/singleton-and-unit-testing/8263599#8263599

Hermans, Felienne. 2021. *The Programmer’s Brain: What Every Programmer Needs to Know About Cognition.* Simon; Schuster.

jestjs.io. 2021. “Testing Asynchronous Code.”
https://jestjs.io/docs/asynchronous

Mancuso, Sandro. 2018. “DevTernity 2018: Sandro Mancuso – Does TDD Really Lead to Good Design?”
<https://www.youtube.com/watch?v=KyFVA4Spcgg

Marabesi, Matheus. 2022. “Improving Testing Assertions.”
https://www.codurance.com/publications/improving-testing-assertionsnce.com

Marabesi, M, and I Frango Silveira. 2020. “EVALUATION of Testable, a Gamified Tool to Improve Unit Test Teaching.” In *INTED2020 Proceedings,* 330–38. IATED.

Martin, Robert C. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.

Martin, Robert C. 2014. “The Little Mocker.”
https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker

Martin, Robert Cecil. 2017. “Clean Architecture a Craftsman’s Guide to Software Structure and Design.”
https://archive.org/details/CleanArchitecture/page/n179/mode/2up

McLaughlin, Brett, Gary Pollice, and David West. 2007. *Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to Ooa&D.* “ O’Reilly Media, Inc.”.

Meszaros, Gerard. 2007. *XUnit Test Patterns: Refactoring Test Code*. Pearson Education.

Okken, Brian. 2022. *Python Testing with Pytest*. Pragmatic Bookshelf.

Percival, Harry, and Bob Gregory. 2020. *Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices.* “ O’Reilly Media, Inc.”.

Pérez, Julio César. 2022. “Una Historia de Testing.”
https://www.meetup.com/pt-BR/codurance-craft-events/events/287021364/.com

Radziwill, Nicole. 2020. “Accelerate: Building and Scaling High Performance Technology Organizations. (Book Review) 2018 Forsgren, N., J. Humble and G. Kim. Portland or: IT Revolution Publishing. 257 Pages.” Taylor & Francis.

Steinberg, Daniel H. 2008. *The Thoughtworks Anthology: Essays on Software Technology and Innovation*. Pragmatic Bookshelf.

Stemmler, Khalil. 2022. *SOLID – Introduction to Software Design and Architecture with Typescript. https://solidbook.io.*

Vocke, Ham. 2018. “The Practical Test Pyramid.”
https://martinfowler.com/articles/practical-test-pyramid.html

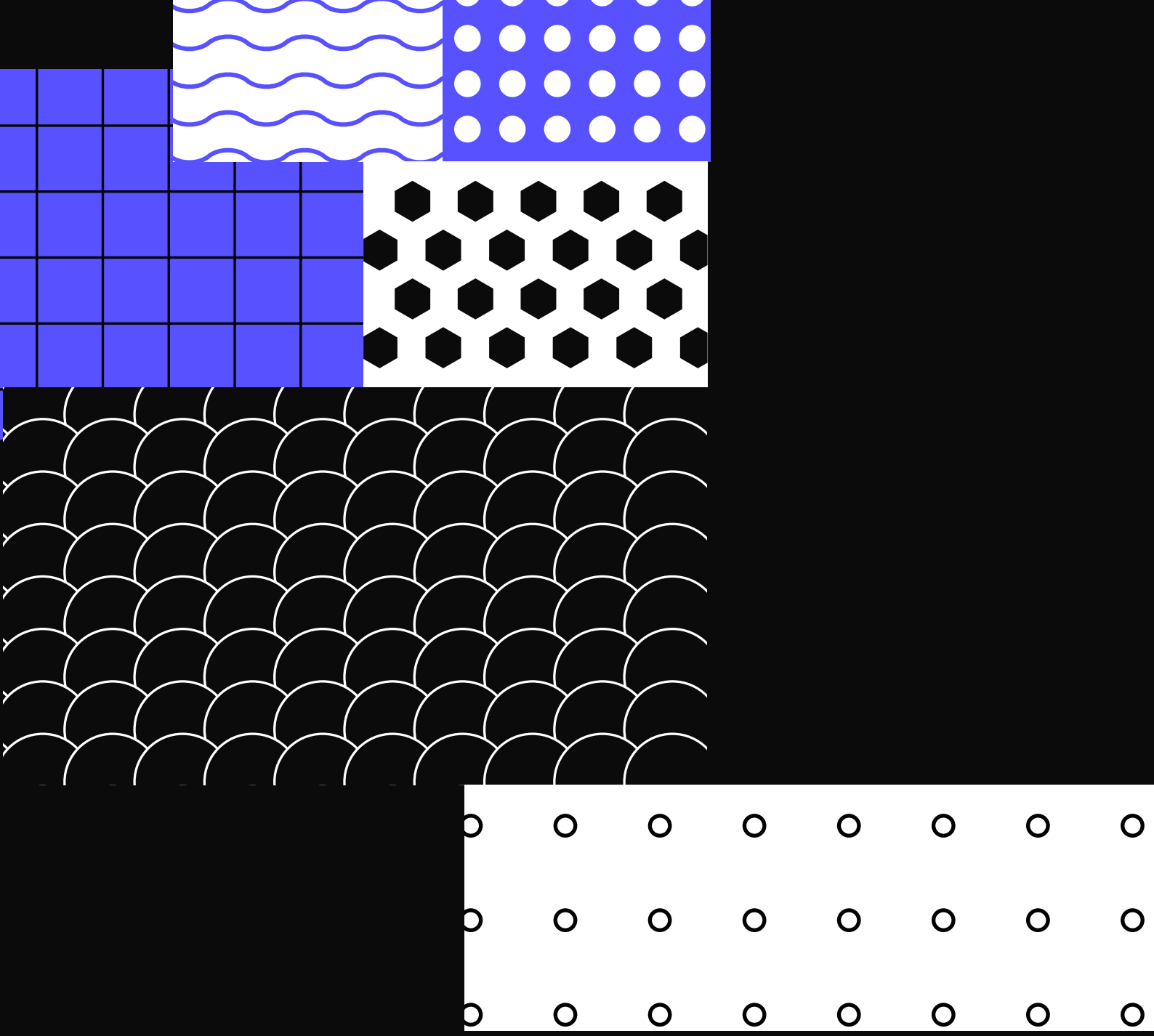
Wang, Yuqing, Maaret Pyhäjärvi, and Mika V. Mäntylä. 2020. “Test Automation Process Improvement in a Devops Team: Experience Report.” In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 314–21. https://doi.org/10.1109/ICSTW50294.2020.00057.

Matheus Marabesi is a computer engineer who is passionate about software development. He is a skilled software craftsperson who enjoys sharing his expertise with others. Matheus is also an aspiring researcher with a focus on software testing and gamification. He has created a gamified tool called Testable to help make learning testing more engaging for software development professionals.

Matheus has been interested in computers since he was young. He started with his first computer running Windows XP, and used to take drawing classes alongside it. However, as he spent more time with his computer, his interest in computing grew and his drawing fell by the wayside. He started attending computer hardware and network training courses, and his interest continued to expand to all aspects of technology. This led him to pursue a degree in engineering.

Currently, Matheus is part of the Codurance Spain Software Craftsperson team. He actively participates in community events such as meetups, coding dojos, and workshops, where he is always willing to share his knowledge and skills. He is also involved in open source projects and forums related to well-designed applications, clean code, testing, and gamification. If you're interested in reading more about his professional contributions, you can check out his blog at https://marabesi.com.

During his free time, he enjoys tinkering with Raspberry Pi, immersing himself in the world of IoT, and going for runs while listening to an electronic music playlist on his headphones.



codurance

hello@codurance.com

codurance.com