1

Preface

The idea for this book appeared after my talk "Refactoring Like a Superhero,"🔗 which I was making in January 2022. For that talk, I've collected various techniques and heuristics of refactoring I wanted to share.

At some point, it became apparent that I couldn't fit everything I wanted to discuss into a 40-minute slot. I had to cut and trim the material for the talk to show the most valuable practices and techniques.

The cut-out material wasn't useless, though. It contained details and clarifications about the techniques' usage and applicability. I decided it would make more sense not to throw out everything that didn't fit but to change the format and release it as an online book. That's how this project came to be.

Who Might Want to Read This Book

This book might be helpful for developers of web services and user applications who write in high-level languages and have a couple of years of experience.

Library developers might also find some ideas, but I'll mainly focus on user applications since I have more experience in that area.

This book doesn't consider the needs and constraints of low-level development. Some heuristics in it might contradict best practices of low-level code. If you write "closer to the hardware," please, be careful, and proceed at your own risk 😃

What This Book Is Not

I don't claim to show "the only correct way" to refactor and write code in this book. If you have a lot of experience, you probably already know most of the techniques described and have your own opinion.

Also, this is not a "step-by-step manual" universally applicable to all projects. My development experience biases my coding habits and programming style. Your experience, projects, and habits may differ from mine, so our views may not be the same. That's okay.

The purpose of this book is to describe a set of practices and heuristics that helped *me* start writing code that *looks good* to me and the people I worked with.

Not all practices have to be applied at all times. Using an idea depends significantly on the project, the context, available resources, and the purpose of refactoring. Try to choose techniques that bring more benefits at less cost.

If something in the book seems helpful to you, discuss it with your colleagues and other developers. Make sure you and your team have the same opinion about the chosen idea's benefits and costs. Don't apply something controversial to your team.

Limitations and Applicability

All of the techniques described here are a compilation of the books I've read and my experience in development. I have spent most of my time developing medium and sometimes large user applications.

My experience imprints on the way I see good code. Basically, the whole book is a snapshot of my perception of development in 2022. My opinion may have changed if you're reading this in the future.

> **By the way** 🐝
>
> I will update the book's text as my opinion changes, but I can't guarantee I will do so promptly and without delay.

As you read the book, be aware of the author's cognitive biases. Weigh the techniques before using them and consider their applicability to your project.

Good to Know Before Reading

In the text, I assume you're already familiar with the concept of refactoring 🔗 and that you have a couple of years of programming experience. I expect you've already encountered issues with task decomposition, "leaky" abstractions, separation of concerns between modules, etc.

I expect you've heard about some of the "programming buzzwords" on this list:

- Separation of concerns
- Coupling and cohesion
- Declarative style

- Abstraction layers
- Command-query separation
- Referential transparency
- Functional pipeline
- Functional core in an imperative shell
- Layered architecture, “Ports and Adapters”
- Direction of dependencies
- Immutability and statelessness
- 12-factor applications

You don’t have to *know* them. We’ll be exploring the terms and techniques as the book progresses. But it’s good if you have a rough idea about their basic meaning.

Why Another Book

There are many books on refactoring, why do we need another one?

By and large, we don’t. All the principles and techniques are described in other books, most likely in a much more detailed, logical, and correct way.

I need this text, first of all, for myself:

- To systematize what I know today;
- To refer to a specific topic when discussing a problem;
- To mark the level of my knowledge so that I understand where to improve and what to learn.

However, I thought that maybe this text could be helpful to someone else, so here we are. I hope you’ll find this book helpful!

---

1. “Refactor Like a Superhero” Talk, https://bespoyasov.me/talks/refactor-like-a-superhero/
2. Code Refactoring, Wikipedia, https://en.wikipedia.org/wiki/Code_refactoring

2

Introduction

Refactoring requires resources. The amount of these resources depends on the size of the project and its code quality. The larger the project and the worse the code, the more difficult it is to clean it up and the more resources it may require.

To justify the investment of resources and find a balance between costs and benefits, we need to understand the benefits and limitations of refactoring.

Benefits for Developers

Code quality is an investment in developers' free time in the future. The simpler and cleaner the code is, the less time we'll spend fixing bugs and developing new features.

Developers may care about different properties of the code. For example, we might want to:

- Find code related to specific parts of the application faster.
- Eliminate misunderstanding about how the code works and avoid miscommunication and conflicts in the team.
- Make it easier to review code and check it against business requirements.
- Painlessly add, change, and delete code without regressions.
- Reduce the time to find and fix bugs and make debugging process more convenient.
- Simplify project exploration for new developers.

This list is incomplete. Other properties may be necessary to a particular team, varying from project to project.

Regular refactoring helps pay attention to code properties before problems appear. It makes daily work more efficient, gives developers extra time and resources, and prevents "big refactorings" in the future.

Benefits for Business

In a perfectly organized development process, there's no need to "sell" refactoring to the business. In such projects, regular code improvement is at the core of the development, and bad code does not accumulate—no need to "explain the benefits to the business" in this case.

However, there are projects where development is organized differently for various reasons. In such projects, as a rule, legacy code tends to accumulate.

We may feel the need to improve the code, but we may not have enough resources to do that. A proposal to "take a week to refactor" might cause a conflict of interests because, to the business, it sounds like "we'll do nothing useful for a week." These are the cases where we may need to "sell" the ideas of the code improvement.

The benefits of refactoring aren't evident to the business because they aren't immediate. We may see them in the future, but it's difficult to predict when.

Usually, to sell the idea of refactoring to business, we should speak the business language, and *sell the result, not the process*. Discuss what exactly we'll get as a result of the time spent:

- We'll spend less time fixing bugs, so the number of unhappy users will drop.
- We'll start implementing new features before our competitors, so they generate new users and profit.
- We'll better understand the requirements and constraints, so we react to unpredicted problems faster.
- We'll make onboarding easier for new developers to make significant changes sooner.
- We'll decrease staff turnover because developers don't run away from good code, only from the bad one.

We can use various metrics to measure code quality. It'll be much easier to determine the necessity of refactoring by relying on the numbers. For example, the costs statistics might help to incorporate regular refactoring into the development process smoothly.

"Good" Code, "Bad" Code

It isn't easy to name a list of *universal* characteristics of a good code. There are a few, but they have limits in applicability, too.

> **For example** 💡
>
> I think of cyclomatic complexity and the number of dependencies as more or less universal characteristics. But we'll talk about them separately in future chapters.

Most of the books I've read also describe good code subjectively.[1][2][3] Different authors use different words, but they always emphasize "readability."

Some studies have tried to determine this "readability."[4][5][6] However, their samples are either small or skewed, so it's difficult to conclude the universal rules of the "good" code.

In practice, we can try to look for a "bad" code rather than a "good" one. It's easier because we can use the help of heuristics and "cognitive alarms" when searching for it.

Cognitive alarms are the feelings we get when reading bad code. I believe that a code needs refactoring if one of these thoughts arises while reading it:

Hard to Read

- It's hard for us to read code if it's unformatted, intertwined, or noisy.
- If there are a lot of unnecessary details in the code, there is no clear entry point.
- If it's hard to follow the code execution, if we need to jump between screens, files, or lines constantly.
- If the code is inconsistent, if it doesn't follow the project rules.

Hard to Change

- Code is hard to change if we need to update many files or double-check the entire application when adding a feature.
- If we aren't sure, we can painlessly remove a particular piece of code.
- If there's no clear entry point or we can't relate a feature with a specific module.
- If there's too much boilerplate code or copypaste.

Hard to Test

- Code is hard to test if we need a "complex infrastructure" for tests or need to mock a lot of functionality.
- If we must emulate the whole app running to check a single function.
- If tests for a task require data irrelevant to the task.

Hard to "Fit in the Head"

- Code doesn't fit in our heads if it's hard to keep track of what's going on in it.
- If by the middle of the module, it's hard to remember what happened at the beginning.
- If the code is "too complicated" and drawing diagrams doesn't help to understand it.

Code Smells

Some of those problems have already been shaped in the form of code smells. *Code smells* are antipatterns that lead to problems.[7]

There are solutions for most of the code smells. Sometimes it's enough to look at the code, find the smell, and apply a specific solution to it.

> **About smells 🐓**
>
> Most often, examples of code smells are given in code written in OOP style, which may not be as valuable in the JavaScript world. Nevertheless, some of the smells are universal and applicable to OOP and multi-paradigm code.

---

1. "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

2. "The Art of Readable Code" by Dustin Boswell, Trevor Foucher, https://www.goodreads.com/book/show/8677004-the-art-of-readable-code

3. "Clean Code" by Robert C. Martin, https://www.goodreads.com/book/show/3735293-clean-code

4. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies, https://github.com/reydne/code-comprehension-review/blob/master/list-papers/AllPhasesMergedPapers-Part1.md

5. Code Readability Testing, an Empirical Study, https://www.researchgate.net/publication/299412540_Code_Readability_Testing_an_Empirical_Study

6. How Readable Code Is, a Readability Experiment https://howreadable.com

7. Code Smells, Refactoring Guru, https://refactoring.guru/refactoring/smells

3

Before Refactoring

To refactor code faster, without regressions, and without "too much extra work," we need to first *study and prepare* the code. In particular, we should:

- Define the refactoring scope and boundaries.
- Cover the code under refactoring with tests.
- Configure linters and, if needed, the compiler more strictly.

In this chapter, we'll discuss why these steps are helpful and how to make them simpler.

Define Scope and Boundaries

Refactoring should not take forever. Instead, it should be a little chunk of work with clear boundaries and a definition of done. It's essential for two reasons:

- We want to stay within our *time and resource budget*.
- Limited changes are *easier to keep track of* to see faster if they break the app.

Refactoring boundaries define the scope of work. They are the junction points between the code we're going to change and everything else. They show where our changes should stop, that is, what code *won't* change.

Defining boundaries in the code isn't always obvious, especially if the modules are tightly coupled. In such cases, we should pay attention to the code's *data and dependencies*.

The more different the data two pieces of code work with, the more likely they are different, independent parts of the program. The junction of these pieces is the boundary that should prevent changes from spreading across the code base.

> **By the way**
>
> Feathers in "Working Effectively with Legacy Code" calls these junctions "seams."
> Sometimes, I will use this term as a synonym.

Boundaries prevent refactoring from taking forever and help you integrate changes into the repository's main branch more often.

> **In detail** 🔬
>
> We'll talk a little more about finding and using boundaries in the following chapters.

Cover with Tests

We need to cover the code inside the boundaries with tests. We'll use them to check if something was broken during refactoring constantly. To make more use of the tests, we should meet several conditions:

Find More Edge Cases

Edge cases help avoid regressions and ensure we haven't broken the code behavior in "exotic" circumstances. ("Exotic" bugs are much harder to fix.🔗)

The more diverse the edge cases, the easier it is to prepare and systematize test data for different situations. The knowledge of the application's behavior in these situations will come in handy in the future, even after refactoring.

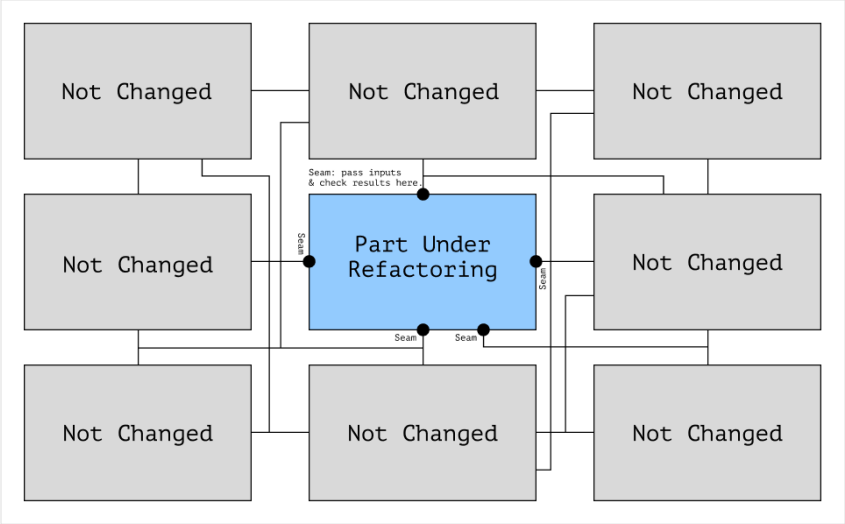
Define Explicit and Implicit Input

The explicit input is the arguments of functions or methods. The implicit input is dependencies, shared or global state, and the context of functions and methods. Systematization of input data will simplify test case creation.

Specify Desired Result

During refactoring, we can't change the code functionality, so the desired result is the actual current behavior of the program. We can capture it as output data (e.g., the result of a function) or as a desired side effect (e.g., state change or API call).

Ideally, the result should exist *on the boundary* of code under refactoring. Checking results on the boundary is easier, while the amount of affected code is minimal.



Result on the border is usually easier to check

Set Up Automatic Tests

When refactoring, we want to test *every* change, even a minuscule one. Manual testing can get tiresome. We might get lazy or forget to test the changes with manual testing.

Automatic tests don't have this problem. They can run in parallel while refactoring and re-test the code each time we save a file. That way, the test results are always in front of our eyes, and we can notice faster what change has broken the code.

What type of tests to use depends on the situation and isn't as important as their existence. If unit tests are enough, we can use only them. If we have to test the work of several modules, we may need an integration or E2E test. The main point here is *automation*. The fewer checks we do by hand, the fewer human errors appear.

Make Linter and Compiler Settings Stricter

This section is somewhat optional, but I like using it. In my opinion, more aggressive linter and compiler settings help spot occasional bugs and bad practices. "More aggressive" settings can include:

Use "Errors" Instead of "Warnings"

Linter warnings are a collective experience of the industry. They can be valuable, but they're easy to miss because they don't make the code compilation "crash with an error."

With errors, on the other hand, the code "won't compile." Errors will force us to update the code or change our linter rules.

> **Keep in mind !**
>
> Not all linter rules are equally helpful. We can choose rules that we think are more important and discard others. However, once we have selected a set of rules, we should follow them strictly.

Configure More Automated Checks

It is a great time to try out new linter rules or other automated tools if the team wants to try them out.

But it's worth remembering that not all rules are equally valuable and appropriate for every project. We should try not to go against the team. If most of the other developers in the team think a particular linter rule is unnecessary, we shouldn't introduce it.

Teams can debate rules and practices to determine the most suitable for them. But remember that no rule is worth having a conflict in the team.

---

1. "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

2. "Debug It!: Find, Repair, and Prevent Bugs in Your Code" by Paul Butcher, https://www.goodreads.com/book/show/6770868-debug-it

4

During Refactoring

After we've defined clear boundaries, examined the code, and covered it with tests, we can start refactoring.

As we work, we want the changes to be as helpful as possible while staying within our boundaries. In this chapter, we'll discuss heuristics that help us do this.

Small Steps

A small step is a minimal change that makes sense. An excellent example of a small step is an *atomic commit*. Such a commit contains a meaningful functionality change and evolves the code from one working state to another. We can develop and change the code base with atomic commits while keeping it valid at every point.

> **By the way** 💡
>
> Atomic commits allow us to bisect the repository in the future while searching for bugs. When code is valid and can compile in each of the commits, we can "time travel" through the repository by switching between different commits.

The size of commits depends on the habits and preferences of the team. Personally, my rule of thumb is "the smaller the commit, the better." For example, I commit separately even function and variable renames on my projects.

Smaller steps encourage us to decompose tasks into simpler ones. This way, extensive features turn into sets of compact tasks, which are not so scary to merge into the main repository branch more often. Without decomposition, such a task could turn into a blocker that drags the whole team down.

> **About CI** 🔬
>
> The point of *continuous integration, CI* is for the team to synchronize code changes with each other as often as possible. This approach and its benefits are described well in "Code That Fits in Your Head" by Mark Seemann and "Beyond Legacy Code" by Scott Bernstein.🔗🔗

Blocking tasks should be avoided in general, but especially when refactoring code. If the whole team is busy with refactoring, it can become expensive. The precedent of being expensive can deprive the project of resources for refactoring at all in the future.

In addition, small steps allow us to "postpone" refactoring at any time and switch to another task. If we work with git, we can use "stash" to save unfinished work via `git stash`. That way, the developers become more flexible and respond faster to sudden bugs and other problems.

Also, the results of small steps are easier to check and evaluate before committing. Such checks help us filter out changes that are not relevant to the current task. For example, they can help us spot accidental changes caused by automated formatters or linters.

And lastly, small steps are easier to describe in commit messages. The scope of such changes is less, so it is easier to explain their meaning in a short sentence.

Small but Detailed Pull Requests

This section follows the previous one, but I want to focus on it separately. The problem with big pull requests is that:

> **No one reviews big pull requests ❗**

If we want to *improve* the code, we need the pull request to *be reviewed*. So our job is to make the reviewer's job easier. To do this, we can:

- Limit the size of changes. So it's easier to find time for a review and understand the PR's goals and meaning. This way, the code review won't look like a big chunk of new work.
- Describe the context of the task in the message to the PR. The reasons, goals, and constraints of the task will help share the understanding of the task with the reviewer. This way, we can anticipate likely questions and answer them in advance. It will speed up the review.

The desire for small but detailed PR also helps break down large tasks into smaller ones and refactor them in small steps.

Tests for Every Change

For the code to evolve through valid states, we will test *every* change, no matter how small.

When using unit tests, we can keep them running next to the editor. When using longer-running tests (like E2E), we can run them before each commit (e.g., on a pre-commit hook) so that invalid code doesn't get into the repository.

The tests should drive us to *commit only valid code*. Each commit will contain a set of *complete and meaningful* changes.

> **By the way** 🧪
>
> It will only work if tests are trustworthy. That's why we emphasized edge cases and detailed specifications of the result in the latest chapter. That help make tests more reliable.

One Technique at a Time

Frequent commits are anchor points in code evolution. The more frequent such points are, the more compact the changes in between. It is useful, for example, when examining changes from the latest commit we just made. Small changes are faster to study, easier to understand, and contain less work, so it's emotionally easier to roll them back in case we need to.

During refactoring, though, it's not always obvious how to make changes smaller and more often. I prefer to follow this rule:

> **Don't mix different refactoring techniques in the same commit ❗**

This rule makes us commit code more often: we renamed a function—made a commit, extracted a variable—made a commit, added code for a future replacement—made a commit, and so on.

As long as we don't mix different techniques in one commit, it's easier for us to track code changes by diffs and find errors like name conflicts.

Complex techniques that are too big for a single commit can be broken up into steps. We can commit each of these steps separately. But when splitting the task, we should remember that each step must leave the code in a valid state.

> **By the way** 👀
>
> Such a combination of atomic commits, continuous integration, and checks before committing changes is sometimes called the "tactical" git.🔗 We will occasionally use this term later as well.

No Features, No Bug Fixes

During refactoring, we may find an idea for a feature or a piece of code that doesn't work correctly. We may want to "fix it along the way," but it's better not to mix bug fixes and new features with refactoring.

Refactoring *should not* change the code functionality. If, during refactoring, we add a feature, and later it needs to be rolled back, we'll have to cherry-pick specific commits or even lines of code into the repo manually.

Instead, it's better to put all the found feature ideas into a separate list and return to them after refactoring. If we find a bug, we should postpone refactoring (`git stash`) and return to it after the fix. Again, working in small steps is more convenient for this maneuverability.

Transformation Priority Premise

Transformation Priority Premise, TPP is a list of actions that help develop code from the naive, most straightforward implementation to a more complex one that meets all the project requirements.🔗

TPP helps *avoid doing everything at once*. It encourages us to gradually update a piece of code, recording each step in the version control system, and integrating changes into the main repository branch more often.

For me, TPP helps me not overload my head with details while writing code. All improvement ideas that come up along the way, I put in a separate list. I then compare this list with TPP and choose what to implement next.

> **Why bother** 🧠
>
> "Offloaded" details free up the brain's "working memory."🔗 The freed resources can be spent on the task details. They will improve attentiveness and focus.

Refactoring Tests Separately

Tests and the application code cover each other. Tests verify that we haven't made mistakes in the application code and vice versa. If we refactor them simultaneously, the probability of missing a bug becomes higher.

It's better to refactor the application code and test code one at a time. If, while refactoring an application, we notice that we need to refactor a test, we should:

- Stash the application code changes;
- Refactor the desired test;
- Verify that the test breaks for a reason it should;
- Get the last changes from the stash and continue working on them.

> **In detail** 🔬
>
> We'll talk a little more about this technique in the "Refactoring Test Code" chapter.

---

1. Atomic Commit, Wikipedia https://en.wikipedia.org/wiki/Atomic_commit
2. git-bisect, Use binary search to find the commit that introduced a bug, https://git-scm.com/docs/git-bisect
3. "Write Better Commits, Build Better Projects" by Victoria Dye, https://github.blog/2022-06-30-write-better-commits-build-better-projects/
4. "Code That Fits in Your Head" by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
5. "Beyond Legacy Code" by David Scott Bernstein, https://www.goodreads.com/book/show/26088456-beyond-legacy-code
6. "Use Git Tactically" by Mark Seeman, https://stackoverflow.blog/2022/04/06/use-git-tactically/
7. Transformation Priority Premise, Wikipedia https://en.wikipedia.org/wiki/Transformation_Priority_Premise
8. Working memory, Capacity, Wikipedia, https://en.wikipedia.org/wiki/Working_memory#Capacity

5

Low-Hanging Fruit

Refactoring a piece of code can require various changes, and it can be challenging to choose how to start. To solve this, we can rank the changes from simple to complex and begin with the simplest ones. It helps to "blow the dust off the code" and start to see more severe problems in it.

By simple changes, we mean code formatting, linter errors, and replacing self-written code with features of the language or environment. In this chapter, we'll discuss these improvements and see why they're helpful at the beginning of refactoring.

Code Formatting

Code formatting is a matter of taste, but it has one useful function. If the code in the project is *consistent*, it takes less time for readers to understand it. That's how habits work: the familiar "shapes" of code help us focus on the meaning instead of characters and words.

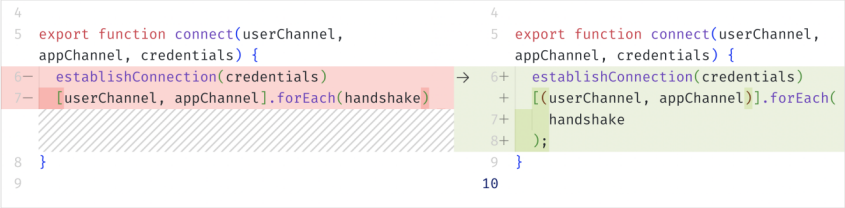
```
// Unformatted code.
// We have to focus harder to see its meaning:

function ProductList({ products }) {
   return <ul>{products.map((product) =><li key={product.name}>
       <Product product={product} /></li>)}</ul>}


// Formatted code makes it easier to read it:

function ProductList({ products }) {
  return (
    <ul>
      {products.map((product) => (
        <li key={product.name}>
          <Product product={product} />
        </li>
      ))}
    </ul>
  );
}
```

It's better to automate code formatting. In the example above, I used automatic code formatted called Prettier,[🔗] but the particular tool is not as important here as the overall approach. If the team is not satisfied with Prettier, we can choose another formatter and use it. The point is to *automate the process*.

However, sometimes the formatter might break the code, for example, when it doesn't consider ASI (Automatic Semicolon Insertion)[🔗] in JavaScript:

```
// Before applying formatter:

export function connect(userChannel, appChannel, credentials) {
  establishConnection(credentials)
  [userChannel, appChannel].forEach(handshake)
}

// After:

export function connect(userChannel, appChannel, credentials) {
  establishConnection(credentials)[(userChannel, appChannel)].forEach(
    handshake
  );
}
```

If we use git "tactically" and check all the changes since the previous commit, we may notice this error in the diff ourselves:



Git shows all changes the formatter caused

...But looking for such errors only manually is unreliable, so it is better to automate this search as well.

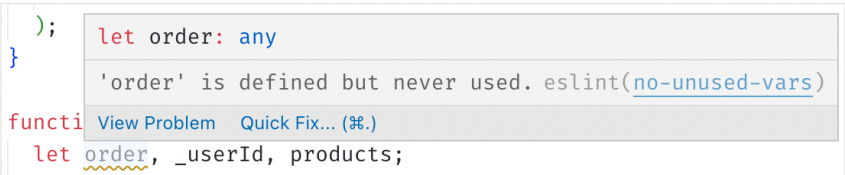
The most reliable strategy is to cover the code with a set of tests before using the formatters. If the tests run beside the editor, we will instantly see what exactly was broken by the formatter. Then we will need to check the results of using automatic tools manually less often.

Applying a formatter can be a refactoring technique on its own so the result can be a commit or even a separate PR. Our main goal is to integrate the changes into the main branch as early as possible so we don't have to handle complex merge conflicts between our changed formatting and updates made by other developers.

Code Linting

After turning linter "warnings" to "errors," we might have a list of such errors. This list can be a task list for the current refactoring iteration.

I prefer to save the work on each linter rule as a separate commit or PR. For example, we could remove all unused code, make it a commit, and move on to the next problem on the list.



Linter highlights unused code that can be removed

If the linter shows many errors, we can turn the rules one by one rather than all simultaneously. The smaller the steps, the easier it is to break the problem into several and solve each separately.

After fixing each rule, we'll need to check if the tests pass. In the future, I will stop emphasizing test-checking to shorten the text. Let's keep in mind that we check the tests after *each* change.

Language Features

Modern languages evolve and get new features. It is especially true for JavaScript since the ES specification is updated yearly.

Sometimes, a new language version feature can replace old self-written helper functions. Built-in language features are faster, more reliable, and easier to work with. So if there's a chance for replacement we can use it.

> **By the way** 🧯
>
> On the frontend, we might need to ensure the feature has the necessary browser support. We can check it with Caniuse.

```JavaScript
// Self-written helper for checking the beginning of a string:
const startsWith = (str, chunk) => str.indexOf(chunk) === 0;
const yup = startsWith("Some String", "So");

// ...Can be replaced with the native string method:
const yup = "Some String".startsWith("So");
```

> **However** 💡
>
> If the self-written implementation differs from the native one and we can't replace it, I'd prefer to mention the difference in the documentation. That way, it will be clear why we're using a self-written function instead of the language feature.

Standard API Features

In addition to language features, it's helpful to keep in mind functions from the standard library and built-in APIs. The right function or data structure can make code more efficient, cleaner, and shorter. For example, in the code snippet below, we can simplify self-written form serialization using `FormData`:

```JavaScript
// We can replace manual value extraction:
const username = form.querySelector('[name="username"]').value;
const password = form.querySelector('[name="password"]').value;
const data = { username, password };

// ...With a standard API call:
const data = Object.fromEntries(new FormData(form));
```

After the changes, there is less code, and it is now more resistant to changes. For example, after refactoring, we don't need to manually update serialization when adding a new field to

the form:

```JavaScript
// Before, we needed to update the `data` object by hand:
// ...
const email = form.querySelector('[name="email"]').value;
const data = { username, email, password };

// Now `FormData` updates the data automatically,
// no need to change the code manually.
```

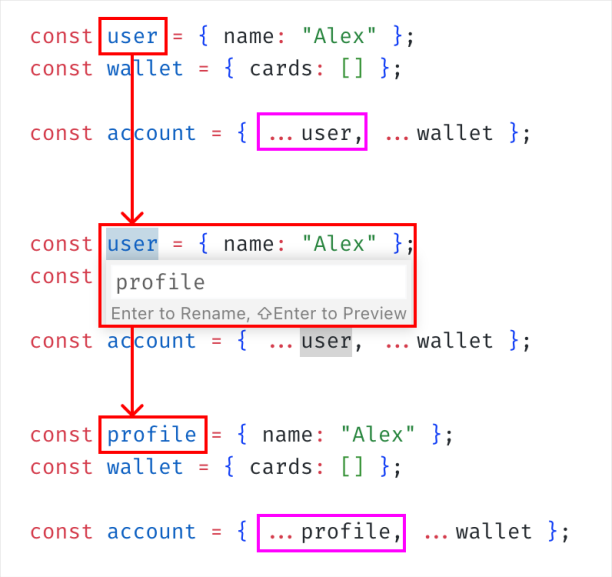
Removing code is beneficial: the less code there is, the fewer potential points of failure in the application. We can use the rule "give most of the work to the language or environment than write ourselves." It's usually more reliable.

> **Clarification** 🤓
>
> There are cases when we can't use only the standard APIs and have to reinvent the wheel. In these cases, we should document why the standard solutions do not fit.
>
> We will discuss in detail how to make the documentation and comments more informative and useful in one of the following chapters.

Environment Features

Along with the language features, we should also highlight the features of the text editor or IDE we're working with. If they have automated refactoring tools, it's worth learning them.

"Rename Symbol," "Extract into Function," and other tools speed up the work and reduce the cognitive load. For example, in VS Code, we can change the name of a function or variable everywhere by using the hotkeys:🔗

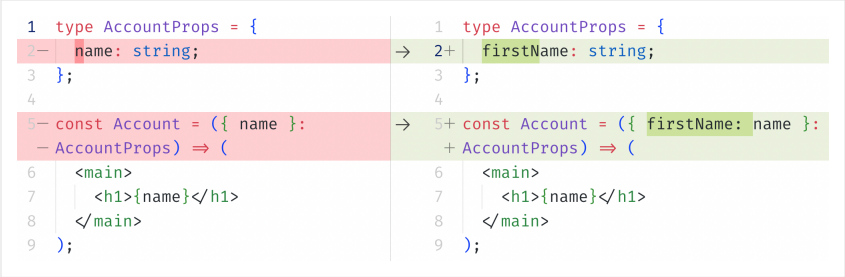


"Rename Symbol" updates the variable name everywhere

However, we should double-check the result of applying these tools. For example, Rename Symbol may "miss" some name or add unnecessary renaming:

```
// For example, we want to replace `name`
// with `firstName` in the `AccountProps`:

type AccountProps = { name: string };
const Account = ({ name }: AccountProps) => <>{name}</>;

// After applying "Rename Symbol,"
// there might appear an "extra renaming":

type AccountProps = { firstName: string };
const Account = ({ firstName: name }: AccountProps) => <>{name}</>;
```

To avoid this, we can use the benefits of "tactical" git again. We can study the diff from the latest commit and check what was renamed and how.



Git shows exactly what has changed since the latest commit

A strategy of small steps helps simplify and maximize such checks' benefits. If we apply only one refactoring technique per commit, there's no noise in the diffs, and we can better see how the changes affect the code.

Linters and tests help us avoid name conflicts and other bugs. For example, we can set up rules that forbid identical variable names, so the linter will error if there's a name conflict. If the linter is running in the background with refactoring, we will see the error immediately and be able to fix it.

---

1. Prettier, an opinionated code formatter, https://prettier.io

2. "Automatic semicolon insertion in JavaScript" by Dr. Axel Rauschmayer, https://2ality.com/2011/05/semicolon-insertion.html

3. List of EcmaScript Proposals, https://proposals.es

4. Can I use, support tables for the web, https://caniuse.com

5. "Use the Right Algorithm and Data Structure" by JC van Winkel, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_89/

6. "How to Convert HTML Form Field Values to a JSON Object" by Jason Lengstorf, https://www.learnwithjason.dev/blog/get-form-values-as-json

7. Refactoring Source Code in VSCode, https://code.visualstudio.com/docs/editor/refactoring

6

Names

After we've "blown the dust off the code," we're ready to consider more severe problems.

The following improvements and techniques we're about to discuss are much harder to rank from simple to complex. It's not always obvious when and which technique needs to be applied.

> **Chapter order** 🎯
>
> I suggest thinking of this and the following chapters not as a "step-by-step manual" but rather as an unordered list of problems and possible solutions.
>
> The chapters in the book are arranged so that the size of suggested code changes gradually grows. But there's no guarantee that the problems will occur in this order when you refactor the real project.
>
> Be careful when analyzing your code. Note and write down the issues in the way you feel most comfortable with, and use this book as supporting material.

Attention to the names of variables, functions, classes, and modules can be Pandora's box in the world of refactoring. "Unclear" names can signal high coupling between modules, inadequate separation of concerns, or "leaking" abstractions. This chapter will discuss how obscure names can help us find problems in our code.

General Guidelines

An "obscure" name can hide valuable project details and the experience of other developers. We need to preserve them, so it's worth diving into the meaning of such a name rather than discarding it altogether. If during refactoring, we see a name whose meaning isn't clear to us, we can try:

- Find someone on the team who knows the meaning of that name;
- Search for the meaning in the code documentation;

- As a last resort, do a series of experiments, changing the variable and seeing how it affects the code.

> **By the way** 🔬
>
> If the effect of a variable is visible on the "seam," tests can help us with such experiments. The test results will show how the different variable values change the code output. It can help us conclude the real variable purpose.
>
> In other cases, we will probably have to evaluate the changes manually. As an option in the debugger, we can observe how the different values affect the other variables and data the code works with.
>
> Such experiments will not *guarantee* that we understand the meaning of the variable correctly,🔗 but they can tell us what knowledge about the project we lack.

It's better to keep the gained knowledge directly in the code by renaming a variable. If we can't do this, it's worth keeping it *as close to the code* as possible. We can use a comment, documentation, a commit message, or a ticket description. The point is to allow developers to find and use this information *easier*.

Too Short Names

We can consider a name "good" if it adequately represents the meaning of the variable or information about the domain. Too short names and uncommon abbreviations don't do that. Sooner or later, someone will misread such a name because all the "knowledgeable" developers will stop working on the project.

> ```
> JavaScript
> ```
>
> ```
> // Names `d`, `c`, and `p` are too short,
> // so it's hard to reason about their meaning:
>
> let d = 0;
> if (c === "HAPPY_FRIDAY") d = p * 0.2;
>
> // Variables with full-word names
> // are easier to understand:
>
> let discount = 0;
> if (coupon === "HAPPY_FRIDAY") discount = price * 0.2;
> ```

One-letter names are okay in concise pieces of code, like `for` -loops. But for business logic, it's better to use more descriptive words.

The same goes for abbreviations. I prefer not to use abbreviations in code except in 2 cases:

- If the abbreviation is a well-known one (e.g., USA, OOP, USD);

- If it's used in this exact form in the domain (e.g., Challenge Rate as CR in encounter calculator for D&D).

In other cases, it's better to choose full word forms for the name:

```JavaScript
// Okay, USD is a common abbreviation:
const usd = {};

// Might be okay if the domain is related to maths
// and the program works with derivatives:
const dx = 0.42;

// Not okay; should use the full form:
const ec = 0.6188;

// Much better:
const elasticityCoefficient = 0.6188;
```

Too Long Names

Too long names signal that the entity is doing too many different things. The key word here is *"different"* because non-cohesive functionality is the most difficult to combine in a single name.

When functionality is non-cohesive, the name tries to convey all its work details in a single phrase. It bloats the name and makes it noisy. We should pay attention to names that contain words like `that`, `which`, `after`, etc.

Most often, long names are a signal of a function that does too much. Such a function most likely uses either too primitive or too abstract terms, and it can't find the right words to express its meaning. The primary heuristic for finding such functions is to devise a shorter name for the function. If we can't do that, the function is probably doing too much.

```JavaScript
async function submitOrderCreationFormIfValid() {
  // ...
}
```

The `submitOrderCreationFormIfValid` function from the example above does three things at once:

- It handles the form submission event;
- Validates the data from the form;
- Creates and sends a new order.

Each step is important enough to be reflected in the name, but this bloats the name. It's better to think about how to split the task into smaller ones and separate the responsibility between the individual functions:

JavaScript

```
// Serializes form data into an object:
function serializeForm() {}

// Validates the data object:
function validateFormData() {}

// Create a new order object from the data:
function createOrder() {}

// Sends the order to the server:
async function sendOrder() {}

// Handles the DOM event by calling other functions:
function handleOrderSubmit() {}
```

Then, instead of one big function trying to do *everything*, we would have a chain of different, smaller ones. Inner functions would contain actions *grouped by meaning*. Their names could take some of the details from the root function name, which would make it easier:

JavaScript

```
async function handleOrderSubmit(event) {
  const formData = serializeForm(event.target);
  const validData = validateFormData(formData);
  const order = createOrder(validData);
  await sendOrder(order);
}
```

It seems like we're making the root function name less accurate because we're taking the details out of it. But it's all a matter of what kind of details we take away.

A good tactic is to see the function name from the *caller's* perspective. Does the form care *how exactly* its submission is handled? Probably not; it only matters that the submission is handled in principle. The handler might want to express *what form* it's going to handle in its name:

```
handle          +
submit event    +
on order form   =
----------------
handleOrderSubmit
```

…But *how exactly* this happens is implementation details of `handleOrderSubmit`. These details aren't actually needed in the name because they are only important inside the function. And there, they can be expressed with inner function names.

> **By the way** 👀
>
> You probably recognized this as an example of separating abstraction layers.🔗 We'll talk more about this in one of the following chapters.

So basically, when we see a long function name, we should figure out precisely what it's trying to tell us. We can try and take out all the details the name carries and divide them into "important outside" and "important inside."

We better keep the details from the first group in the original function name and represent the details from the second group as inner function names. It helps to break down the task into simpler ones and group related functionality together.

> **By the way** 🛠️
>
> The logic of the function itself may still be complex. But there are usually fewer problems with naming if related details are grouped together.

Option for Naming Functions

Regarding functions, the A/HC/LC pattern helps maneuver between "too short" and "too long" names.🔗 This pattern suggests combining the action, its subject, and its object:

```
prefix? + action (A) + high context (HC) + low context? (LC)
```

We can use it as a reference or starting point for naming functions. It's not always possible to follow the pattern, so we can deviate from it and change the name if needed.

Different Entities with Identical Names

During refactoring, we should also consider the feeling of "confusion." It can occur when different entities in the code have identical names.

When reading, we mostly rely on the names of classes, functions, and variables to understand the meaning of the code. If the name doesn't correlate with a variable 1 to 1, we must figure out its meaning on the fly every time. To do this, we must remember a lot of "meta-information" about variables. It makes code harder to read.

> **By the way**
>
> In some cases, the same applies to different names for the same variable. For example, it's better to avoid calling domain entities different names, but more on that later.

To make the code easier to read, it's better to use *different* names for *different* entities:

```JavaScript
// Here, `user` is an object with user data:
function isOldEnough(user, minAge) {
  return user.age >= minAge;
}

// And here, `user` is a user name:
function findUser(user, users) {
  return users.find(({ name }) => name === user);
}

// At first glance, it's hard to tell them apart,
// because the identical names force us to think
// that the variables have the same meaning.
// This can be misleading.

// To avoid this, we have to either remember
// the difference between how each variable is used,
// or express the meaning of the variable
// more precisely directly through its name:
function findUser(userName, users) {
  return users.find(({ name }) => name === userName);
}
```

Identical names for different entities are especially dangerous if they're close. The close location creates a "common context" for the variables, reinforcing the feeling of "sameness."

> **However** 🧐
>
> There are exceptions to this rule, of course. For example, we can infer the distinction from the types or the use context. Although, *by default*, it's better to use different names for different entities.

Ubiquitous Language

Ubiquitous language can help us fight naming problems. It is a set of terms that describe the domain, which the whole team uses. "The whole team" includes developers and the product team, designers, product owners, stakeholders, etc.

> **By the way** 🎙️
>
> The term comes from the *Domain-Driven Design, DDD* methodology.🔗 I find it helpful in describing business logic, you might find it helpful too.

In practice, ubiquitous language tells us that if "business people" use the term "Order" to describe orders, we should use this word in our code, tests, documentation, and verbal

communication.

The power of ubiquitous language is *unambiguity*. If everyone calls a thing by the same name, we'll lose less in the "translation between business and development."

> **Read more** 📚
>
> More about this you can find in "Domain Modelling Made Functional" by Scott Wlaschin.🔗 I highly recommend reading it.

When we use terms close to reality, the mental model of a program becomes closer to *what* it's modeling. In this case, it's much easier to spot incorrect program behavior.

Lying Names

Sometimes during refactoring, we can find names that aren't accurate. Inaccuracy may vary from slight discrepancy to complete lies. We should fix false names as soon as possible.

If we're unsure how to call the variable properly, we should write down why the current name doesn't fit. For example, if we come across code like this:

```JavaScript
const trend = currentValue - previousValue;

// "Trend" here is the difference between
// "current" and "previous" values.
```

...And we've noticed in conversations that the team uses the term "Delta" (not "Trend"), then it's worth mentioning this inaccuracy. We might say something like:

> "
>
> - The term "Trend" probably doesn't accurately describe the value;
> - The team more often uses the word "Delta," including product owners;
> - The project specifics may require trends built from more than two points.

We can raise these concerns to the team and consider renaming them. If the name obviously lies, we can skip the discussion. But when in doubt, it's worth discussing the concern with other developers and the product owner first.

Domain Types

In languages with static typing, we can use types to convey details about the domain. The types take the entity name's load by putting some details into the type. The name becomes more succinct, but it hardly loses any meaning because of the type.

> **By the way** 💬
>
> We could argue that the type is only visible in signatures or IDE popups, while the name is always visible.
>
> Yes, but multiple longer names would make the code noisy, and their value would disappear. And secondly, I usually extract details that aren't needed *instantly* and can be painlessly looked up in the type.
>
> For me, it seems like a perfectly acceptable compromise.

For example, it's convenient to use types to describe various data states that will appear in the application throughout its life cycle.

```TypeScript
type CreatedOrder = {
  createdAt: TimeStamp;
  createdBy: UserId;
  products: ProductList;
};

type ProcessedOrder = {
  createdAt: TimeStamp;
  createdBy: UserId;
  products: ProductList;
  address: Delivery;
};
```

These types describe different data states (essentially different entities) with different names. But also disallow using the "wrong" type where it doesn't fit. For example, we can forbid attempts to send unprepared orders:

```TypeScript
function sendOrder(order: ProcessedOrder) {
  // ...
}

const order: CreatedOrder = {
  /*...*/
};

// Function call below won't compile
// because the argument type doesn't satisfy the signature.
// It can be translated as: "The order isn't ready yet to be sent."
sendOrder(order);
```

> **Boolean flags?** 🚩
>
> We will discuss why using Boolean flags for denoting different states is not as good as different types in the chapter about static typing.

1. Post Hoc Ergo Propter Hoc, Wikipedia, https://en.wikipedia.org/wiki/Post_hoc_ergo_propter_hoc

2. Abstraction layer, Wikipedia, https://en.wikipedia.org/wiki/Abstraction_layer

3. Naming functions, A/HC/LC Pattern, https://github.com/kettanaito/naming-cheatsheet#ahclc-pattern

4. "Domain-Driven Design" by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design

5. "Domain Modeling Made Functional" by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional

7

Code Duplication

The main goal of refactoring is to make the code more readable. One way to achieve this is to reduce the amount of noise in it.

Code duplication makes the code noisy because it contains no useful information. However, not all duplication is evil, and it even can be a development tool. So when refactoring, we should understand what kind of duplication we're dealing with.

In this chapter, we'll discuss what to look for when searching for duplication and how to know when it's time to eliminate it.

Not All Duplication is Evil

If two pieces of code have the same purpose, contain the same set of actions, and process the same data, that is *direct* duplication. We can safely get rid of it by extracting the repeating code into a variable, function, or module.

But there are cases when two pieces of code seem "similar" but later turn out different. If we merge them too early, it'll be harder to split such code later. In general, it's much easier to combine identical code than to break modules merged earlier.

When we're not sure that we're facing two *really* identical pieces of code, we can mark these places with unique labels. We can write an assumption about what's duplicated there in these labels.

JavaScript

```
/** @duplicate Applies a discount coupon to the order. */
function applyCoupon(order, coupon) {}

/** @duplicate Applies a discount coupon to the order. */
function applyDiscount(order, discount) {}
```

Such labels will only be helpful if we conduct their regular reviews. During the review, we should check if we have learned something new about possible duplicates, which either

confirms that they're identical or disproves it by showing the difference between them.

> **By the way** ⏰
>
> I consider regular audits in my projects as part of paying the technical debt. I make to-do lists for such periodic tasks. Within the list, I specify what needs to be done within the task. The technique of regular audits and its benefits are well described in "Jedi Techniques" by Maxim Dorofeev.🔗

If it becomes apparent during an audit of a label that the duplication described in it is *direct*, we can proceed with refactoring code with that label. If the code turns out to be different, we can remove the label. This approach helps not to rush with code generalization but also not to forget places where possible duplication exists.

Variables for Data

Duplicated data or calculation results are convenient to put into variables. For example, it helps unravel complex conditions or highlight the steps of data transformations.

```JavaScript
// If conditions are relatively close to each other:

if (user.age < 18) toggleParentControl();
// ...
if (user.age < 18) askParents();

// ...We can extract the expression into a variable:

const isChild = user.age < 18;

if (isChild) toggleParentControl();
// ...
if (isChild) askParents();
```

Sometimes the data duplication can be less obvious and more difficult to spot:

JavaScript

```
// The second condition is turned “inside out”
// and uses the `years` variable instead of the object field.

if (user.age < 18) askParents();
// ...
const { age: years } = user;
if (years >= 18) askDocuments();

// But we can still get rid of it,
// by extracting the expression into a variable:

const isChild = user.age < 18;

if (isChild) askParents();
// ...
if (!isChild) askDocuments();
```

More info 🔬

We’ll talk about simplifying and unraveling complex conditions in more detail in one of the following chapters.

Functions for Actions

We can extract duplicated actions and data transformations into functions and methods. To detect them, we can use the “sameness check”:

- Actions are duplicated if they have the same goal—the desired result;
- Have the same scope—the part of the application they affect;
- Have the same direct input—arguments and parameters;
- Have the same indirect input—dependencies and imported modules.

In the example below, the code snippets pass this test:

JavaScript

```
// - Goal: to add a field with the absolute discount value to the order;
// - Scope: the order object;
// - Direct input: order object, relative discount value;
// - Indirect input: function that converts percents to absolute value.

// a)
const fromPercent = (amount, percent) => (amount * percent) / 100;

const order = {};
order.discount = fromPercent(order.total, 50);

// b)
const order = {};
const discount = (order.total * percent) / 100;
const discounted = { ...order, discount };

// Actions in “a” and “b” are the same,
// so we can extract them into a function:

function applyDiscount(order, percent) {
  const discount = (order.total * percent) / 100;
  return { ...order, discount };
}
```

In the other example, the goal and direct input are the same, but the dependencies are different:

JavaScript

```
// The first snippet calculates discount in percent,
// the second one applies the “discount of the day”
// by using the `todayDiscount` function.

// a)
const order = {};
const discount = (order.total * percent) / 100;
const discounted = { ...order, discount };

// b)
const todayDiscount = () => {
  // ...Match the discount to today's date.
};

const order = {};
const discount = todayDiscount();
const discounted = { ...order, discount };
```

In the example above, fragment “b” has the `todayDiscount` function among its dependencies. Because of it, the action sets differ enough to be considered “similar” but not “the same.”

We can use `@duplicate` labels and wait a bit to get more information about how they should work. When we know exactly how these functions should work, we can “generalize”

the actions:

```JavaScript
// The generalized `applyDiscount` function will take
// the absolute discount value:

function applyDiscount(order, discount) {
  return {...order, discount}
}

// Differences in the calculation (percentages, “discount of the day,” etc.)
// are collected as a separate set of functions:

const discountOptions = {
  percent: (order, percent)  => order.total * percent / 100
  daily: daysDiscount()
}

// As a result, we get a generalized action for applying a discount
// and a dictionary with discounts of different kinds.
// Then, the application of any discount will now become uniform:

const a = applyDiscount(order, discountOptions.daily)
const b = applyDiscount(order, discountOptions.percent(order, 40))
```

> **More info** 🔬
>
> We will discuss generalized algorithms, their use, and parameterization in a separate chapter.

---

1. “Jedi Techniques” by Maxim Dorofeev, Translated summary, https://bespoyasov.me/blog/jedi-techniques/

8

Abstraction

Overly detailed code is "noisy" and difficult to fit in your head. It is overloaded with information and does so much that it's hard to understand what it does.

Generally, noisy code reflects a complex action with many steps. The more complex the action, the more details we need to consider, and the more steps have to describe. Together, these details overwhelm the reader and make the code complicated.

In this chapter, we'll discuss how to use abstraction to reduce noise in code and make it less complicated.

Intent and Implementation

> Abstraction is the elimination of the irrelevant and the amplification of the essential"

In an example in the "Names" chapter, we've divided details of the function name into two groups: "important outside" and "important inside." This way, we've simplified the name and balanced the amount of information in the function name and its body. We kept essential details in the name and hid the less important ones in the implementation. This details separation "by levels" is called abstraction.

Abstraction helps tame complexity by separating *intention* and *implementation*. Intention describes *what* we're going to do, and implementation describes *how* we will do it. The intention is essential at the "top" level when describing an entity and its interaction with the environment. The implementation details are important at the level "below", when we focus on the entity's internal processes.

```JavaScript
// Name and signature reflect the intention...
function isChild(user) {
  // ...And function body reflects the implementation.
  return user.age < 18;
}

// When we use the function with others,
// we care about its purpose and goals,
// not about its implementation details:
if (isChild(user)) toggleParentControl();
```

Our brain can only work with a limited amount of information at a time. Abstraction helps us focus on the details that are important *now*. This focus is especially needed when working with code where details from "different levels" are mixed.

Consider an example. Let's imagine we have a function, `subscribeToFeed`, which checks the validity of the given email at the start:

```JavaScript
function subscribeToFeed(email) {
  if (!email.includes("@") || !email.includes(".")) return false;

  const recipients = addRecipient(email);
  confirmFeedSubscription(recipients);
}
```

If we compare email validation with the other actions (`addRecipient`, `confirmFeedSubscription`), we'll see that it speaks in terms "too primitive" for this task.

While other functions speak in terms of "emails," "feeds," and "subscriptions," the validation tells about `"@"` and `"."` characters. Because of this, we have to "jump" between the validation details and its purpose when we read the function.

The `subscribeToFeed` function wants to know if the address is valid. But it's not crucial how exactly we check the email to this function. These details aren't needed here.

We can extract (abstract) the email check into a separate function `isValidEmail`:

```JavaScript
function isValidEmail(email) {
  return email.includes("@") && email.includes(".");
}

function subscribeToFeed(email) {
  if (!isValidEmail(email)) return false;

  const recipients = addRecipient(email);
  confirmFeedSubscription(recipients);
}
```

The `isValidEmail` function name now reflects the entire "set of validation actions" as a single phrase. The name uses terms close to those used by the names of the functions around it. This closeness helps focus on the actions' goals rather than their internal processes.

> **By the way** 👀
>
> This is a bit harder to achieve in code with side effects, but we'll discuss that in a separate chapter.

When we *call* the `isValidEmail` function, we focus on its name and intent. At this point, we care about how the function interacts with the entities around it, that is, "what happens if the email is invalid."

If we care about validation rules, we study the body of the function—the implementation. At that point, we care about how the function decides whether to return `true` or `false`.

> **By the way** 🤖
>
> In languages with static typing, the function signature can also express the intention. We'll talk more about this in the chapter on static typing.

Abstraction helps us to "dive" into complex concepts and processes gradually. It gives us information about the system in chunks. At each "level of detail," we have only the information we need to understand the system at that level. Mark Seemann calls this fractal architecture, and I find this metaphor very useful.🔗

Fractal Architecture

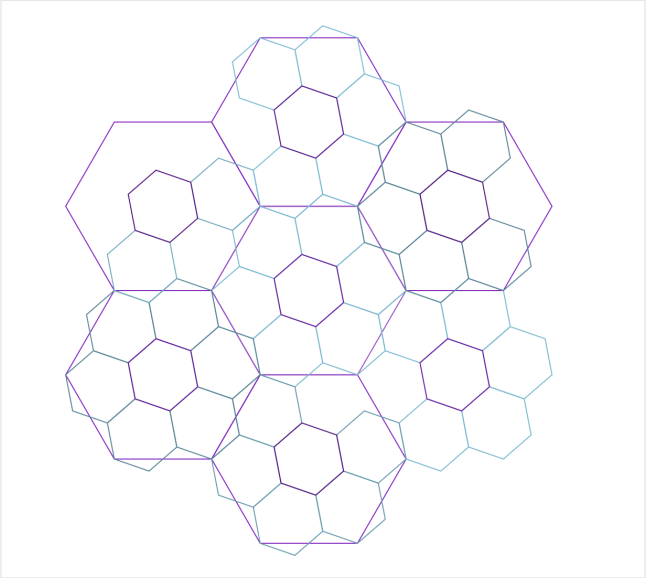
Compared to a computer, our brains are computationally weak. It's difficult for us to multiply large numbers or hold more than ten concepts in our heads simultaneously.

When refactoring, we should keep an eye on how difficult it is to "fit a piece of code in our head." If it's difficult to remember all the details, the code has a problem.

In good code, there's exactly as much information on the screen as the reader needs at any given moment. Mark Seemann suggests writing programs so that the number of constituent parts at each "level of detail" doesn't exceed some limit. We can use this heuristic to check if the code is overly detailed.🔗

> **By the way 🧠**
>
> Mark suggests the number 7 as the limit. He relies on the premise that we can keep 7±2 objects in our heads.[1][2] He also says that the specific number is not crucial. The main goal is to have such a limit in principle.

To visualize "levels," he suggests using a grid of hexagons. Each hexagon is a part of the system, which can be detailed more profoundly. At each level of detail, we'll see no more than N parts important to that level. If we need to know how a particular part works, we can "zoom in" a hexagon and see what it consists of.



So the program breaks down into chunks, which break down into chunks, which break down into chunks...

> **Copyright note ©**
>
> Picture above was generated by the tool from the article about Fractal Hex Flowers.[3]

Let's look at an example to understand how it helps us to refactor code. Consider an app where we show a dashboard for logged-in users and a login page for others.

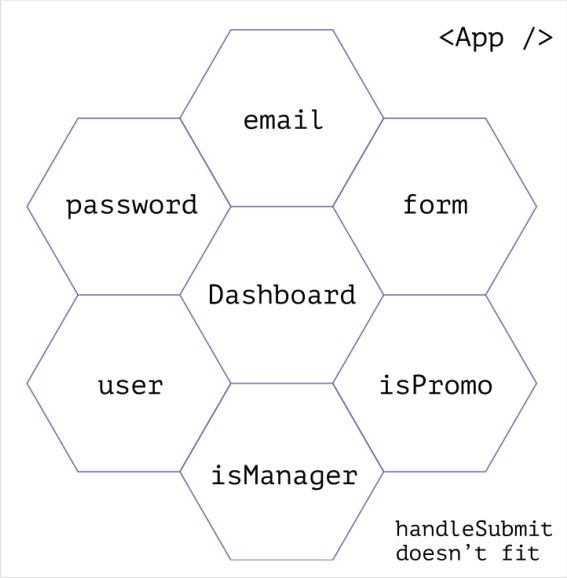
The entry point to the application (the top level of detail) might look something like this:

JavaScript

```
const App = () => {
  const user = currentUser();
  const isManager = hasManagerRole(user);
  const isPromoAccount = checkPromoAccount(location);

  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const handleSubmit = () => {
    /*...*/
  };

  return isManager || isPromoAccount ? (
    <Dashboard />
  ) : (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
        value={email}
        onChange={({ target }) => setEmail(target.value)}
      />
      <input
        type="password"
        password={password}
        onChange={({ target }) => setPassword(target.value)}
      />
      <button>Login</button>
    </form>
  );
};
```

It's quite possible to understand such code, but it will take a comparatively longer time because of the number of details. That's because the amount of information in this code is close to the limits of our working memory.

If we express this code on the hexagon diagram, we'll see that some parts just don't fit:



Objects and functions reflected on hex tiles; one of them doesn't fit

The code would be much easier to explore and understand if, at the top level, we "prepared" the reader and told them *what* the `App` component does. In that case, variable and subcomponent names would express the intent and add up to a "story":

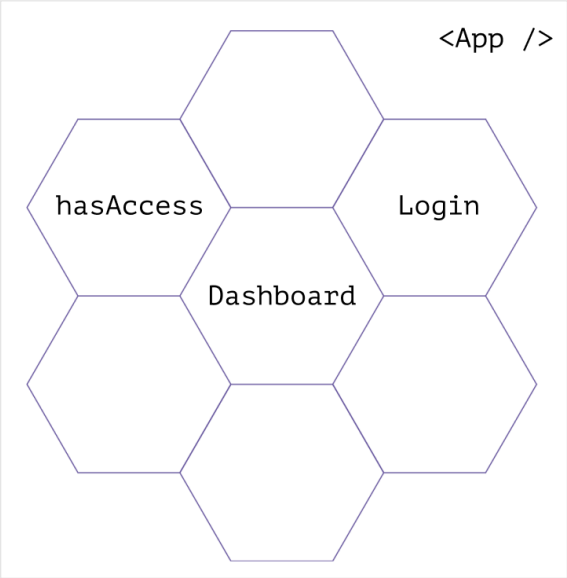
JavaScript

```
const App = () => {
  const hasAccess = useHasAccess();
  return hasAccess ? <Dashboard /> : <Login />;
};

/**
 * If a user has access (`hasAccess`) to the control panel,
 * the application will show them the panel component (`Dashboard`).
 * If not, they will be prompted to log in (`Login`).
 */
```

The implementation of the corresponding functions and components would reflect the details of the "story." For example, we could explain how to determine whether a user has access to the control panel through the implementation of the `useHasAccess` hook:

```JavaScript
function useHasAccess() {
  const user = currentUser();
  const isManager = hasManagerRole(user);
  const isPromoAccount = checkPromoAccount(location);
  return isManager || isPromoAccount;
}

/**
 * We'll check if the current user (`currentUser`)
 * is a manager (`hasManagerRole`).
 * We'll also check if the application is running under a promo account,
 * in which the control panel is available to everyone (`checkPromoAccount`).
 */
```

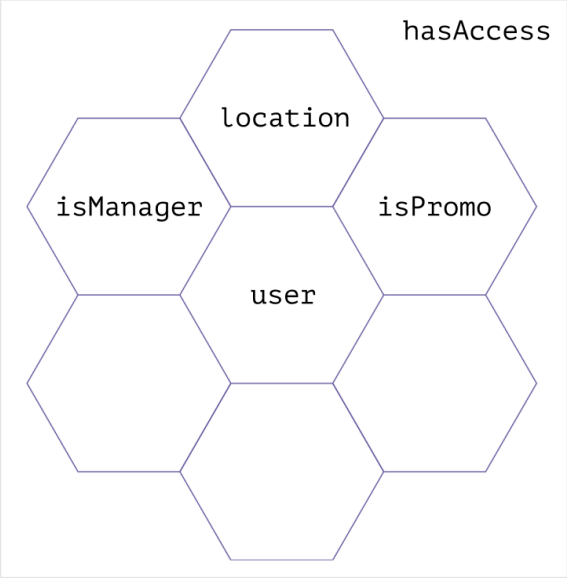
So at the top level of detail, we would see only three parts: `hasAccess`, `Dashboard`, and `Login`. This code is much easier to "load" in our heads and focus on the relationships between its parts.



Top layer of application detail in the form of hex tiles

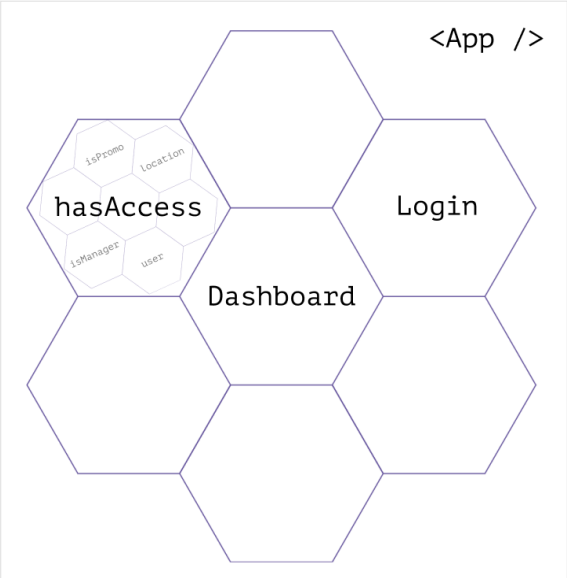
If we needed to detail a part of the "story," we could "zoom in" one of the cells and examine its structure.

For example, in `useHasAccess` we can see how its four parts work together. At this level, it doesn't really matter what happens "level above" because we focus on the structure of the `useHasAccess`.



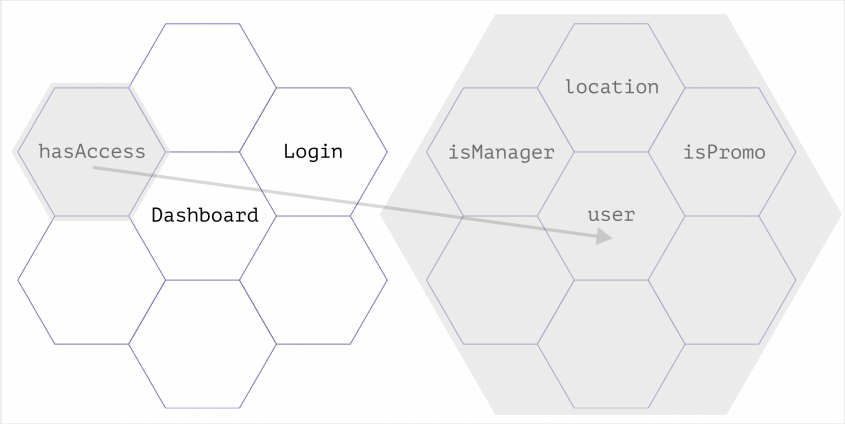
Detailed tiling of the `useHasAccess` hook

It's called fractal architecture because we can nest one level of detail into another:



Levels of detail are nested in on another like a Russian doll

...And switch our attention between levels at any moment:



Switching attention between levels

Each of the cells can be "zoomed in" to its parts. And each of its parts can also be "zoomed in" further. This way we can go deeper and deeper into the system but control how much information we consume.

At each level, there's only a limited *comfortable* amount of information waiting for us. Such code is much easier to "fit in one's head."

Abstraction is at the core of this approach. We can "zoom out" to look at a module's interactions with others or "zoom in" to study the details of how it works.

> **By the way** 📚
>
> Fractal architecture is somewhat reminiscent of the Zoom World concept from "The Humane Interface" by Jeff Raskin.🔗

Separation of Concerns

Abstraction forces us to divide the code into parts, but it's not always obvious how to do this. To make this task easier, we can use the *Separation of Concerns, SoC* principle.🔗 It offers to divide the system into such parts, each of which is responsible for only one task.

"Responsibility" and "task" are somewhat vague terms. So we can understand them as a limited set of data and actions that are *related to each other more strongly than other* data and actions.

> **By the way** 🔗
>
> You might remember the term cohesion from that description.🔗 We'll discuss it a bit more in the chapter on module integration.

Parts of well-divided code do not overlap and do not duplicate each other's functionality. The work of such a system is then based on the composition of its parts. During design and refactoring, such a separation forces us to decompose complex tasks into simpler ones.

Task Decomposition

When we see a large piece of code, we should first consider how many different tasks there are. To do this, we can count how many data sets and actions are in that code.

For example, let's look at the submitting login form function:

```JavaScript
async function submitLoginForm(event) {
  const form = event.target;
  const data = {};

  if (!form.email.value || !form.password.value) return;
  data.email = form.email.value;
  data.password = form.password.value;

  const response = await fetch("/api/login", {
    method: "POST",
    body: JSON.stringify(data),
  });
  return response.json();
}
```

It's pretty compact, only 13 lines, but we can count three tasks in it:

- Form data extraction;
- Data validation;
- Network requests.

We can also count them by changing the conditions of a task and see *which code will change because of it*. For example, if we add a checkbox to the form, the data extraction code will change:

```JavaScript
async function submitLoginForm(event) {
  // ...

  data.email = form.email.value;
  data.password = form.password.value;

  // New field will appear in the object:
  data.rememberMe = form.rememberMe.checked;

  // ...
}
```

And if we change the API scheme, only the network part will change:

```JavaScript
async function submitLoginForm(event) {
  // ...

  // Argument for `fetch` will change:
  const response = await fetch("/api/v2/login", {
    method: "POST",
    body: JSON.stringify(data),
  });

  // ...
}
```

Such a check helps count *reasons for change* in the code and correlate fragments to them. The number of different reasons will tell us how many tasks the code solves.

> **By the way** 🏀
>
> Not always a large piece of code has many tasks. A complex algorithm implementation might be extensive but solve just one task. We can also check this by changing the task conditions and seeing what code changes.

Single Responsibility Principle

Code that changes for different reasons is best kept separate, and code that changes for the same reason is best kept together. It is known as the *Single Responsibility Principle, SRP*.🔗🔗

We can apply this principle to refactor the `submitLoginForm` function from the example above. Let's extract each task into a separate function and see how `submitLoginForm` code changes. Let's start with the data extraction:

JavaScript

```
// Move data extraction to a separate function.
// Now it all is gathered here, and we know exactly
// where to look if we need to know its details.
function extractLoginData(form) {
  const data = {};

  data.email = form.email.value;
  data.password = form.password.value;

  return data;
}

async function submitLoginForm(event) {
  const form = event.target;

  // Inside `submitLoginForm` we now focus
  // only on using the extracted data.
  const data = extractLoginData(form);

  if (!form.email.value || !form.password.value) return;

  const response = await fetch("/api/login", {
    method: "POST",
    body: JSON.stringify(data),
  });
  return response.json();
}
```

Next, let's think about validation. We now see that validating the DOM object as before makes no sense. We need to validate the data, but it doesn't matter where we get it. By separating responsibility this way, we can get rid of unwanted coupling between tasks.

JavaScript

```
// All validation is now gathered in the `isValidLogin` function.
// Inside it, we check the data, not the properties of the DOM object:
function isValidLogin({ email, password }) {
  return !!email && !!password;
}

async function submitLoginForm(event) {
  const form = event.target;
  const data = extractLoginData(form);
  if (!isValidLogin(data)) return;

  const response = await fetch("/api/login", {
    method: "POST",
    body: JSON.stringify(data),
  });
  return response.json();
}
```

The API call can also be a separate function:

```JavaScript
// All the networking now is in the `loginUser` function.
async function loginUser(data) {
  const method = "POST";
  const body = JSON.stringify(data);

  const response = await fetch("/api/login", { method, body });
  return await response.json();
}

async function submitLoginForm(event) {
  const form = event.target;
  const data = extractLoginData(form);
  if (!isValidLogin(data)) return;

  return await loginUser(data);
}
```

The resulting code is easier to modify because the extracted functions limit the "scopes of responsibility" between tasks. Changes within one of the functions are less likely to cause changes in other functions. For example, when updating validation rules, only the `isValidLogin` function code will change:

```JavaScript
function isValidLogin({ email, password }) {
  // Now checking that the email contains `@` character:
  return email.includes("@") && !!password;
}

// Functions `extractLoginData`, `loginUser`, and `submitLoginForm`
// are not changed.
```

This separation makes it easier to test and develop functions in isolation from each other. And the higher the isolation, the lower the chance of making an accidental mistake when updating the code.

Encapsulation

The single responsibility principle helps to think of code parts (functions, modules, objects) as independent parts of an application.

The parts communicate through APIs (protocols, contracts, interfaces) and do not interfere with each other's inner details. We can call this relationship between entities *encapsulation*.

Encapsulation is often described as just concealing data or restricting access to it, but:

> The most important notion [of encapsulation] is that an object should guarantee that it'll never be in an invalid state… The [encapsulated] object knows best what "valid" means, and how to make that guarantee

Poor encapsulation leads to repeated checks in code and errors due to invalid data. It can be detected by leaky abstractions and high coupling between modules. In one of the following chapters, we'll discuss the coupling in more detail, but now let's focus on leaky abstractions.

Let's try to determine what's wrong with the `makePurchase` function in the example below:

```JavaScript
// purchase.js
import { createOrder } from "./order";

async function makePurchase(user, cart, coupon) {
  if (!cart.products.length) throw new Error("Cart is empty!");

  const order = createOrder(user, cart);
  order.discount = coupon === "HAPPY_FRIDAY" ? order.total * 0.2 : 0;

  await sendOrder(order);
}
```

The main problem with this code is that there's no guarantee that the `sendOrder` function will get a *valid* order. The `makePurchase` function *changes* the state of the `order` object created by *another* module. It behaves as if it knows which state for the `order` object is valid and which is not. But:

> **Making sure the data is valid is an internal task of a particular module !**

Simply put, only the module that knows how to apply an order discount can do it correctly. In our case this is `order.js` —the order was created by it, so the discount should be applied by it as well:

JavaScript

```
// order.js
export function createOrder() {
  /*...*/
}

export function applyDiscount(order, coupon) {
  const discount = coupon === "HAPPY_FRIDAY" ? order.total * 0.2 : 0;
  return { ...order, discount };
}

// purchase.js
import { createOrder, applyDiscount } from "./order";

async function makePurchase(user, cart, coupon) {
  if (!cart.products.length) throw new Error("Cart is empty!");

  const order = createOrder(user, cart);
  const discounted = applyDiscount(order, coupon);
  await sendOrder(discounted);
}

// Now making sure the order data is valid
// is an internal task of the `order` module,
// not the responsibility of the code that calls it.
```

By the way 🥶

In some languages, we can disallow changing data after creating using immutable structures. They make it impossible to bring the data to an invalid state from the outside.

In JavaScript, we can make objects immutable with `Object.freeze`,[🔗] but this is often an overhead. Usually, it's enough to *treat* the data as immutable.

The same problems apply to the cart emptiness check. Although the `makePurchase` function does not change the cart data, it still behaves as if it knows which cart state is valid and which is not.

It's better to leave the emptiness check to the module that creates the cart and knows how to keep it valid:

```JavaScript
// cart.js
export function isEmpty(cart) {
  return !cart.products.length;
}

// purchase.js
import { isEmpty } from "./cart";
import { createOrder, applyDiscount } from "./order";

async function makePurchase(user, cart, coupon) {
  if (isEmpty(cart)) throw new Error("Cart is empty!");

  const order = createOrder(user, cart);
  const discounted = applyDiscount(order, coupon);
  await sendOrder(discounted);
}
```

In the updated code, the `makePurchase` function doesn't change the order state directly and doesn't decide whether the cart is valid or not. Instead, it calls the *public API* of other modules.

It doesn't mean that all errors will automatically disappear after this change—after all, other modules may contain errors themselves. But we've separated responsibilities between modules, so we'll know what to fix if there's an error with order data or cart validation.

We've also limited the scope of change in the code. While the public API of modules isn't changed, the updates and fixes of those modules *will be limited* to their boundaries and won't go outside them.

Among other things, such code is easier to cover with tests and check against project requirements.

---

1. "Agile Principles, Patterns, and Practices in C#" by Robert C. Martin, https://www.goodreads.com/quotes/8806618-abstraction-is-the-elimination-of-the-irrelevant-and-the-amplification

2. "Code That Fits in Your Head" by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head

3. Working memory, Capacity, Wikipedia, https://en.wikipedia.org/wiki/Working_memory#Capacity

4. "Thinking, Fast and Slow" by Daniel Kahneman, https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow

5. "Fractal hex flowers" by Mark Seemann, https://observablehq.com/@ploeh/fractal-hex-flowers

6. "The Humane Interface" by Jef Raskin, https://www.goodreads.com/book/show/344726.The_Humane_Interface

7. Separation of Concerns, Wikipedia, https://en.wikipedia.org/wiki/Separation_of_concerns

8. Cohesion in Computer Science, Wikipedia, https://en.wikipedia.org/wiki/Cohesion_(computer_science)

9. Single Responsibility Principle, Principles of OOD, http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

10. The Single Responsibility Principle by Robert C. Martin, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_76/

11. Encapsulation, Wikipedia, https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)

12. `Object.freeze()`, MDN, https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze

9

Functional Pipeline

As we saw in the previous chapter, abstraction helps to divide the program by "levels of detail." We pay attention to the most important details at each level and use the most appropriate terms.

In user applications, one of such levels describes the *business logic*—the domain processes that make the application unique and gain profit. Those, in other words, are the problems that business wants the developers to solve.

> **For example** 👀
>
> For an online store, the business logic would be creating orders and checking out. For transport company—route and traffic load optimization.

Business logic speaks in the language of the domain. It describes workflows as sequences of events and consequences: "When a user inputs a discount coupon, the application checks its validity and reduces the order price."

Such language is "far from code." Because of the high abstraction level, translating business workflows into code can be difficult. Functional pipeline, which we'll discuss in this chapter, helps to describe business workflows in code more precisely and closer to reality.

Data Transformations

Business workflows are data transformations. For example, applying a discount to an order can be expressed as a transition from one data state to another:

```
"Coupon Application":

[Created Order] + [Valid Coupon] -> [Discounted Order]
```

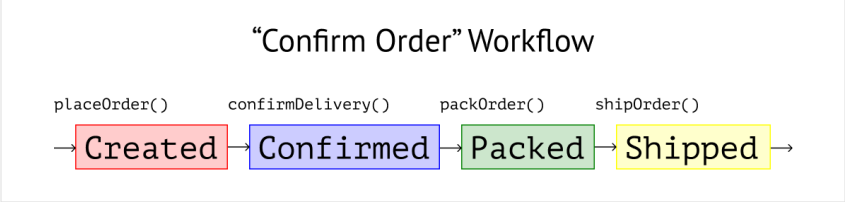
In larger applications, transformations can be more extended, and the data can go through several steps:

```
"Selecting Product Recommendations":

[Product Cart] + [Shopping History] ->
  [Product Categories] + [Recommendation Weights] ->
  [Recommendation List]
```

In poorly organized code, business workflows don't resemble such chains. They are complicated, non-obvious, and often don't speak the language of the domain. As a result, instead of a clear workflow description, we end up with something like:

```
"Selecting Product Recommendations":

[Product Cart] + ... + [Magic 🔮] -> [Recommendation List]
```

In well-organized code, the workflows look linear, and the data in them go through several steps one at a time. The final state of the data is the workflow's desired result.



Data goes through a chain of different states. The output is the desired workflow result

This kind of code organization is called a *functional pipeline*. When refactoring business logic, we can focus on it to make the workflows in the code clearer and more apparent.

Data States

In "Domain Modeling Made Functional," Scott Wlaschin describes how to design a program based on business workflows and their data.[🔗] The suggestion is to represent the workflow steps as separate functions. We can use this idea as a basis for refactoring.

To do this, we first need to highlight all the stages that data passes through. These stages will help us understand how to divide the workflow and what should consider in the code.

Let's analyze this approach with the example of an online store. Suppose the `makeOrder` function compiles an order, applies discount coupons, and adds promo products:

```JavaScript
function makeOrder(user, products, coupon) {
  if (!user || !products.length) throw new InvalidOrderDataError();
  const data = {
    createdAt: Date.now(),
    products,
    total: totalPrice(products),
    discount: selectDiscount(data, coupon),
  };

  if (!selectDiscount(data, coupon)) data.discount = 0;
  if (data.total >= 2000 && isPromoParticipant(user)) {
    data.products.push(FREE_PRODUCT_OF_THE_DAY);
  }

  data.id = generateId();
  data.user = user.id;
  return data;
}
```

The function isn't big, but it does quite a lot:

- It validates the input data;
- Creates an order object;
- Applies a discount from the passed coupon;
- Adds promo products under certain conditions.

At a glance, it's hard to distinguish each of these actions in the function code. There are so many details that it's difficult to see the individual workflow steps.

> **Not only that** 💊
>
> The order object changes within the function chaotically. And although formally, its encapsulation isn't broken—the object is created and changed in the same function—it feels like `makeOrder` is "trying to do someone else's job."

Let's highlight the workflow steps and data states that appear in them:

```
"Show order in UI":

- "Validate Input Data":
  [Raw Unvalidated Input] -> [Validated User] + [Validated Product List]

- "Create Order":
  [User] + [Product List] -> [Created Order]

- "Apply Discount Coupon":
  [Order] + [Coupon] -> [Discounted Order]

- "Apply Promo":
  [Order] + [User] -> [Order with Promo Products]
```

We aim to make the function code look like this list during refactoring. We can start by grouping the code into "sections," each of which will represent a different workflow step:

JavaScript

```
function makeOrder(user, products, coupon) {
  // Validate Input:
  if (!user || !products.length) throw new InvalidOrderDataError();

  // Create Order:
  const data = {
    createdAt: Date.now(),
    products,
    total: totalPrice(products),
  };
  data.id = generateId();
  data.user = user.id;

  // Apply Discount:
  const discount = selectDiscount(data, coupon);
  data.discount = discount ?? 0;

  // Apply Promos:
  if (data.total >= 2000 && isPromoParticipant(user)) {
    data.products.push(FREE_PRODUCT_OF_THE_DAY);
  }

  return data;
}
```

Grouping the steps will help find abstraction problems in the code: if we can think of a meaningful name for a step, we can probably extract its code into a function. In the example above, the comments with the step names fully reflect their intent. Let's extract the steps into separate functions:

```JavaScript
// Create Order:
function createOrder(user, products) {
  return {
    id: generateId(),
    createdAt: Date.now(),
    user: user.id,

    products,
    total: totalPrice(products),
  };
}

// Apply Discount:
function applyCoupon(order, coupon) {
  const discount = selectDiscount(order, coupon) ?? 0;
  return { ...order, discount };
}

// Apply Promos:
function applyPromo(order, user) {
  if (!isPromoParticipant(user) || order.total < 2000) return order;

  const products = [...order.products, FREE_PRODUCT_OF_THE_DAY];
  return { ...order, products };
}
```

The `makeOrder` function then would look like this:

```JavaScript
function makeOrder(user, products, coupon) {
  if (!user || !products.length) throw new InvalidOrderDataError();

  const created = createOrder(user, products);
  const withDiscount = applyCoupon(created, coupon);
  const order = applyPromo(withDiscount, user);

  return order;
}
```

After the changes, the workflow steps are encapsulated in separate functions. These functions only change the order object, keeping it valid. The `makeOrder` function doesn't change data uncontrollably anymore but only calls those functions. It makes invalid orders less likely and the testing of data transformations easier.

The code of `makeOrder` now resembles the list of workflow steps we've started with. The details of each step are hidden behind the name of the corresponding function. The name describes the entire step, which makes the code easier to read.

Also, when adding a new step to the workflow, we now only have to insert a new function in the right place. The other conversions remain unchanged:

```JavaScript
function makeOrder(user, products, coupon, shipDate) {
  if (!user || !products.length) throw new InvalidOrderDataError();

  const created = createOrder(user, products);
  const withDiscount = applyCoupon(created, coupon);
  const withPromo = applyPromo(withDiscount, user);
  const order = addShipment(withPromo, shipDate); // New workflow step.

  return order;
}
```

And when we remove a step, it's easier to find the function to delete—deleting a function call guarantees removing all the code related to that step in the process.

> **By the way** 🕵️
>
> It's not always easy to spot a pipeline. Business logic can be "scattered" across the codebase. In such cases, it can be helpful to draw a diagram of how the application parts communicate with each other to discover patterns.

Unrepresentable Invalid States

Some business workflows only need data in certain states to work. For example, we don't want to ship an order until it's paid or if it misses the delivery address. Such orders are invalid for this workflow.

We can make the code more reliable if we "forbid" the passing of invalid data. To do that, we can design the code in such a way that makes it harder or impossible to pass invalid data.

For example, we can do this using types in languages with static typing. We can describe each data state as a separate type and specify what types are valid for a workflow. *This adds domain constraints directly to the signature* of functions and methods.

> **However** 👀
>
> Clearly, we have to remember about constraints of a particular language. For example, in TypeScript, it's harder to achieve "validation in signature," and it still will be limited by the JS-runtime. But even without the "true validation," this technique helps to reflect more domain knowledge directly into the code.

For example, let's look at the `CustomerEmail` type, which describes the email address of the store's user:

```TypeScript
type CustomerEmail = {
  value: EmailAddress;
  verified: boolean;
};
```

The type has a `verified` flag that shows if the email is verified. The problem with the flag is that it doesn't explain under *what conditions* it'll be `true`. There's not enough knowledge in type about email verification.

> **The code has to compensate for this lack of knowledge somehow. Most often, with extra data checks at runtime !**

For example, imagine a link in the store's UI to restore the user account. When clicked, it should send the user to the "Reset Password" page, but only if their email is verified:

```TypeScript
function restoreAccount(email: CustomerEmail): void {
  if (email.verified) {
    // Send the user to the "Reset Password" page.
  } else {
    return;
  }
}
```

With the current `CustomerEmail` implementation, the `restoreAccount` function accepts invalid data half the time.

It can be fine if the type contains only one such flag. But the more flags there are, the more different states the type has simultaneously and the more probable the errors due to the inconsistent data.

We can solve this by separating different data states into different types:

```TypeScript
// For unverified emails, we use one type:
type UnverifiedEmail = {
  /*...*/
};

// ...And for verified emails, we use another:
type VerifiedEmail = {
  /*...*/
};

// The "any email" can be used
// when the verification isn't important:
type CustomerEmail = UnverifiedEmail | VerifiedEmail;
```

Then for different workflows, we can require different data types:

```TypeScript
// If the function cares about the email verification,
// it can require a specific type in its signature:

function restorePassword(email: VerifiedEmail): void {}
function verifyEmail(email: UnverifiedEmail): void {}

// If the function can handle any email, it can use the common type.
// This way, we can see the requirements for email verification
// right in the function signature:

function isValidEmail(email: CustomerEmail): boolean {}
```

Now, the function signatures are more accurate in describing the domain because they convey more knowledge about it. The functions `restorePassword` and `verifyEmail` warn about their requirements and constraints. The function `isValidEmail` says that it's ready to handle any email, and verification isn't essential.

> **However** 🚧
>
> In the case of TypeScript, type aliases may not be enough. We may want to ensure that we can't create a non-verified email with the `VerifiedEmail` type.
>
> For this, we can use type branding🔗 or agree to create entities only with special classes or factories.
>
> However, for descriptive purposes—transferring knowledge about the domain—aliases may suffice just fine.

Data Validation

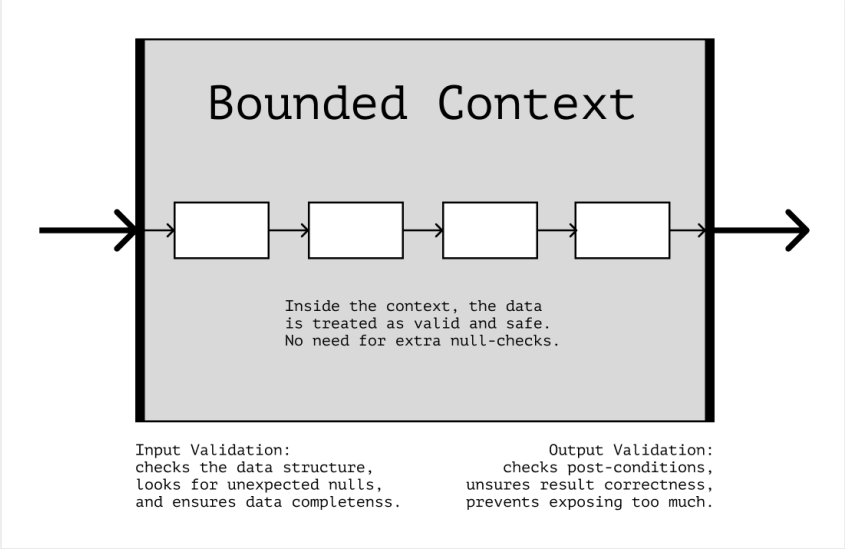
The functional pipeline relies on linear code execution. Steps within a workflow execute one after the other and pass data down through the chain of transformations.

For this idea to work, data within the workflow must be safe and not break the pipeline. However, we can't guarantee that any "external" data is safe. So in the code, we want to separate zones where data can be trusted and where it can't.

> **Islands of security** 🏝
>
> Business workflows ideally should become "islands" where the data is verified and secure.

DDD has an analog for such islands—*bounded contexts*.🔗🔗🔗 Simply put, a bounded context is a set of functions that refer to some part of an application.

According to DDD, validating data is more convenient *at the boundaries* of contexts, for example, at the context input. In this case, "inside" the context, we don't need additional checks because the data is already validated and safe.



All validation occurs at the boundaries; the data inside the context is considered valid and safe

We can use this rule in our code to get rid of unnecessary data checks at runtime. By validating the data at the beginning of the workflow, we can assume later that it meets our requirements.

Then, for example, in the `CartProducts` component, instead of ad-hoc checks for the existence of products and their properties inside the render function:

```javascript
function CartProducts({ items }) {
  return (
    !!items && (
      <ul>
        {items.map((item) =>
          item ? <li key={item.id}>{item.name ?? "—"}</li> : null
        )}
      </ul>
    )
  );
}
```

...We would check the data once at the beginning of the workflow:

```JavaScript
function validateCart(cart) {
  if (!exists(cart)) return [];
  if (hasInvalidItem(cart)) return [];

  return cart;
}

// ...

const validCart = validateCart(serverCart);
```

...And later would use it without any additional checks:

```JavaScript
function CartProducts({ items }) {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}
```

Missing States

Often input validation helps us discover data states we did not notice earlier. For example, the code of the `CartProducts` component in the previous snippet has become simpler, and the flaws in it have become easier to spot:

```JavaScript
// If we render a valid but empty cart,
// the component will render an empty list:

const validEmptyCart = [];
<CartProducts items={validEmptyCart} />;

// Results in <ul></ul>
```

The “Empty cart” state is valid but represents an edge case. Together with input validation, the functional pipeline makes such cases more noticeable because they fall out of “regular” code execution. And the more prominent the edge cases are, the sooner we can detect and handle them:

JavaScript

```
// To fix the problem with the empty list, we can split
// the “Empty cart” and “Cart with products” states
// into different components:

const EmptyCart = () => <p>The cart is empty</p>;
const CartProducts = ({ items }) => {};

// Then, when rendering, we can first handle all the edge cases,
// and then proceed to Happy Path:

function Cart({ serverCart }) {
  const cart = validateCart(serverCart);

  if (isEmpty(cart)) return <EmptyCart />;
  return <CartProducts items={cart} />;
}
```

> **However** 💡
>
> We remember that refactoring must not change the code functionality, so it's better to fix bugs separately. In one of the last chapters, we will discuss how to address problems found in the code but not mix refactoring with bug fixes.

Such edge case handling, as in the `Cart` component, helps us detect more potential edge cases in the earlier stages of development. Considering these edge cases makes the program more reliable and accurate in describing the business workflows.

> **By the way** 👀
>
> Handling edge cases before working with Happy Path might remind you of the technique called “Early return.” We will discuss it more closely in the chapter on conditions and code complexity.

DTO and Deserialization

Validation at the beginning is also useful if the data can get corrupted by serialization or deserialization.🔗

Typically, information between parts of a system is transferred as a *Data Transfer Object, DTO.*🔗🔗 These are “packages of information” that travel from one part of an application to another—from server to client, for example.

The structure and format of DTOs are intentionally simple: only strings, numbers, booleans, arrays, and objects. For example, JSON, which is often used to communicate between the server and the client, has no complex types or structures.

During “translation” between complex domain types and intentionally simple DTOs, something can go wrong, and the data can become invalid. This data can break the workflow if not checked before use.

> **In detail** 📚
>
> You can read more about the functional pipeline, business workflows, bonded contexts, data validation, and DDD in “Domain Modeling Made Functional” by Scott Wlaschin.🔗 Great book, highly recommended.

Data Mapping and Selectors

We might need the same data for different purposes. For example, the UI may render a shopping cart differently depending on the user’s settings.

The functional pipeline suggests “preparing” data for such situations in advance. For example, we might want to select the required fragments from the original data in advance, transform some data sets into others or even merge several data sets into one.

> **Beware the simplification** 🚧
>
> From the description above, you might remember some terms: mapping, projection, slice, lens, and mapping.🔗🔗🔗🔗
>
> I decided not to use them in this book to avoid introducing too many new concepts. Instead, I will use the word “selector” in the text as a general synonym for all these terms.

Data selectors help to decouple modules that use similar but slightly different data. For example, let’s look at the `CartProducts` component, which renders a shopping cart:

> ```
> function CartProducts({ serverCart }) {
>   return (
>     <ul>
>       {serverCart.map((item) => (
>         <li key={item.id}>
>           {item.product.name}: {item.product.price} × {item.count}
>         </li>
>       ))}
>     </ul>
>   );
> }
> ```
> JavaScript

Right now, it relies on the cart data structure, which comes from the server. If the structure changes, we’ll have to change the component too:

```JavaScript
// If products start to arrive separately, we'll have to search
// for a particular product during the rendering.

function CartProducts({ serverCart, serverProducts }) {
  return (
    <ul>
      {serverCart.map((item) => {
        const product = serverProducts.find(
          (product) => item.productId === product.id
        ).name;

        return (
          <li key={item.id}>
            {product.name}:{product.price} × {item.count}
          </li>
        );
      })}
    </ul>
  );
}
```

Data selectors can help us decouple the server response and the data structure we use for rendering. For example, we can represent such a selector as a function:

```JavaScript
// The `toClientCart` function “converts” the data into a structure,
// which the application components will use.

function toClientCart(cart, products) {
  return cart.map(({ productId, ...item }) => {
    const product = products.find(({ id }) => productId === id);
    return { ...item, product };
  });
}
```

Then, we will convert the data using this function before rendering the component:

```JavaScript
const serverCart = await fetchCart(userId)
const cart = toClientCart(serverCart, serverProducts)

// ...

<CartProducts items={cart} />
```

The component then will rely on the data structure that is defined *by us*, not a third-party:

JavaScript

```
function CartProducts({ items }) {
  return (
    <ul>
      {items.map(({ id, count, product }) => (
        <li key={id}>
          {product.name} {product.price} × {count}
        </li>
      ))}
    </ul>
  );
}
```

So if the API response changes, we won't need to update all the components that use that data. We would only need to update the selector function.

> **By the way** 🔌
>
> This is especially useful if the server often breaks backward compatibility with client code. This technique can be considered a special case of the "Adapter" pattern.

It's also convenient to use selectors if the UI has different representations for the same data. For example, in addition to the cart, the store can have a list of all available products with some of them marked as "In Cart".

To render such a list, we can reuse already existing data but "prepare" them in a slightly different way:

JavaScript

```
// We use the existing server data,
// but create a different structure:

function toClientShowcase(products, cart) {
  return products.map((product) => ({
    ...product,
    inCart: cart.some(({ productId }) => productId === product.id),
  }));
}
```

The `Showcase` component doesn't need to know anything about the server response. It only works with the selector's result:

```JavaScript
function Showcase({ items }) {
  return (
    <ul>
      {items.map(({ product, inCart }) => {
        <li key={product.id}>
          {product.name} <input type="checkbox" checked={inCart} disabled />
        </li>;
      })}
    </ul>
  );
}
```

This approach helps to separate the responsibility between the code: components only care about the render, and selectors "convert" the data.

Components become less coupled to the rest of the code because they rely on structures we fully control. It makes it easier, for example, to replace one component with another if we need to update the UI.

Testing data transformations becomes easier because any selector is a regular function. We don't need a complex infrastructure to test it. Testing transformations within a component, for example, would require its rendering.

We have more control over the data we want (or don't want) to include in the selector's result. We can keep only the fields that a particular component uses and filter out everything else.

It's usually best to apply selection right after the data is validated. At that point, we *already know* that the data is safe, but *don't rely* on its structure anywhere in the code yet. This gives us the ability to convert the data for a particular task.

> **Note** 🔗
>
> The data in this example goes through the chain of states: "Raw Server Data" → "Valid Data" → "Prepared for a Task" → "Displayed in the UI."
>
> Functional pipeline helps to describe *any task* as a similar chain. It makes the decomposition easier because chains help build a clear mental model of the workflow we express in code.

---

1. "Domain Modeling Made Functional" by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional

2. "Branding and Type-Tagging" by Kevin B. Greene, https://medium.com/@KevinBGreene/surviving-the-typescript-ecosystem-branding-and-type-tagging-6cf6e516523d

3. "Bounded Context in DDD" by Martin Fowler, https://www.martinfowler.com/bliki/BoundedContext.html

4. "Domain-Driven Design" by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design

5. Serialization, Wikipedia, https://en.wikipedia.org/wiki/Serialization

6. Data Transfer Object, DTO, Wikipedia, https://en.wikipedia.org/wiki/Data_transfer_object

7. “Data Mapper” by Martin Fowler, https://martinfowler.com/eaaCatalog/dataMapper.html

8. Projection operations, C# Guide, https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/projection-operations

9. API Slice Overview, Redux Toolkit Docs, https://redux-toolkit.js.org/rtk-query/api/created-api/overview#api-slice-overview

10. “Lenses in Functional Programming” by Albert Steckermeier, https://sinusoid.es/misc/lager/lenses.pdf

11. Adapter Pattern, Refactoring Guru, https://refactoring.guru/design-patterns/adapter

10

Conditions and Complexity

The previous chapter discussed the benefits of the functional pipeline and linear code execution. In real-world applications, however, it isn't always obvious how to make code linear.

Big applications often have complex logic rules with multiple conditions. Conditions make the helpful app: they describe its behavior in different situations. But they also make the code more complicated.

This chapter will discuss unraveling complex conditions and what patterns can help.

Cyclomatic and Cognitive Complexity

To understand how exactly conditions make code more complex, let's first try to define "complexity".

One heuristic for finding complex code is paying attention to its nesting. The higher the nesting, the more complex the code. We can understand *what* awaits us just by looking at a snippet with deeply nested code:

```
function doSomething() {
__while (...) {
____if (...) {
______for (...) {
________if (...) {
__________break;
________}
________for (...) {
__________if (...) {
____________continue
__________}
________}
______}
____}
__}
}
```

The empty space on the left side of the text shows how much information the snippet contains. The more space there is, the more details we have to keep in mind, and the more complicated the code.

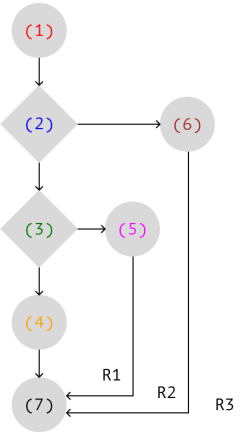
> **By the way**
>
> In the book "Your Code as a Crime Scene" Adam Tornhill calls such a space "negative."

When reading code, we build a model of how it works in our head. This model describes what happens to the data and how the instructions are executed.

Each condition and loop adds to the mental model a new "path." These paths show how to go from the beginning of the code to its end. The more different "paths" there are, the harder it is to keep track of them all simultaneously. The number of "paths" we can take from the beginning of a function to its end is called the *cyclomatic complexity* of the function.

We can visualize the number of "paths" in the function as a graph. Each new condition or loop adds a new "branch," which makes the graph and the function more complex.

```
function selectDiscount(order, coupon) {
  let discount;(1)

  if (order.total >= 1000) {(2)
    if (coupon === HAPPY_FRIDAY) {(3)
      discount = order.total * 0.2;(4)
    }
    else {
      discount = 200;(5)
    }
  } else {
    discount = 0;(6)
  }

  return discount;(7)
}
```



Graph of function with complexity 3. We can calculate the complexity as the difference between the graph nodes and edges or the number of regions on it

The more "branches" the function has, the harder it is to work with.

> **By the way** 🧠
>
> In addition to cyclomatic complexity, there is also *cognitive complexity*.
>
> Cyclomatic complexity counts the number of "different paths" in a function. Cognitive complexity counts the function's "breaks in the linear flow."
>
> Because of this, it's often said that the cyclomatic complexity shows how difficult a function is to test (how many branches to cover in tests), and cognitive complexity shows how difficult it is to understand.
>
> We won't focus on the differences between the two because they won't be significant for this book. Later in the text, we'll use the term "complexity" as a synonym for both properties.

An important feature of such properties is that they're *measurable*. We can choose *limits* for measurable properties and automate their checks. For example, we can set up the linter to tell us when the code complexity exceeds the selected limit:

```
function addSymbol(state: KeyboardState, value: string):
KeyboardState

Function 'addSymbol' has a complexity of 12. Maximum allowed is
10. eslint(complexity)

View Problem   Quick Fix... (⌘.)
function addSymbol(state: KeyboardState, value: string): KeyboardState
```

Linter errors when the complexity exceeds the limit

The specific number depends on the property, the project, the language, and the team. For cyclomatic complexity, Mark Seemann in "Code That Fits in Your Head" suggests using the number 7.🔗 Wikipedia offers number 10.🔗 I usually use 10 in my projects because it's a "round" number, but all this is highly arbitrary.

Flat Over Nested

Negative space and large nesting are a consequence of high code complexity. To make the code simpler, when refactoring, we can stick to this heuristic:

> **Make the conditions flat. ...And then simplify the code even more ❗**

Flat conditions "straighten out" the execution flow. They decrease the number of details that we need to keep track of at the same time. They help us see patterns in the code and simplify it even more. The flatter the code, the easier it is to understand the meaning of a function or a module.

> **By the way 🐍**
>
> It resembles a principle from Zen of Python: "Flat is better than nested."🔗 In my opinion, this only confirms that we're on the right track.

Early Return

The easiest and most common technique we can use to "flatten" code is the early return. It helps to filter out condition branches one by one and no longer keep them in our heads.

As an example, let's look at the `usersController` function. It creates a new user based on its arguments. The conditions in the function force us to keep *all of its branches* in mind until the very end because they all contain some important behavior.

JavaScript

```
function usersController(req, res) {
  if (req.body) {
    const { name, email } = req.body;
    if (name && email) {
      const user = createUser({ name, email });
      if (user) {
        res.json({ user });
      } else {
        res.status(500);
      }
    } else {
      res.status(400).json({ error: "Name and email required" });
    }
  } else {
    res.status(400).json({ error: "Invalid request payload." });
  }
}
```

Notice, however, that the code inside `else` blocks handles "edge cases"—errors and invalid data. We can simplify the function by "inverting" the condition and handling edge cases first, leaving only the "happy path" at the end:

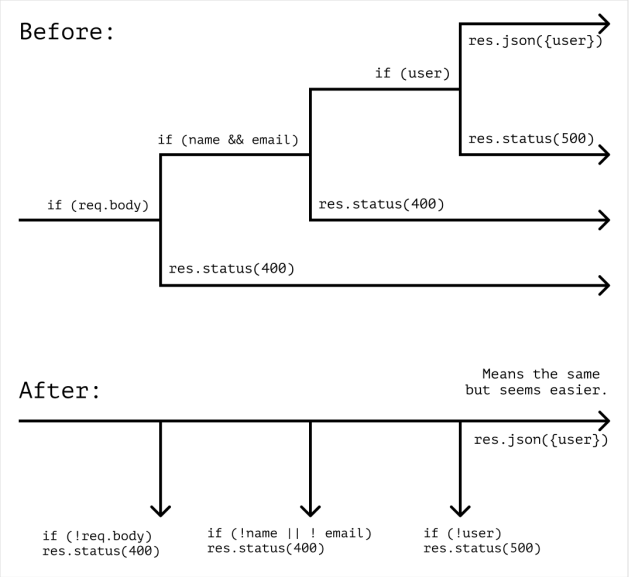
JavaScript

```
function usersController(req, res) {
  if (!req.body) {
    return res.status(400).json({ error: "Invalid request payload." });
  }

  const { name, email } = req.body;
  if (!name || !email) {
    return res.status(400).json({ error: "Name and email required" });
  }

  const user = createUser({ name, email });
  if (!user) return res.status(500);

  res.json({ user });
}
```

The total amount of information and cases the function handles hasn't changed. But we've reduced the number of branches that we need to *remember at the same time*.

We check the edge cases one by one and *terminate* the function if we come across one of them. If the function continues to work, it means no edge cases have happened.

Filtering edge cases one by one allows us to forget the checked ones. They no longer take up our attention and working memory.



The condition becomes easier to understand because we don't need to remember so many details

> **However** 🛡️
>
> Early return may not be suitable for "defensive" programming: when we explicitly handle each branch of a condition.
>
> In my experience, there haven't been many projects that have required defensive programming. So I tend to use early return by default and switch to defensive programming in exceptional cases.

Component Rendering

Early return can be useful to simplify the rendering of React components. For example, the code of the `Cover` component below is relatively difficult to read because of the conditions:

> JavaScript
>
> ```
> function Cover({ error, isLoading, data }) {
>   if (!error && !isLoading) {
>     const image = data.image ?? DEFAULT_COVER_IMAGE;
>     return <Picture src={image} />;
>   } else if (isLoading) {
>     return <Loader />;
>   } else {
>     return <Error message={error} />;
>   }
> }
> ```

We can simplify it by "inverting" the condition and handling the "loading" and "error" states first:

```JavaScript
function Cover({ error, isLoading, data }) {
  if (isLoading) return <Loader />;
  if (error) return <Error message={error} />;

  const image = data.image ?? DEFAULT_COVER_IMAGE;
  return <Picture src={image} />;
}
```

We remember that the *number of branches* in the condition remains the same after applying the early return. This means that the complexity of the code may still be high. We should look at the metrics and see if the code has become simpler.

If the complexity of the function is still high, we should check if it has issues with abstraction or separation of concerns. We'll probably notice that the function can be split into several:

```JavaScript
// In this case, we can separate the "rendering of the picture"
// and the "decision what component to render."
//
// We'll extract the picture rendering in `CoverPicture` component
// and use it when the data is loaded without errors.

function CoverPicture(image) {
  const source = image ?? DEFAULT_COVER_IMAGE;
  return <Picture src={source} />;
}

// The `Cover` component will become an "entry point"
// and will decide what to show:
// `Loader`, `Error`, or `CoverPicture`.

function Cover({ error, isLoading, data }) {
  if (isLoading) return <Loader />;
  if (error) return <Error message={error} />;
  return <CoverPicture image={data.image} />;
}
```

After such refactoring, we should consider the names of the changed functions and components. The old names may become inaccurate after we improve the separation of their responsibilities. In the example above, we should think if the names `Cover` and `CoverPicture` adequately reflect the meaning of these components.

> **By the way** 🤖
>
> In general, finite state machines are better for working with UI than the early return.[🔗] But if we can't implement them in our project, the early return can significantly simplify the code.

Conceptually, early return inside a render function is similar to validating data at the beginning of a business workflow. We just filter out "wrong" UI states instead of invalid data.

Variables, Predicates, and Boolean Algebra

We can't "invert" every condition. If the condition branches are tightly intertwined, we may not find where to start untangling them. To simplify such conditions, we need to find patterns in them.

The first thing we should do is extract repetitive condition parts into variables. Variable names will abstract the details and help focus on the condition's meaning. This will allow us to see if we can simplify the condition with boolean logic.

De Morgan's Laws

De Morgan's laws are a set of transformation rules that link logical operations through negation.[🔗] They can help us "unfold" parentheses in conditions:

```
!(A && B) === !A || !B;
!(A || B) === !A && !B;
```

We can use De Morgan's laws to make the condition less noisy. For example, the condition below has too much detail, and it's hard to see its meaning "through the characters":

JavaScript

```
if (user.score >= 50) {
  if (user.armor < 50 || user.resistance !== 1) {
  }
} else if (user.score < 50) {
  if (user.resistance === 1 && user.armor >= 50) {
  }
}
```

As a first step, we can extract repetitive expressions into variables:

JavaScript

```
const hasHighScore = user.score >= 50;
const hasHeavyArmor = user.armor >= 50;
const hasResistance = user.resistance === 1;
```

...Then the condition will become a combination of these variables:

```JavaScript
if (hasHighScore) {
  if (!hasHeavyArmor || !hasResistance) {
  }
} else if (!hasHighScore) {
  if (hasResistance && hasHeavyArmor) {
  }
}
```

In such a condition, it's much easier to notice patterns within `if` blocks and simplify them. In particular, we can apply De Morgan's first law to simplify the combinations of the variables `hasHeavyArmor` and `hasResistance`:

```JavaScript
// Extract 2nd nested condition into a variable:
const hasAdvantage = hasHeavyArmor && hasResistance;

// Notice that the 1st nested condition
// can transform into `!hasAdvantage`
// according to the first De Morgan's law.
//
// !A || !B === !(A && B)
//
// A -> hasHeavyArmor
// B -> hasResistance
//
// !hasHeavyArmor || !hasResistance
//    === !(hasHeavyArmor && hasResistance)
//    === !hasAdvantage

// Then the whole condition becomes:
if (hasHighScore && !hasAdvantage) {
} else if (!hasHighScore && hasAdvantage) {
}
```

Predicates

Not all conditions can be extracted into a variable. For example, extracting a condition is more difficult if it depends on other variables or dynamic data.

For such cases, we can use *predicates*—functions that take an arbitrary number of arguments and return `true` or `false`. The function `isAdult` in the snippet below is an example of a predicate:

```JavaScript
const isAdult = (user) => user.age >= 21;

isAdult({ age: 25 }); // true
isAdult({ age: 15 }); // false
```

We can store the condition “blueprint” in predicates. That is, describe the principle of comparison between different values in the function:

```JavaScript
// The condition in the example below compares variables
// with the same reference value—21:
if (user1.age >= 21) {
} else if (user2.age < 21) {
}

// We can put the “blueprint” of this comparison
// in the `isAdult` predicate and use it instead:
if (isAdult(user1)) {
} else if (!isAdult(user2)) {
}
```

Predicate names reflect the meaning of the performed check as a whole. They help to fix “leaky” abstractions and make the code less noisy. For example, in the snippet below, it’s hard to tell the meaning of the condition check at first glance:

```JavaScript
if (user.account < totalPrice(cart.products) - cart.discount) {
  throw new Error("Not enough money.");
}
```

If we extract the comparison in the `hasEnoughMoney` predicate, it’ll be easier to understand its meaning from the function name:

```JavaScript
function hasEnoughMoney(user, cart) {
  return user.account >= totalPrice(cart.products) - cart.discount;
}

if (!hasEnoughMoney(user, cart)) {
  throw new Error("Not enough money.");
}
```

In addition, predicates, like variables, help to notice patterns in complex conditions and simplify them with boolean logic.

Primitive Pattern Matching

When refactoring, we should also consider multiple `else-if` statements. If such conditions select a value, more declarative constructions can replace them.

For example, let’s look at the `showErrorMessage` function, which maps a validation error to a message for it:

```TypeScript
type ValidationMessage = string;
type ValidationError = "MissingEmail" | "MissingPassword" | "TooShortPassword";

function showErrorMessage(errorType: ValidationError): ValidationMessage {
  let message = "";

  if (errorType === "MissingEmail") {
    message = "Email is required.";
  } else if (errorType === "MissingPassword") {
    message = "Password is required.";
  }

  return message;
}
```

The function is supposed to check all possible variants of `ValidationError` and choose a message, but it misses a message for `TooShortPassword`. Without special checks (e.g., exhaustiveness check), the TypeScript compiler can't help us here. So it's relatively easy to miss something and not even notice it.

We can replace multiple conditions with more declarative constructions. For example, we can collect all error messages into a dictionary with error types as keys and messages as values. Selecting errors from such a dictionary may be more reliable because the compiler can spot missing keys:

```TypeScript
function showErrorMessage(errorType: ValidationError): ValidationMessage {
  // In the `messages` variable type, we explicitly specify,
  // that all possible errors should be listed in it:
  const messages: Record<ValidationError, ValidationMessage> = {
    MissingEmail: "Email is required.",
    MissingPassword: "Password is required.",

    // If any key is missing, the compiler will tell us about it:
    // “Property 'TooShortPassword' is missing in type...”
  };

  return messages[errorType];
}
```

Conceptually, this is similar to pattern matching. We match `errorType` to the keys of the `messages` object and select the appropriate value by it.

This selection is somewhat similar to how `switch` works, only, in this case, type checking doesn't require additional actions (such as an exhaustiveness check). In TypeScript, this is probably the cheapest way to simulate type-safe "pattern matching" without third-party libraries.

> **By the way** 🔍
>
> It's worth noting that this technique is more suitable for simple cases and doesn't look as cool as real pattern matching in functional languages.
>
> Native syntax for pattern matching in JS is still in Stage 1 at the time of writing. We can use third-party libraries like `ts-pattern` for more complex cases.

In JavaScript code, there isn't type checking, of course. But the declarative style of this approach also makes the manual search for errors easier. The more explicitly we match keys and values, the easier it is to spot mismatches.

> **Read more** 👓
>
> We'll talk about declarative style and its benefits in detail in one of the following chapters.

Strategy

Multiple `else-if` statements can choose not just a value but the program behavior. The choice of behavior among similar variants can indicate insufficient separation of concerns or poor polymorphism. One possible solution to this problem is identical to the refactoring techniques from the previous section.

Let's look at the `notifyUser` function, for example. It sends a message to a user in one of three possible ways. The choice of the particular notification option depends on the conditions within this function:

> TypeScript
>
```
function notifyUser(message) {
  if (method === methods.popup) {
    const popup = document.querySelector(".popup");
    popup.innerText = message;
    popup.style.display = "block";
  } else if (method === methods.remote) {
    notificationService.setup(TOKEN);
    notificationService.sendMessage(message);
  } else if (method === methods.hidden) {
    console.log(message);
  }
}
```

The problem with the function is that it mixes different tasks. It chooses how to send a message *and* sends the message. Adding a new notification option or changing an existing one will affect the whole function. Maybe even the parts that *aren't related to that change*.

Because of this, we can't use one of the notification options separately in the rest of the code. We can't test different options in isolation from each other, either. Also, if we add new options, the `notifyUser` function will become much bigger and more complex.

Instead, we can separate the *choice* of the behavior from the *behavior* itself:

TypeScript

```
// Let's extract each notification option into a separate function:
function showPopupMessage(message) {
  const popup = document.querySelector(".popup");
  popup.innerText = message;
  popup.style.display = "block";
}

// If the function signature is incompatible with the others:
function notifyUser(token, message) {
  notificationService.setup(token);
  notificationService.sendMessage(message);
}

// ...We can adapt it:
const showNotificationMessage = (message) => notifyUser(TOKEN, message);

// Next, we prepare a set of such functions,
// one for each value from `methods`:
const notifiers = {
  [methods.popup]: showPopupMessage,
  [methods.hidden]: console.log,
  [methods.remote]: showNotificationMessage,
};

// When using it, we choose the behavior
// depending on the `method` parameter:
function notifyUser(message) {
  const notifier = notifiers[method] ?? defaultNotifier;
  notifier(message);
}

// It's also possible to select the function beforehand,
// if `method` is known before `notifyUser` is executed.
```

This implementation makes it easier to add and delete notification options:

```TypeScript
// To add a new option, we add a new function...
function showAlertMessage(message) {
  window.alert(message);
}

const notifiers = {
  // ...
  // ...And “register” it in the object:
  [methods.browser]: showAlertMessage,
};

// Other code is unchanged.
```

Changes to an individual function won't go beyond it and won't affect `notifyUser` or other functions. Testing and using such functions independently is much easier.

The separation of choice and behavior is essentially the “Strategy” pattern.[🔗] It can be hard to recognize it in code without classes because examples of this pattern are most often shown in the OOP paradigm, but it is. The only difference is that while in OOP, every behavior strategy is a class, in our example, it's a function.

Null-Object

“The same but slightly different” functionality can also cause unnecessary checks and conditions.

For example, let's say we create a mobile app for iOS, Android, and the web. In the mobile version, we want to add lock-screen widgets. Both iOS and Android have native APIs for making such widgets. Let's pretend we have JS adapters for such APIs.

The `Device` interface describes the adapters' functionality. Adapters contain a platform identifier and a method to update the widget:

```TypeScript
interface Device {
  platform: Platform;
  updateWidget(data: WidgetData);
}
```

There are no widgets on the web, so we decide to disable this functionality for the web app. One way to do this is to detect the platform and enable the feature by condition:

```TypeScript
const iosDevice: Device = {
  platform: Platform.Ios,
  updateWidget: (data) => {}, // Bridge to the native iOS API...
};

const androidDevice: Device = {
  platform: Platform.Android,
  updateWidget: (data) => {}, // Bridge to the native Android API...
};

function update(device: Device, data: WidgetData) {
  if (
    device.platform === Platform.Android ||
    device.platform === Platform.Ios
  ) {
    // Calling `updateWidget` only on iOS and Android devices:
    device.updateWidget(data);
  }
}
```

It's not a problem if we have one or two such checks. But the more features we detect this way, the more conditions will appear in our code. If we feel that such checks are used too often, we can use the second option to solve the problem.

Let's add a "dummy device object" for the web app that will implement the `Device` interface but won't do anything in response to an `updateWidget` call:

```TypeScript
const webDevice: Device = {
  platform: Platform.Web,
  updateWidget() {},
};
```

Such "dummy objects" are called *null objects*.[9] They are usually used where we need a method call but don't need the implementation of that call.

When using a null object, we no longer need to check the type of the device—we just need to call the method. If the current device is a web browser, the method call will pass without doing anything:

```TypeScript
function update(device: Device, data: WidgetData) {
  device.updateWidget(data);

  // webDevice.updateWidget();
  // void;

  // The condition is no longer needed.
}
```

The null object is sometimes considered **antipattern**.[🔗] It's better to decide if it's appropriate to use it depending on the task, ways of use, and team preferences. It's best to consult with other developers before using it and make sure no one has any reasons against it.

---

1. "Your Code As a Crime Scene" by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene

2. Cyclomatic Complexity, Wikipedia, https://en.wikipedia.org/wiki/Cyclomatic_complexity

3. Control-Flow Graph, Wikipedia, https://en.wikipedia.org/wiki/Control-flow_graph

4. Control flow graph & cyclomatic complexity for following procedure, Stackoverflow, https://stackoverflow.com/a/2670135/3141337

5. "Cognitive Complexity. A new way of measuring understandability" by G. Ann Campbell, SonarSource SA, https://www.sonarsource.com/docs/CognitiveComplexity.pdf

6. "Code That Fits in Your Head" by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head

7. The Zen of Python, https://peps.python.org/pep-0020/#the-zen-of-python

8. "Application State Management with Finite State Machines" by Alex Bespoyasov, https://bespoyasov.me/blog/fsm-to-the-rescue/

9. De Morgan's Laws, Wikipedia, https://en.wikipedia.org/wiki/De_Morgan%27s_laws

10. `switch-exhaustiveness-check`, ES Lint TypeScript, https://typescript-eslint.io/rules/switch-exhaustiveness-check/

11. Pattern Matching, Wikipedia, https://en.wikipedia.org/wiki/Pattern_matching

12. ECMAScript Pattern Matching Proposal, https://github.com/tc39/proposal-pattern-matching

13. `ts-pattern`, Library for Pattern Matching in TypeScript, https://github.com/gvergnaud/ts-pattern

14. "Bringing Pattern Matching to TypeScript" by Gabriel Vergnaud, https://dev.to/gvergnaud/bringing-pattern-matching-to-typescript-introducing-ts-pattern-v3-0-o1k

15. Pattern matching in Haskell, Learn You Haskell, http://learnyouahaskell.com/syntax-in-functions#pattern-matching

16. Polymorphism, Wikipedia, https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

17. Strategy Pattern, Refactoring Guru, https://refactoring.guru/design-patterns/strategy

18. Introduce Null Object, Refactoring Guru, https://refactoring.guru/introduce-null-object

19. Null object pattern, Criticism, Wikipedia, https://en.wikipedia.org/wiki/Null_object_pattern#Criticism

11

Side Effects

Any interaction between an app and the real world is a side effect. Saving data on the server, rendering components in the UI, or accessing the browser API—all these are side effects. They're necessary for a useful application but are hard to reason about and can cause errors.

In this chapter, we'll discuss how to design programs and refactor code so that effects don't make the code more complicated and error-prone. We'll explore the benefits of functional programming and immutable data structures. We'll also discuss how to organize an app's effects and the difference between CQS and CQRS.

Pure Functions

The main problem with effects is that they're *unpredictable*. They change the state, so we can't be sure that the result of the code will always be the same.

On the other hand, *pure functions* don't produce effects, don't depend on the state, and always return the same result when given the same arguments. It makes their results predictable and reproducible.

When refactoring, we'll try to use pure functions more often and express more functionality with them. But to understand the benefits of this approach, let's first discuss the nature of pure functions.

Referential Transparency

It's easier to see the advantage of pure functions during debugging. We can use binary search to speed up the search for bugs in the problematic code snippet. Using it, we divide the snippet into two halves, detect an error in one of them and then look for bugs only in that half:

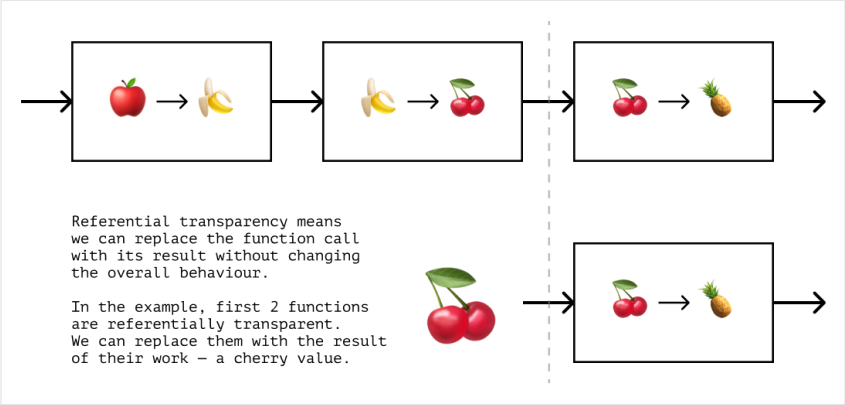
```
Let's say this is the problematic piece of code.
The error in it is marked with “X”:

[.....................X...........]

1.
To find the error in this snippet,
we divide the functionality in half
and check each half:

[...............|....X...........]

2.
If the left half works fine,
the error is in the right one:

[               |....X...........]

3.
We split the right part in half
and check each half in it:

[               |....X..|........]

4.
The right part works okay,
so the error is in the left one:

[               |....X..|        ]

5.
We again divide the problematic part in half and repeat the search.
We do this until we find the exact spot with the error:

[               |...|X..|        ]
[                   |X..|        ]
[                   |X|.|        ]
[                   |X|           ]

With each iteration, the problematic code area is halved.
This approach saves a lot of time when debugging.

In general, after 3-6 iterations, it becomes clear,
what the problem is and where it is.
```

> **By the way** 🔪
>
> Sometimes, such a binary search process is called bisection by analogy with the bisection in git.🔗

In a sequence of pure functions, the data flows from one function to another, being passed as arguments. The advantage of such a sequence is that we can “cut” it *anywhere*. The

functions after the "cut" don't care what functions have been called before them. They'll work the same way if given the same arguments.

We can literally replace the *calls* of functions before the "cut" with *their results*, and the program's behavior won't change. Such a property is called *referential transparency*,[2] and makes searching for errors and testing much more straightforward.



Referential transparency allows us to replace a function call with its result

With side effects, this is much harder to achieve. So when refactoring, we'll try and reduce the number of effects and use pure functions as often as possible.

Immutable by Default

As we said earlier, code with effects is less predictable. When refactoring, we should first check if we can rewrite it without effects. In practice, this most often means replacing the "shared state" with a chain of data transformations.

For example, look at the `prepareExport` function, which prepares store order data for export. It calculates the totals and the latest shipping date for each item.

```JavaScript
function prepareExport(items) {
  let latestShipmentDate = 0;

  for (const item of items) {
    item.subtotal = item.price * item.count;

    if (item.shipmentDate >= latestShipmentDate) {
      latestShipmentDate = item.shipmentDate;
    }
  }

  for (const item of items) {
    item.shipmentDate = latestShipmentDate;
  }

  return items;
}
```

The `subtotal` calculation within the function updates objects in the `items` array. However, other calculations also depend on this array, and changing it will affect them too.

It means that when calculating `subtotal` we'll have to consider how such changes will affect the `shipmentDate` calculation. The more extensive the function, the more actions will be affected, and the more details we must keep in mind.

Moreover, since `items` is an array, its changes will be visible *outside* the `prepareExport` function. They might affect code that we may know nothing about. In this case, it'll be impossible to predict potential problems.

Instead of keeping track of the effects, we can try to avoid them. Let's rewrite the code not to change the shared state but to express the problem as a sequence of steps.

The result of each step will be a *new* set of data. The steps themselves won't depend on any shared variables so that they won't affect each other's work:

```JavaScript
function prepareExport(items) {
  // 1. Calculate the subtotals.
  //    The result is a new array.
  const withSubtotals = items.map((item) => ({
    ...item,
    subtotal: item.price * item.count,
  }));

  // 2. Calculate the shipment date.
  //    The result of the previous step is the input.
  let latestShipmentDate = 0;
  for (const item of withSubtotals) {
    if (item.shipmentDate >= latestShipmentDate) {
      latestShipmentDate = item.shipmentDate;
    }
  }

  // 3. Append the date to each position.
  //    The result is yet another new array.
  const withShipment = items.map((item) => ({
    ...item,
    shipmentDate: latestShipmentDate,
  }));

  // 4. Return the result of step 3,
  //    as the result of the function.
  return withShipment;
}
```

We also can extract the steps into separate functions and, if necessary, refactor each one separately:

```JavaScript
function calculateSubtotals(items) {
  return items.map((item) => ({ ...item, subtotal: item.price * item.count }));
}

function calculateLatestShipment(items) {
  const latestDate = Math.max(...items.map((item) => item.shipmentDate));
  return items.map((item) => ({ ...item, shipmentDate: latestDate }));
}
```

Then the `prepareExport` function will look like the result of a data transformation sequence:

```JavaScript
function prepareExport(items) {
  const withSubtotals = calculateSubtotals(items);
  const withShipment = calculateLatestShipment(withSubtotals);
  return withShipment;
}

// items -> withSubtotals -> withShipment
```

Or even like this, if we use the Hack Pipe Operator, which at the time of writing is in Stage 2:

```JavaScript
const prepareExport =
  items |> calculateSubtotals(%) |> calculateLatestShipment(%);
```

It's not always necessary to extract the steps into separate functions, but it's convenient in several cases:

- If we need to test each step in isolation from the others.
- If we want to use data transformations elsewhere in the application.
- If we want to make each data state of the sequence more explicit.

Note that in the new implementation, the `prepareExport` function *doesn't change* the original data. Instead, it *creates a copy* of the data and changes it. The `items` array stays untouched, which prevents errors in the code outside the function.

The steps inside `prepareExport` are now connected only by their *input and output data*. They have no shared state that can affect their operation. It makes it easier for us to build a mental model of how the `prepareExport` function works. The function becomes a chain of data transformations, each of which is isolated from the others and can't be impacted from the outside.

> **Abstraction and encapsulation 👀**
>
> By abstracting each step into a separate function with a clear name, we make the meaning of the whole chain more explicit. It helps to focus on the individual steps without worrying about the others. Isolation helps to ensure the data validity at each step because it prevents the function from being affected "from the outside."

Immutability can be demanding on memory and cost performance. It isn't usually a problem on the frontend, but it's still worth knowing the potential issues. A mutable approach might be a better fit if we're chasing performance.

> **By the way 🚜**
>
> In JavaScript, real immutability is difficult to achieve. For truly immutable objects, we'd need `Object.freeze`, which is rarely used.
>
> But "real" immutability isn't always needed. Most of the time, it's enough to *write and treat* the code as if it were working with immutable data.

Functional Core in Imperative Shell

Immutability and pure functions are nice, but as we mentioned, we can't create a helpful app without any effects. Interactions with the outside world—receiving and saving data or rendering it in the UI—are always effects. Without these interactions, the application is useless.

Since the problem with effects is that they're unpredictable, our main concern with them is to:

> **Minimize the number of effects and isolate them from other code ❗**

There's a technique for managing effects called the *functional core in an imperative shell* or *Impureim Sandwich*.[🔗🔗] Using this approach, we describe the application logic as pure functions and "push" all interaction with the outside world *to the edges of the application*.

An "impureim sandwich": impure effect for reading the data, pure logic, impure effect for saving the data

Consider an example. The `updateUserInfo` function mixes transforming data with saving and reading it in the DOM:

```JavaScript
function updateUserInfo(event) {
  const { email, birthYear, password } = event.target;
  const root = document.querySelector(".userInfo");

  root.querySelector(".age").innerText = new Date().getFullYear() - birthYear;
  root.querySelector(".password").innerText = password.replace(/./g, "*");
  root.querySelector(".login").innerText = email.slice(
    0,
    email.lastIndexOf("@")
  );
}
```

Let's try to separate logic from effects. First, we can do this right inside the function by grouping the code into "sections":

```JavaScript
function updateUserInfo(event) {
  // Read data:
  const { email, birthYear, password } = event.target;

  // Transform data:
  const age = new Date().getFullYear() - birthYear;
  const username = email.slice(0, email.lastIndexOf("@"));
  const hiddenPassword = password.replace(/./g, "*");

  // "Save" data, in our case, render it in the UI:
  const root = document.querySelector(".userInfo");
  root.querySelector(".age").innerText = age;
  root.querySelector(".password").innerText = hiddenPassword;
  root.querySelector(".login").innerText = username;
}
```

Then we can extract the data transformation into a separate function. It would know nothing about reading and saving data and would only deal with the logic of transformation:

```JavaScript
function toPublicAccount({ email, birthYear, password, currentYear }) {
  return {
    age: currentYear - birthYear,
    username: email.slice(0, email.lastIndexOf("@")),
    hiddenPassword: password.replace(/./g, "*"),
  };
}
```

Then we can use the `toPublicAccount` function inside `updateUserInfo` like this:

```JavaScript
function updateUserInfo(event) {
  // Read data:
  const { email, birthYear, password } = event.target;
  const currentYear = new Date().getFullYear();

  // Transform:
  const { age, username, hiddenPassword } = toPublicAccount({
    email,
    birthYear,
    password,
    currentYear,
  });

  // “Save”:
  const root = document.querySelector(".userInfo");
  root.querySelector(".age").innerText = age;
  root.querySelector(".password").innerText = hiddenPassword;
  root.querySelector(".login").innerText = username;
}
```

To check if the code has gotten better, we can try and write tests for data transformation logic.[🔗] In the first version of the code, the test would look like this:

```JavaScript
// 1. In the case of `updateUserInfo`,
//    we need to create mocks for DOM and Event:
const dom = jsdom(/*...*/);
const event = {
  target: {
    email: "test@test.com",
    password: "strong-password-1234",
    birthYear: 1994,
  },
};

// We also need to mock the current date,
// so that the test results are reproducible:
jest.useFakeTimers().setSystemTime(new Date("2022-01-01"));

// Mocks and timers need to be reset after each test,
// so not to affect other tests:
afterAll(() => jest.useRealTimers());

describe("when given a user info object", () => {
  it("should calculate the user age", () => {
    updateUserInfo(event);

    // The result is checked by the DOM-node contents:
    const node = dom.document.querySelector(".userInfo .age");

    // It loses the data type
    // because DOM nodes contain only strings:
    expect(node.innerText).toEqual("28");
  });
});
```

Now let's write a test for the `toPublicAccount` function and compare it with the previous one:

```JavaScript
// 2. In the case of `toPublicAccount`, we don't need mocks.
//    For testing it, we only need input data
//    and the expected result.

describe("when given a user info object", () => {
  it("should calculate the user age", () => {
    const { age } = toPublicAccount({
      email: "test@test.com",
      password: "strong-password-1234",
      birthYear: 1994,
      currentYear: 2022,
    });

    // We can test the function by direct comparison,
    // no information is lost, including the data types:
    expect(age).toEqual(28);
  });
});
```

In the first case, the `updateUserInfo` function handles different tasks: transforming data and interacting with the UI. Its tests confirm this—they check how the data has changed and use DOM mocks.

If another similar function appears in its tests, we'll have to mock the DOM *again* to check for changes in the data. It should become a concern because there's obvious duplication with no additional benefits.

The test is much easier in the second case because we don't need to create mocks. We only need the input data and the expected result to test pure functions. (This is why it's often said that pure functions are intrinsically testable.)

Interacting with the DOM becomes a separate task. The mocks for the DOM will appear in the tests for the module that deals with UI interactions and nowhere else.

This way, we simplify the function code and improve the separation of concerns between the different parts of the application.

> **By the way** 🔌
>
> Here, we don't imply that "mocks are always bad." Sometimes mocks are the only way to test the desired effect, such as in adapters.
>
> The point is that if we have to write a mock to test *logic*, there's likely a better way to organize the code.

After refactoring, we can see that the task of the `updateUserInfo` function has turned into a "composition" of other functions' functionality. It now brings together reading data, transforming it, and saving it in storage.

The structure of the function has begun to resemble the sandwich with effects and logic. With an adequate separation of responsibility, the "layers" of the sandwich become completely independent. It makes the data transformations predictable and isolated from the effects.

Adapters for Effects

Separating logic and effects helps to detect duplication in code that reads and saves the data. It can be noticeable by identical mocks in tests or similar code in the effects themselves.

If different parts of an app interact with the outside world similarly, we can extract that interaction to a separate entity, the *adapter*. Adapters reduce duplication, decouple the application code from the outside world, and make testing of the effects easier:

JavaScript

```
// If we notice the same functionality
// when reading or saving data:

function updateUserInfo(user) {
  // ...

  if (window?.localStorage) {
    window.localStorage.setItem("user", JSON.stringify(user));
  }
}

function updateOrder(order) {
  // ...

  if (window?.localStorage) {
    window.localStorage.setItem("order", JSON.stringify(order));
  }
}

// We can extract it into an adapter:

const storageAdapter = {
  update(key, value) {
    if (window?.localStorage) {
      window.localStorage.setItem(key, JSON.stringify(value));
    }
  },
};

// And then use only the adapter
// in the other app code:

function updateUserInfo(user) {
  // ...
  storageAdapter.update("user", user);
}

function updateOrder(order) {
  // ...
  storageAdapter.update("order", order);
}

// This way, all the logic of interacting
// with the storage is described in the `storageAdapter`.
//
// If we need to test that interaction, we only need to test the adapter's methc
// without running additional checks in functions like `updateUserInfo`.
```

In detail 👀

We will discuss adapters in more detail in a separate chapter on architecture.

Commands and Queries

When we have pushed interaction with the outside world to the edges of the application, we can think about how exactly they affect the world. Different effects have different consequences:

- Some effects only *read* information and don't change the state of the world.
- Other effects *change* (add, update, and delete) the state.

The principle that divides the code responsible for these tasks is called *Command-Query Separation (CQS)*.

According to CQS, *queries* only return data and don't change the state, while *commands* change the state and return nothing. The purpose of CQS is to:

> **Keep reading and changing data separate !**

Mixing commands and queries makes the code complicated and unsafe. It's difficult to predict the result of a function if it can change the data during its call. For this reason, when refactoring, we should pay attention to the effects that violate CQS.

For example, let's look at the signature of the `getLogEntry` function:

```TypeScript
function getLogEntry(id: Id<LogEntry>): LogEntry {}
```

From the types, we can assume that this function somehow *gets* data from the logs. It can become a surprise if, in the implementation, we see:

```TypeScript
function getLogEntry(id: Id<LogEntry>): LogEntry {
  const entry =
    logger.getById(id) ?? logger.createEntry(id, Date.now(), "Access");

  return entry;
}
```

The problem with the function is its unpredictability. We can't know in advance what result we'll get after the function call. We can try to solve this problem by adding details to the function name:

```TypeScript
function getOrCreateLogEntry(id: Id<LogEntry>): LogEntry {}
```

There's more information now, but we still have no idea what the function will do. It can still either read an existing log entry or create a new one.

The less predictable a function is, the more problems we'll have debugging it. Debugging is faster when we have to check fewer assumptions. When we can't predict the behavior of a function, we need to *test more assumptions*. The debugging of such a function will take longer.

Also, unpredictable effects are much harder to test.[🔗] For example, to test the `getOrCreateLogEntry` function, we would have to write a test somewhat like this one:

```TypeScript
afterEach(() => jest.clearAllMocks());
afterAll(() => jest.restoreAllMocks());

describe("when given an ID that exists in the service", () => {
  it("should return the entry with that ID", () => {
    jest.spyOn(logger, "getById").mockImplementation(() => testEntry);
    const result = getOrCreateLogEntry("test-entry-id");
    expect(result).toEqual(testEntry);
  });
});

describe("when given an ID of an entry that doesn't exist", () => {
  it("should create a new entry with the given ID", () => {
    jest.spyOn(logger, "getById").mockImplementation(() => null);
    const spy = jest.spyOn(logger, "createEntry");

    const result = getOrCreateLogEntry("test-entry-id");
    expect(spy).toHaveBeenCalledWith("test-entry-id", timeStub, "Access");

    expect(result).toEqual({
      createdAt: timeStub,
      id: "test-entry-id",
      type: "Access",
    });
  });
});
```

From the number of mocks, we can assume that the function does "too much." If not written carefully, the mocks themselves may affect each other's functionality, making tests unreliable and fragile.

And finally, creation and reading data are conceptually different operations. Their reasons to change are different, so they better be kept separate.

Let's split this function into two:

```TypeScript
function readLogEntry(id: Id<LogEntry>): MaybeNull<LogEntry> {}
function createLogEntry(id: Id<LogEntry>): void {}
```

According to the signatures, we now see that the first function returns a result, while the second function *does something and returns nothing*. The signature already implies that the second function *changes* the state, so it's an effect.

```TypeScript
function readLogEntry(id: Id<LogEntry>): MaybeNull<LogEntry> {
  return logger.getById(id) ?? null;
}

function createLogEntry(id: Id<LogEntry>): void {
  logger.createEntry(id, Date.now(), "Access");
}
```

The code has become more predictable because the function signature no longer deceives us. On the contrary, it now helps us predict behavior before considering the implementation.

The tests for both functions now are independent. We don't need to mock the internal functionality of the `logger` service anymore to test the details of each effect. It's enough to check that the functions call the correct methods with the required data:

```TypeScript
// readLogEntry.test.ts

describe("when given an ID", () => {
  it("should call the logger service with that ID", () => {
    const spy = jest.spyOn(logger, "getById");
    readLogEntry("test-entry-id");
    expect(spy).toHaveBeenCalledWith("test-entry-id");
  });
});

// createLogEntry.test.ts

describe("when given an ID", () => {
  it("should call the logger service createEntry with that ID and default entry
    const spy = jest.spyOn(logger, "createEntry");
    createLogEntry("test-entry-id");
    expect(spy).toHaveBeenCalledWith("test-entry-id", timeStub, "Access");
  });
});
```

> **Simplification** 🚧
>
> Often, for adapters, we also need to check if they correctly convert data between the service and our app (a.k.a "adapt the interface"). But in this particular example, I didn't find it necessary and didn't focus on it.

CQS and Generated IDs

In the backend development, there's a typical pattern that contradicts CQS. Imagine an app where the database module returns a generated ID in response to creating an entity. The "create" action is a command and shouldn't return anything but returns an ID, so it violates the CQS.

In general, I don't find this violation fatal. After all, CQS is a recommendation. Its applicability should be evaluated on a case-by-case basis. If returning the ID is a common pattern in the project, there's nothing wrong with using it. We just should be consistent with it and document the reasons behind it.

But if we don't want to deviate from CQS, we can pass the ID along with the data we want to save. It won't fit every project, but it might be a valuable option to consider.

> **Read more** 👀
>
> This solution is well described in the article "CQS versus server-generated IDs" by Mark Seemann.🔗 He explains the solution itself, its variations, applicability, and drawbacks in detail.

CQRS

Speaking of the backend, it's worth mentioning the CRUD operations and CQRS.🔗🔗 When designing an API, we may want to use the same data types for reading and writing data:

```TypeScript
type UserModel = {
  id: Id<User>;
  name: FullName;
  birthDate: DateTime;
  role: UserRole;
};

function readUser(id: Id<User>): UserModel {}
function updateUser(user: UserModel): void {}
```

In most cases, such a solution is sufficient and will not cause problems. However, it can become a problem if we read and write data differently. For example, if we want to update the user data only partially.

The `updateUser` function requires the entire `UserModel` object as input, so we cannot update individual fields. We have to pass the whole updated object into the function.

If a project encounters such a problem, the *Command-Query Responsibility Segregation, CQRS* might help.🔗 This principle extends the idea of CQS by suggesting using different types (also called "models") for reading and writing data.

Continuing with the `UserModel` example, we can express the essence of CQRS this way:

```TypeScript
// For reading we use one type, `ReadUserModel`:
function readUser(id: Id<User>): ReadUserModel {}

// For writing we use another type, `UpdateUserModel`
function updateUser(user: UpdateUserModel): void {}
```

Independent models make it explicit what data to provide when writing and what data to expect when reading. For example, we can describe the `ReadUserModel` type as a set of mandatory fields that are guaranteed to exist in the data when reading:

```TypeScript
type ReadUserModel = {
  id: Id<User>;
  name: FullName;
  birthDate: DateTime;
  role: UserRole;
};
```

For updates, we can use a different type:

```TypeScript
type UpdateUserModel = {
  // ID is mandatory to make it clear,
  // which user's data to update:
  id: Id<User>;

  // Everything else is optional,
  // to update only what we need:
  name?: FullName;
  birthDate?: DateTime;

  // The role, for example, cannot be updated at all,
  // that is why this field doesn't exist here.
};
```

This way, neither of the types prevents us from declaring *different* expectations for reads and writes.

> **Be careful 🚧**
>
> CQRS increases the amount of code (models, objects, types) in the project. It's worth discussing the approach with the team before using it and ensuring there are no arguments against it.
>
> The main reasons to use CQRS are the differences in types for reading and writing and the difference in the backend load for reading and writing operations that sometimes leads to separate scaling of reads and writes.
>
> In other cases, using a generic model will probably be easier and cheaper.

1. git-bisect, Use binary search to find the commit that introduced a bug, https://git-scm.com/docs/git-bisect

2. Referential Transparency, Haskell Wiki, https://wiki.haskell.org/Referential_transparency

3. "A pipe operator for JavaScript" by Axel Rauschmayer, https://2ality.com/2022/01/pipe-operator.html

4. "Functional Core in Imperative Shell" by Gary Bernhardt, https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell

5. "Impureim Sandwich" by Mark Seemann, https://blog.ploeh.dk/2020/03/02/impureim-sandwich/

6. "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing

7. "Command-Query Separation" by Martin Fowler, https://martinfowler.com/bliki/CommandQuerySeparation.html

8. "CQS versus server generated IDs" by Mark Seemann, https://blog.ploeh.dk/2014/08/11/cqs-versus-server-generated-ids/

9. Create, Read, Update, and Delete, Wikipedia, https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

10. "Command-Query Responsibility Segregation" by Martin Fowler, https://martinfowler.com/bliki/CQRS.html

12

Error Handling

Error handling is a big topic. We can do it in many ways, and details will depend on the project, its constraints, the code style, and often the team preferences.

In this chapter, we won't try to cover every possible way to handle errors. Instead, we'll look at general refactoring techniques that are appropriate for *some* projects and can help improve the code.

> **Clarification** 🎯
>
> In the text, I don't want to "sell" any particular way to handle errors. Instead, I want to show what I focus on when refactoring the code. I plan to discuss how to deal with various project constraints and limitations that can make it difficult to refactor error handling.
>
> **I can be wrong**. These techniques help *me* but this doesn't mean they're universally applicable. I recommend studying how error handling works in your project before using them. Discuss the ideas with your team and ensure everyone is okay with them.

Proper error handling helps cope with abnormal situations in an application and debug problematic code. We can achieve it in many ways, but most often, when refactoring, we can focus on three heuristics:

- If an error makes the correct code execution impossible, we should handle it instead of trying to continue working "normally."
- We should make error handling centralized but not mix different kinds of errors.
- We must not silently "swallow" any errors, especially unhandled ones.

In this chapter, we'll see how each point of the list is practical and how to implement them in our code. But before we get down to refactoring, let's agree on the terms and discuss the types of errors.

Types of Errors

In "Domain Modeling Made Functional," Scott Wlaschin divides errors into three types:

- *Domain errors*. These errors are expected in the business workflows, their cause lies in the application domain, and we know how to handle them. (For example, dividing by 0 in a calculator app is a domain error because this operation constraint is a part of the maths domain.)
- *Infrastructural errors*. These are also expected, and we know how to handle them, but they're related to the infrastructure, not the business logic. (For example, a failed network request is an infrastructural error because we expect that it might happen, but it's not a part of the domain.)
- *Panics*. These are unexpected errors. We don't know how to handle them and recover the app after they happen. (For example, getting `null` where we shouldn't have is a panic because we don't know how to make the application state valid again.)

> **By the way**
>
> I don't use the term "exceptions" because in different sources, "exceptions" and "errors" mean exactly the opposite. I will use the term "panics" instead.

We will often refer to this list during refactoring. It'll help us to choose appropriate handling techniques for specific cases.

Handling Techniques

As we said, error handling techniques depend on the specific language, project limitations, and team preferences. But most often, error handling is done using these techniques:

- Throwing panics;
- Using result containers;
- Combining panics and containers;
- Combining containers and functional binding.

We won't "rate" them "good or bad." Instead, we'll investigate the pros and cons of each of them and study what project limitations can force us to use a particular technique.

For example, first, we'll discuss how to refactor code in a project where we can only use panics. Then we'll discuss how result containers can be helpful and when to use them. Closer to the end of the chapter, we'll see how to combine the different techniques.

> **Be careful** 🚧
>
> The techniques in the book are ideas, not rules. The decision to apply them should be made case-by-case, in consultation with other developers.

Throwing Panics

The most common way to deal with errors in JavaScript code is to throw a panic with a native `throw new Error()` statement. It's usually the first thing that pops into the head when we want to "indicate that something went wrong." And it's reasonable since the `Error` object works natively out of the box and doesn't require any additional tools.

> **By the way** 👀
>
> Besides `Error`, there's also `Promise.reject`, but it's more related to asynchronous operations. For example, in business logic, it's almost never used.

The main problem is that panics are more suitable for unexpected errors than expected ones. They describe situations that lead the program to an inconsistent state that *can't be recovered from*.

Expected errors, however, are recoverable, and we know how to handle them. In business logic, for example, most of the errors are recoverable. It's even recommended to avoid panics in the business logic because it should work even after removing all the panics.

In practice, however, this separation isn't always maintained. There are projects where panics and errors are mixed. In such projects, it's often difficult to introduce new methods of error handling, and we might have to deal with errors only using panics. But even in such projects, we can still improve the code.

Consider an example. Let's say we have a function `getUser` that calls the backend API to fetch data about a user. Once it gets a response, it parses it and stores the result in the storage.

```JavaScript
async function getUser(id) {
  const dto = await fetchUser(id);
  const user = dto ? parseUser(dto) : null;
  if (user) storage.setUser(user);
  else storage.setError("Something went wrong.");
}
```

The `fetchUser` function requests the network and returns the DTO from the server or `null` if the server responds with an error:

```JavaScript
async function fetchUser(url) {
  const response = await fetch(url);
  const { value, error } = await response.json();
  if (error) return null;
  return value;
}
```

The `parseUser` function parses the server response and returns the user object or `null` if the DTO is invalid:

```JavaScript
function parseUser(dto: UserDto): User | null {
  if (!dto || !dto.firstName || !dto.lastName || !dto.email) return null;
  return { ...dto, fullName: `${dto.firstName} ${dto.lastName}` };
}
```

To understand how to start refactoring this code, let's first identify the problems in it:

- There's no error handling as such. We return `null` from the functions when something goes wrong, but we *swallow* the error reasons and don't handle them in any way.
- Because of `null` in the results of `fetchUser` and `parseUser`, we lose the context of the error. So we have to *check the data for the same errors again* at the level above. Duplicate checks make the code noisy.
- The `fetchUser` function addresses *only some* problems, and there's no explicit delegation of unexpected errors to other modules. It makes the code "unsafe" because the app can crash anytime.
- We *don't distinguish* between infrastructural and domain errors. We might need this information to find bugs in the app faster when analyzing issues and bug reports.

Let's try to fix these problems.

Unexpected Errors and Missing Context

One of the reasons for refactoring error handling is the uncertainty of what errors we will catch. If we miss an error it might crash the app when we don't expect it to happen.

For example, if an unexpected error occurs inside the `fetchUser` function, the application will crash:

```JavaScript
async function fetchUser(url) {
  const response = await fetch(url);

  // Let's say, after unpacking JSON, we got `null` instead of an object.
  // The next line will then throw `null is not an object`:
  const { value, error } = await response.json();
  // ...
}
```

We can solve this by adding `try-catch` at the level above. Then the `getUser` function will catch the thrown error, and we'll be able to handle it:

```JavaScript
async function getUser(id) {
  try {
    const dto = await fetchUser(id);
    const user = dto ? parseUser(dto) : null;
    if (user) storage.setUser(user);
    else storage.setError("Something went wrong.");
  } catch (error) {
    storage.setError("Couldn't fetch the data.");
  }
}
```

However, if we throw panics to handle *all* errors, we can accidentally mix expected and unexpected errors. For example, if we also handle validation errors of `parseUser` this way:

```JavaScript
function parseUser(dto: UserDto): User {
  if (!dto) throw new Error("Missing user DRO.");

  const { firstName, lastName, email } = dto;
  if (!firstName || !lastName || !email) throw new Error("Invalid user DRO.");

  return { ...dto, fullName: `${firstName} ${lastName}` };
}
```

Then `try-catch` at the level above will catch them, but we won't be able to distinguish between network errors and validation errors:

```JavaScript
async function getUser(id) {
  try {
    // ...
  } catch (error) {
    // Is `error` a network error or a validation error?
    // Is it expected or not?
    //
    // We can distinguish errors by the message
    // that was passed to the `Error` constructor,
    // but this is unreliable.
  }
}
```

> **Why difference matters** 🤔
>
> We want to distinguish between errors and panics because they may need to be handled differently. Depending on the project requirements, we may want, for example, to log panics in an alert monitoring service and log expected errors in the analytics service. The easier it is to distinguish them, the less code we'll need to handle those separately.

We can use different error types to distinguish panics from domain and infrastructure errors.

Different Error Types

In the case of JavaScript, an "error type" is a separate class that extends `Error`. In such extensions, we can specify the name and kind of the error and some additional information in separate fields.

For example, to distinguish between network errors and validation errors, we can create such types:

```JavaScript
// Validation errors:
class InvalidUserDto extends Error {
  constructor(message) {
    super(message ?? "The given User DTO is invalid.");
    this.name = this.constructor.name;
  }
}

// API errors:
class NetworkError extends Error {
  constructor(message, status, traceId) {
    this.name = this.constructor.name;
    this.message = message ?? messageFromStatus(status);

    // We can extend the type with additional fields for logging:
    this.status = status;
    this.traceId = traceId;
  }
}
```

> **By the way** 👀
>
> If we need to throw several errors at once during validation instead of throwing one at a time, we can extend classes with additional fields or use `AggregateError`.

Then when catching, we can understand what exactly happened from the type:

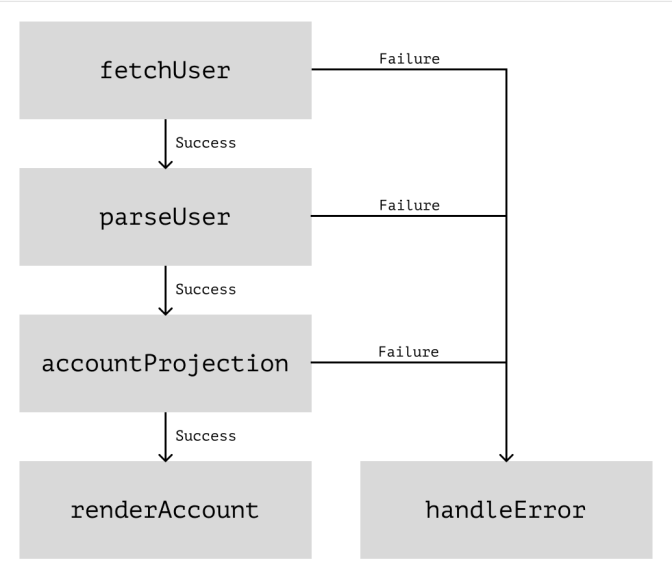
```JavaScript
async function getUser(id) {
  try {
    // ...
  } catch (error) {
    if (error instanceof InvalidUserDto) {
    } else if (error instanceof NetworkError) {
    } else throw error;
  }
}
```

Fail Fast

This way of error handling has many disadvantages: it uses panics in the business logic code and violates LSP inside the `catch` block. But despite this, it has an advantage: when an error occurs, we don't try to continue working but go on to handle it.

We don't want to continue working because an error is an *inconsistent state* of the application. In such a state, we cannot guarantee that the application data is valid, and then we can continue working with it.

If we terminate execution at this point and move on to handling the error, we protect ourselves from further errors that could damage the data even more. In addition, we clean up the domain model code, removing unnecessary checks from it.



Code execution becomes linear. All error branches lead immediately to error handling

Rethrow

You might notice that in the last example, on the last line of the `catch` block, a new expression `else throw error` appeared. It is a so-called *rethrow*. We use it as a mechanism that helps us *not to swallow* errors that we can't handle.

If the current error handler checked the error for all known types and couldn't determine what to do with it, it can rethrow it to a level above. On that level, the error will be handled by "someone smarter, who knows what to do."

> **However ❓**
>
> This, of course, raises the question of who will handle it on the level above. We'll answer this at the end of the chapter.

After the refactoring, the functions start looking linear, and the number of checks for `null` decreases:

```JavaScript
async function getUser(id) {
  try {
    const dto = await fetchUser(id);
    const user = parseUser(dto);
    storage.setUser(user);
  } catch (error) {
    if (error instanceof InvalidUserDto || error instanceof NetworkError) {
      storage.setError(error.message);
    } else throw error;
  }
}
```

Advantages

We have achieved some improvements. Although they are few, they can simplify working with the code:

- Execution has become more *linear*. We stopped re-checking the data for errors already checked, so the code flow became more "straightforward."
- The code moves on to error handling as soon as *normal execution* becomes impossible. We stop the error at the beginning, preventing it from bringing the application to an inconsistent state.
- The *context of the errors is saved*, so we have full information about what happened when processing begins.

Problems

However, throwing has its problems as well:

- A thrown error can crash the app at the run time if it isn't handled.
- Yet there are no tools in the language that *force us* to handle the potential errors, so it's easy to forget or skip handling them.
- There's almost no syntactic difference between errors and panics in the code, which makes it hard to separate them from each other.
- At the same time, using panics in the domain code is a smell because domain errors aren't panics; we expect them.
- Checking for all potential errors is possible, but it looks ugly because of `instanceof`.
- We can avoid the use of `instanceof` with error subclasses for each "application layer," but this makes the error model more complex.
- It isn't clear *who* should handle a thrown error. Relying on "conventions" isn't safe because there are no tools in the language to enforce conventions.
- There are no explicit rules for wrapping "low-level" code (`fetch`, Browser API, etc.) in `try-catch`.
- Performance may suffer because each `Error` object collects stack and other information.

- Function signatures don't tell us that the functions may throw an error; we can only learn about the errors from the source code.

> **Clarification** 🎯
>
> In TypeScript, we can use the `never` type to warn about possible errors in a function through its signature. This type doesn't tell *what* errors to expect, but at least it hints at their *possibility*. It makes the signature a bit more accurate but probably isn't enough to describe errors in more detail.

If a project can use *only* panics, that's probably the maximum we can do. In a good way, we should avoid using panics in business logic. But if there are some constraints in the project that force us to do so, it makes sense to make the error handling look roughly like this.

However, there are other ways to handle errors besides throwing panics. If we examine the project and see something that resembles *result containers* in the code, we can improve the `getUser` function further.

Result Containers

As we mentioned before, some errors are expected. For example, we can expect the API to return "404" in response to a request for a non-existing page. This situation is non-standard, but we know how to handle it.

It might be convenient to use *result containers* to handle expected errors. A container is a type that lives in one of two states: either the result of a successful operation or the occurred error.

> TypeScript
>
> ```
> // The container is a "box" that can be in 1 of 2 states:
> // - `Success`, for returning a result
> // - `Failure`, for returning an error
>
> type Result<TOk, TErr> = Success<TOk> | Failure<TErr>;
> type Success<T> = { ok: true; value: T };
> type Failure<E> = { ok: false; error: E };
> ```

> **Clarification** 🦄
>
> The implementation of `Result` in the example is intentionally approximate and incomplete so as not to pretend to be "canonical."
>
> Implementing your own container from scratch is an interesting task, but I'd recommend using solutions known to the community in production. For example, `Either` from the fp/ts library would be good for such tasks.
>
> If your project already uses some implementation of containers, study its capabilities. It probably already has everything you need.

Using a container, we could rewrite the `parseUser` function something like this:

```TypeScript
type MissingDtoError = "MissingDTO";
type InvalidDtoError = "InvalidDTO";
type ValidationError = MissingDtoError | InvalidDtoError;

function parseUser(dto: UserDto): Result<User, ValidationError> {
  if (!dto) return Result.failure("MissingDTO");

  const { firstName, lastName, email } = dto;
  if (!firstName || !lastName || !email) {
    return Result.failure("InvalidDtoError");
  }

  return Result.success({ ...dto, fullName: `${firstName} ${lastName}` });
}
```

The function now returns a *"box" with either result or error*. This "box" encapsulates information about what happened and returns all of it to a higher level.

More Accurate Signature

With a container, we see possible errors right in the function signature. We don't need to examine the source code to know what errors we expect. *All of the expected outcomes* of the operation are reflected in the returned type.

Using containers has a downside, of course. To use data from the result at the level above, we have to "unpack" the container:

```TypeScript
async function getUser(id) {
  try {
    const { value: dto, error: networkError } = await fetchUser(id);
    const { value: user, error: parseError } = parseUser(dto);
    storage.setUser(user);
  } catch (error) {
    // ...
  }
}
```

If there is an error, the `value` field will be empty when unpacking. That way, passing the data to the next function won't be possible. So further work is possible *only* if there are no errors or if they are handled. This way, the container signature forces us to remember potential errors and handle them:

```TypeScript
async function getUser(id) {
  try {
    const { value: dto, error: networkError } = await fetchUser(id);
    // Handle the `networkError`...

    const { value: user, error: parseError } = parseUser(dto);
    // Handle the `parseError`...

    storage.setUser(user);
  } catch (error) {
    // Handle unexpected situations...
  }
}
```

Containers are probably a better solution for describing expected errors than panics. They are syntactically different from panics, which may help to think of them as *expected*. Also, we don't need to *throw* containers, which means they won't break the application while we unpack them correctly.

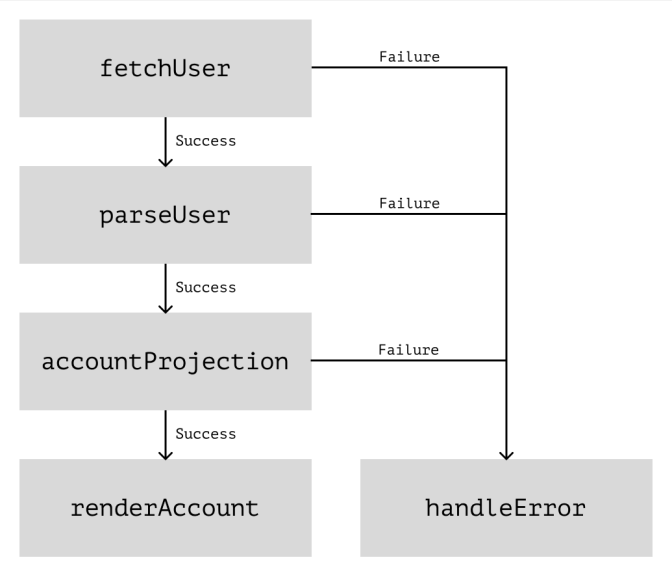
> **By the way** 📦
>
> In Node.js, in the days of callbacks, unsafe operations returned a container—a tuple of error and value:[9]
>
> `function errorFirstCallback(err, value) {}`
>
> If there were an error, the `err` field would contain it. If there were no errors, `err` would be empty. It's also known as error-first callbacks.

Explicit Handling

When unpacking the containers, we can set up error handling so that when an error occurs, we interrupt the "normal" execution and handle the error:

```TypeScript
async function getUser(id) {
  try {
    const { value: dto, error: networkError } = await fetchUser(id);
    if (networkError) return handleError(networkError);
    // Or `throw new Error(networkError)` and handle it later.

    const { value: user, error: parseError } = parseUser(dto);
    if (parseError) return handleError(parseError);

    storage.setUser(user);
  } catch (error) {
    // ...
  }
}
```

This approach is also known as “Fail Fast.”



The results chain breaks where an error occurs, and the code starts handling the error

Clarification

In imperative code with many operations, it will look cumbersome. But if the number of operations is small (1-2), it’s not so terrible. We’ll talk about how to unpack containers more elegantly later.

Centralized Handling

Despite the need to handle errors explicitly, we can still set up centralized error handling. We can use a separate function for handling use case errors like this:

```TypeScript
type ValidationError = MissingDtoError | InvalidDtoError;
type NetworkError = BadRequest | NotFound | ServerError;
type UseCaseError = ValidationError | NetworkError;

// The `handleGetUserErrors` function handles errors in the `getUser` use case:
function handleGetUserErrors(error: UseCaseError): void {
  const messages: Record<UseCaseError, ErrorMessage> = {
    MissingDTO: "The user DTO is required.",
    InvalidDTO: "The given DTO is invalid.",
    // `Record<UseCaseError, ErrorMessage>` will make sure
    // we've covered all the expected errors.
  };

  // If the error is unexpected, rethrow:
  const message = messages[error];
  if (!message) throw new Error("Unexpected error when getting user data.");

  // If we expect it, we handle it:
  storage.setError(message);

  // If necessary, we can add some infrastructure stuff:
  logger.logError(error);
  analytics.captureUserActions();
}
```

Panics are Separated

Since the “low-level” code doesn’t return the container but throws panics, we need to wrap such operations in `try-catch` to return the container in case of a problem:

```TypeScript
type NetworkError = BadRequest | NotFound | InternalError;

async function fetchUser(url) {
  try {
    const response = await fetch(url);
    const { value, error } = await response.json();
    return error ? Result.failure("BadRequest") : Result.success(value);
  } catch (error) {
    //
    // If we expected the problem, wrap it in the container:
    const reason = errorFromStatus(error);
    if (reason) return Result.failure(reason);
    //
    // If not, rethrow:
    else throw new Error("Unexpected error when fetching user data.");
  }
}
```

We would need to wrap *every* call to the "low-level" API—in our case, every request to the network. It may increase the amount of code. However, if the scheme of working with these APIs is the same, we can reduce duplication by using decorators:

```TypeScript
// The decorator will accept the “unsafe” function,
// and call it inside `try-catch`.
// When an error occurs, it will check,
// if the error is expected and returns the container
// or rethrow the error.

function robustRequest(request) {
  return async function perform(...args) {
    try {
      return await request(...args);
    } catch (error) {
      const reason = errorFromStatus(error);
      if (reason) return Result.failure(reason);
      else throw new Error("Unexpected error when making a request.");
    }
  };
}

// We can use such a decorator like this:

const safeFetchUser = robustRequest(fetchUser);
const safeCreatePost = robustRequest(createPost);
```

Multiple Errors

We can store additional data in the `error` field of the container. We can pass objects, arrays, or even `Error` instances into this field. For example, it can be helpful for validation errors containing a list of invalid fields.

> **By the way** 👍
>
> We can handle multiple errors with the Notification pattern.[🔗] It can also be considered a sort of container.

Unpacking

The main problem, of course, still persists. We still have to unpack containers and interrupt code execution manually. It's not hard to do it a couple of times, but more frequent manual unpacking can make the code noisy. In this situation, functional binding can help us.

Binding Results

> **Before start** 🚂
>
> The essence of functional binding is well written in "Railway Oriented Programming" by Scott Wlaschin.[🔗] I suggest you read that article before continuing.

> **Also** 🙃
>
> If your project uses manual unpacking and you're okay with it, it isn't necessary to use the functional binding. It's only needed if you feel like manual unpacking is tiresome.

The main idea behind binding is to make the *container* take care of calling several functions one after the other. Right now, we can't call functions directly one by one—their inputs and outputs aren't compatible with each other:

> ```TypeScript
> function fetchUser(id: UserId): Result<UserDTO, FetchUserError> {}
> function parseUser(dto: UserDTO): Result<User, ParseUserError> {}
>
> // We want to call them in a chain like this:
> // fetchUser -> parseUser -> storage.setUser
>
> // But we can't do it now because the output of one function
> // doesn't match the input type of the following function.
> //
> // The `fetchUser` function returns `Result<UserDTO, FetchUserError>`,
> // but `parseUser` needs just `UserDTO`.
> ```

We can think of binding as "adapting" the input-output types of functions so that they can be chained together. If we were to do it manually, we'd do something like:

```TypeScript
function parseUser(
  result: Result<UserDTO, FetchUserError>
): Result<User, ParseUserError> {}
```

But it's not an option because we don't want a function to depend on the result of a previous operation. Instead, we want this "adaptation" to happen automatically.

For this, "railway-oriented programming" suggests using special functions-adapters. Depending on the implementation, these adapters can be either functions or container methods. The details aren't that important; the main point is the idea of "automating the adaptation":

```TypeScript
// The implementation is schematic!

const Result = {
  // ...

  // If the current container is in the `Success` state,
  // we "unpack" the value and pass it to the input of the `fn` function.
  // If the container is in the `Failure` state,
  // we return a new container `Failure<E>` with `error`,
  // which was in that container.
  bind: (fn) => (ok ? fn(value) : failure(error)),

  // At the end of the execution chain
  // we need to handle `error` or a result:
  match: (handleError, handleValue) =>
    ok ? handleValue(value) : handleError(error),
};

// The `bind` method is going to be used to chain functions;
// and the `match` method will "unpack" the final results of the hole chain.
//
// In the `match` method, there are two arguments:
// - the first function is an error handler that should handle errors of the fur
// - the second function is the result handler.
```

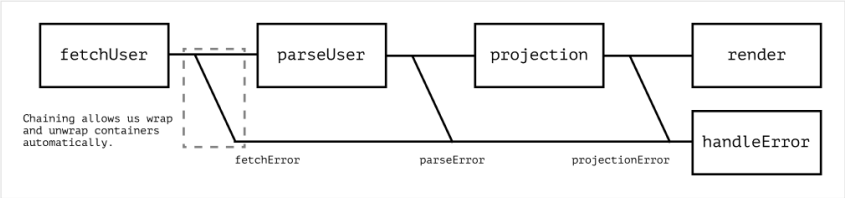
Now, we can bind functions in a chain like this:

```TypeScript
async function getUser(id) {
  (await fetchUser(id))
    .bind(parseUser)
    .match(handleGetUserErrors, storage.setUser);
}
```

An error in any of the steps immediately breaks the execution of such a function. After that, the `handleGetUserErrors` function starts to handle it. In combination with Exhaustiveness Check[🔗] we can safely handle all the errors for `getUser` without having to unpack containers manually.

> **By the way** ⛓
>
> Binding can look differently depending on the style and paradigm of the project. It can be done with methods, or it can be a function like `pipe`, which "pushes" the results from one function to another.
>
> If your project has containers, take a closer look at their implementation. Maybe they have the binding too. If not, look at existing solutions like fp/ts or sweet-monads and pick the one you like best.

So the point of binding is to *automate* the chaining and unpacking of results. It's like we link all results into a sequence of "forks." An error from any function travels down the fork to the "error track" and goes along it all the way to the end. It's similar to connecting railroad tracks —that's why the programming is "railway oriented" 😃



Binding helps to adapt function inputs and outputs

▶ What about native binding in JavaScript?

Problems

Despite being attractive, the functional binding also has problems:

- The code gets more complex. This can increase the entry threshold into a project.
- The approach requires conventions. We must check how the containers and wrappers for low-level functions are used. (And if they're used at all).
- In non-functional code, such code can look questionable and ambiguous.
- JavaScript lacks native pattern matching for easier use of such ideas.

When to Prefer Panics

Containers can't wholly replace panics. They're slightly different tools. Therefore, panics can be helpful when working with containers, too.

For example, a container might not be necessary if it doesn't matter what error occurred. You can find a good checklist of choosing between containers and panics in the article "Against Railway-Oriented Programming" by Scott Wlaschin.

Cross-Cutting Concerns

When error handling is consistent throughout the project, it's easier for us to track down the bugs in the code by, for example, a bug report. But apart from that, centralized error handling allows us to compose cross-cutting concerns like logging conveniently.

If error handlers are isolated, they can be *decorated* with additional functionality. For example, we can add logging to error handlers with decorators:

```TypeScript
// services/logger.js
// Logging service provides a function
// for sending a new event:

const logEvent = (entry: LogEntry) => {};

// infrastructure/logger.js
// To avoid adding logs to each handler separately,
// we can create a decorator that takes a function,
// calls it and logs the result of the call:

const withLogger =
  (fn) =>
  (...args) => {
    const result = fn(...args);
    const entry = createEntry({ args, result });
    logEvent(entry);
  };

// network.js
// To use it, it'll be enough to wrap
// the handler with the logging decorator:

const handleNetworkError = (error: NetworkError) => {};
const errorHandler = withLogger(handleNetworkError);
```

In general, it's convenient to compose cross-cutting concerns with decorators. Decorators make the "service" functionality isolated from the application code, and they're easy to "inject" into different parts of the code.

> **Clarification** 🖼️
>
> Decorators have limitations. They won't fit in all cases. For example, if we need to send a message to the log in the middle of a function, it would probably be hard to do that with a decorator.
>
> Though in such cases, it's usually worth considering why we need to log something *in the middle* of a function. Maybe it's better to split the function into several ones.

So Many Different Handlers

Potentially dangerous functions may be different and may be associated with different APIs. The requirements to error handling in such functions may differ. For example, some APIs require specifying the `.onerror` property or listening for an error event instead of using `try-catch`.

Centralized handling helps to solve this problem too because it decouples the *handlers themselves* from the *place where they're used*. One handler becomes possible to use with different APIs and handle errors from different modules with it.

> **By the way** 🔌
>
> If the signature of the error handler and service API are incompatible, we can use an adapter to solve this problem.🔗

Error Handling Hierarchy

Earlier, we mentioned rethrowing errors. It's delegating an error to a handler "above" if the current one doesn't know how to handle it.

Delegation creates a hierarchy of handlers. As with any hierarchy, it's best to keep it flat. It isn't always possible to achieve this, but the flatter the hierarchy, the easier it is to handle. One convenient way of organizing handlers consists of 3 levels:

Wrappers for "Low-Level" Code

At the "low" level we wrap in `try-catch` browser APIs, functions for working with the network, communicating with the device, etc.

Wrapper functions catch low-level code panics. Depending on the error handling style, they further convert the panics into containers or rethrow them.

Use Case Handlers

At this level, we handle use case errors: business logic errors and expected low-level infrastructure errors.

> **By the way** 🍰
>
> Such handlers can catch errors in a single use case or a set of related features. The latter can be called application slice handlers.

Signaling the user of problems is usually easiest at this level because there's access to services that can, for example, display a message on the screen or send a request to the alert monitor. It won't be the best place in all cases, but most of the time, it's convenient to build the heavy lifting of error handling at this level.

Last Resort Handlers

At the entire application or web page level, we catch all the errors and panics not handled before.

Here it's useful to log the error in the alert monitoring tool for future analysis. Panics are the easiest to work with at this level because they have a stack trace that makes it easier to search for the cause of the problem when analyzing the error.

> **For example** 📌
>
> In React, Error Boundaries can be handy in use case handlers and last resort handlers.🔗 Error Boundaries errors during rendering and render a separate UI where developers can inform the user about the problem.

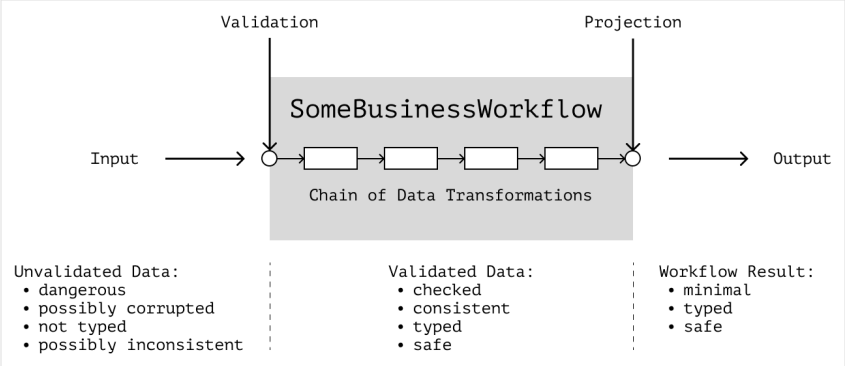
> **By the way** 🗒
>
> In JavaScript, we can catch unhandled errors at the global object level with special events.🔗🔗🔗 These events usually contain information about why and where the error occurred, so it is practical to combine their handling with logging and alert monitoring tools.

Data Prevalidation

In the chapter on the functional pipeline, we mentioned data prevalidation at the input to the application. We can also think of it as a part of error handling because it interrupts program execution when invalid data appear.

Prevalidation cleans the business logic code from irrelevant checks and allows us to focus on data transformations in the business workflows. With prevalidation, all validation errors are collected in one place, and their handling becomes more flexible. For example, prevalidation makes it easier to bundle validation errors into sets and handle them all at once rather than one at a time.

When refactoring error handling, we can take advantage of this idea and move the data checks closer to the input of the application:



Outside data is dangerous, inside data is safe; invalid data will terminate execution and pass control to the error handler

> **Read more** 🛟
>
> We will talk more about prevalidation, postvalidation, selectors, and data security in the chapter about architecture.

---

1. "Domain Modeling Made Functional" by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional

2. "Errors Are Not Exceptions" by swyx, https://www.swyx.io/errors-not-exceptions

3. "The Error Model" by Joe Duffy, http://joeduffyblog.com/2016/02/07/the-error-model/

4. "Replacing Throwing Exceptions with Notification in Validations" by Martin Fowler, https://martinfowler.com/articles/replaceThrowWithNotification.html

5. "The Pragmatic Programmer" by Andy Hunt, https://www.goodreads.com/book/show/4099.The_Pragmatic_Programmer

6. `AggregateError`, MDN, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AggregateError

7. "A behavioral notion of subtyping" by Barbara H. Liskov, Jeannette M. Wing, https://dl.acm.org/doi/10.1145/197320.197383

8. More on Functions, `never`, TypeScript: Documentation, https://www.typescriptlang.org/docs/handbook/2/functions.html#never

9. fp/ts, Typed functional programming in TypeScript, https://github.com/gcanti/fp-ts

10. Error-first callbacks, Node.js Documentation, https://nodejs.org/api/errors.html#error-first-callbacks

11. Fail-fast, Wikipedia, https://en.wikipedia.org/wiki/Fail-fast

12. "Notification" by Martin Fowler, https://martinfowler.com/eaaDev/Notification.html

13. "Railway Oriented Programming" by Scott Wlaschin, https://fsharpforfunandprofit.com/rop/

14. `switch-exhaustiveness-check`, ES Lint TypeScript, https://typescript-eslint.io/rules/switch-exhaustiveness-check/

15. sweet-monads, Easy-to-use monads implementation with static types definition, https://github.com/JSMonk/sweet-monads

16. neverthrow, Type-Safe Errors for JS & TypeScript, https://github.com/supermacro/neverthrow

17. `pipe`, fp/ts, https://gcanti.github.io/fp-ts/modules/function.ts.html#pipe

18. "Against Railway-Oriented Programming" by by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/against-railway-oriented-programming/

19. Cross-Cutting Concern, Wikipedia, https://en.wikipedia.org/wiki/Cross-cutting_concern

20. “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head

21. Decorator Pattern, Refactoring Guru https://refactoring.guru/design-patterns/decorator

22. Adapter Pattern, Refactoring Guru, https://refactoring.guru/design-patterns/adapter

23. Error Boundaries, React Docs, https://reactjs.org/docs/error-boundaries.html

24. Window: `error` event, MDN Web Docs, https://developer.mozilla.org/en-US/docs/Web/API/Window/error_event

25. Window: `unhandledrejection` event, MDN Web Docs, https://developer.mozilla.org/en-US/docs/Web/API/Window/unhandledrejection_event

26. “Dealing with Unhandled Exceptions”, by Alexander Zlatkov https://blog.sessionstack.com/how-javascript-works-exceptions-best-practices-for-synchronous-and-asynchronous-environments-39f66b59f012#ecc9

13

Module Integration

Complex applications consist of many parts. The interaction between the parts affects the organization and complexity of the code. The simpler and more obvious the interaction, the easier it is to read and modify the code. In this chapter, we'll discuss how to spot overly convoluted code organization and how to simplify the interaction between application parts.

Coupling and Cohesion

Complicated code can cause fear of changing it. If we feel like "everything will fall apart" or "we'll have to update a lot of code" after making a change, we don't want to touch it because we don't feel safe doing so. Such feelings appear when application modules know too much about each other.

> **By the way** 🧩
>
> By a *module* we mean an isolated part of an application responsible for a specific task that communicates with the outside world or other modules through the public API.

When a module directly affects the code of other modules, its changes will also spread to the other modules. Stopping the propagation of changes in such a code base becomes difficult. Changes related to a particular task begin to affect code that isn't related to it. As a result, making changes in the code becomes scary because anything can break, and we must retest the whole application afterward.

The degree to which one module knows about the structure of other modules is called *coupling*.🔗 The higher the coupling, the harder it is to make changes in isolation to a particular module.

Separation of Concerns

In a well-organized application, working on a task causes changes only in the code related to that task. This principle is known as *Separation of Concerns, SoC*.🔗

SoC helps limit the spread of changes across the code base. If all the code responsible for one task is located in one module, the changes for this task will affect only that module. Everything that isn't related to the task is outside the module and won't affect its code.

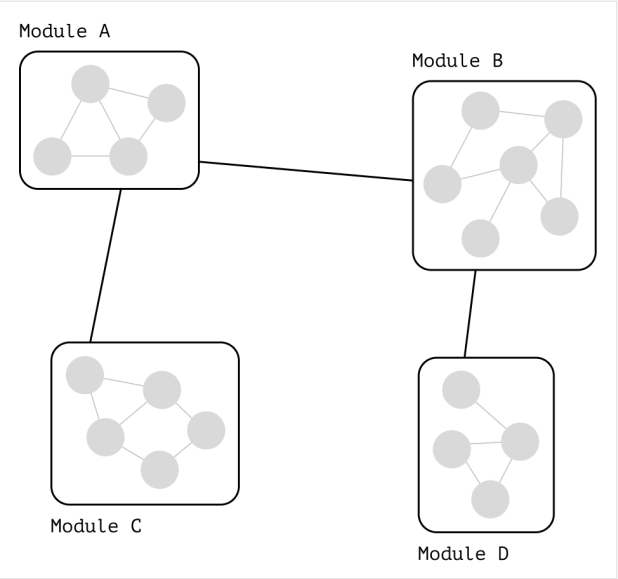
The degree to which the code is related to the task is called *cohesion*.[🔗] The higher the cohesion, the closer a module's code is related to the task it was written for. With higher cohesion, it's easier to find the module related to a particular task in the code base.

Integration Rule

The first and foremost thing we should check when analyzing the interaction of modules during refactoring is the integration rule:

> **Low coupling, high cohesion !**

A program composed according to this rule looks like "islands of cohesive functionality" connected by "bridges" of public APIs, events, or messages:



"Islands" in the code are responsible for related domain tasks and communicate with each other via "bridges" of public APIs

Task Decomposition

Let's look at an example to understand how to find weak module separation during refactoring. The `purchase` module from the snippet below is heavily coupled with the `cart` module. It uses the internal details of the cart object (object structure and `products` field type) to check if it's empty:

TypeScript

```
// purchase.ts

async function makePurchase(user, cart) {
  if (!cart.products.length) throw new Error("Cart is empty!");

  const order = createOrder(user, cart);
  await sendOrder(order);
}
```

The problem with this code is the encapsulation violation. The `purchase` module *doesn't and shouldn't know* how to check if the cart is empty properly.

The details of checking the cart aren't part of the "making a purchase" task. The *fact* that the cart isn't empty is important to it, but it doesn't matter *how* that fact is determined. Implementing the validation is the task of the cart module because it's the one that creates that object and knows how to keep it valid:

TypeScript

```
// cart.ts
// Extract the emptiness check into the `cart` module:
export function isEmpty(cart) {
  return !cart.products.length;
}

// purchase.ts
import { createOrder } from "./order";
import { isEmpty } from "./cart";

async function makePurchase(user, cart) {
  if (isEmpty(cart)) throw new Error("Cart is empty!");

  const order = createOrder(user, cart);
  await sendOrder(order);
}
```

Now, if the internal structure of the cart changes for some reason, the changes will be limited to the `cart` module:

```TypeScript
// cart.ts
type Cart = {
  // Was `products`, became `items`:
  items: ProductList;
};

export function isEmpty(cart) {
  // The only place that requires updates:
  return !cart.items.length;
}

// purchase.ts
async function makePurchase(user, cart) {
  if (isEmpty(cart)) throw new Error("Cart is empty!");
  // Other places in the code stay intact.
  // We've limited the changes propagation.
}
```

Modules that use the `isEmpty` function from the public API will remain unchanged. If modules were to use the cart's internal structure directly, they would all have to be updated when the property changes.

Search for Cohesion

It's often not clear whether a task belongs to a specific module or not. To determine it, we can look at the data the module or function uses.

The "data" is the *input and output parameters*, and the *dependencies and context* that the module uses. The less the data of one module is similar to the data of another, the more likely it's that they relate to different tasks. If, for example, a function often works with data from a neighboring module, it most likely should be part of that module.

> **By the way** 🐓
>
> We may know this problem as the Feature Envy code smell.

Let's imagine we're refactoring a finance management application that can track user expenses. Suppose we see code like this in the module responsible for the budget:

JavaScript

```
// budget.js

// Creates a new budget:
function createBudget(amount, days) {
  const daily = Math.floor(amount / days);
  return { amount, days, daily };
}

// Calculates how much was spent in total:
function totalSpent(history) {
  return history.reduce((tally, record) => tally + record.amount, 0);
}

// Adds a new expense, decreasing the current money amount
// and adding a new spending record in the history:
function addSpending(record, { budget, history }) {
  const newBudget = { ...budget, amount: budget.amount - record.amount };
  const newHistory = [...history, record];

  return {
    budget: newBudget,
    history: newHistory,
  };
}
```

The `budget` module is responsible for data transformations of the *budget*. However, we see functions that don't work only with it:

- The `totalSpent` function doesn't work with the budget at all, it works with the history of expense records;
- The `addSpending` function works with the budget but also uses the expense history.

From the data that these functions work with, we can conclude that they aren't so much about `budget`. For example, `totalSpent` is more related to the history, while `addSpending` is more like an "Add Spending" use case functionality.

Let's try to break up the code by features, separating history and the use case into separate modules:

```JavaScript
// budget.js
// Here's only the code related to the budget:

function createBudget(amount, days) {
  const daily = Math.floor(amount / days);
  return { amount, days, daily };
}

function decreaseBy(budget, record) {
  const updated = budget.amount - record.amount;
  return { ...budget, amount: updated };
}

// history.js
// Here's only the code related to the expense history:

function totalSpent(history) {
  return history.reduce((tally, record) => tally + record.amount, 0);
}

function appendRecord(history, record) {
  return [...history, record];
}

// addSpending.js
// Here's the “Add Spending” use case:
// - decrease the budget amount,
// - add the new history record.

function addSpending(spending, appState) {
  const budget = decreaseBy(state.budget, spending);
  const history = appendRecord(state.history, spending);
  return { budget, history };
}
```

Such strict separation of modules and features may not be necessary for simple applications. But if the application needs to scale and more use cases need to be added, the functionality of different modules will probably need to be composed in various ways. Unclear module separation can lead to implicit dependencies between the modules, which makes it harder to compose and reuse functionality.

For scaling, we should keep an eye on coupling and cohesion. If we're sure we'll be extending and reusing functionality, it's better to divide the code into modules so that there are a minimum of implicit dependencies between them.

Contracts

The public API of a module can be called a *contract*.[🔗] Contracts fixate guarantees of one entity over others: they require certain arguments and obligate functions to return a specific

result. They allow other parts of the program to rely not on the module's entity but only on its "promises" and to base their work on them.

Let's look at an example to see why this is useful. In the code below, we rely on the structure of the `api` module, thereby increasing the coupling:

```TypeScript
// ...
await api.post(api.baseUrl + "/" + api.createUserUrl, { body: user });
// ...
await api.post(api.baseUrl + "/posts/" + api.posts.create, post);
```

The `api` module doesn't explicitly promise how it will work. So when we use it, we *need to know how* it works. But the direct dependence on the `api` module details increases the coupling: if we now change the `api` module, we have to change the code which uses it. High coupling slows down app development.

Instead, the `api` module could declare a *contract*, a set of guarantees describing how it'll work:

```TypeScript
type ApiResponse = {
  state: "OK" | "ERROR";
};

interface ApiClient {
  createUser(user: User): Promise<ApiResponse>;
  createPost(post: Post): Promise<ApiResponse>;
}
```

Then we would implement this contract inside the `api` module, only exposing the public API and not revealing any extra details:

```TypeScript
const client: ApiClient = {
  createUser: (user) =>
    api.post(api.baseUrl + "/" + api.createUserUrl, { body: user }),

  createPost: (post) =>
    api.post(api.baseUrl + "/posts/" + api.posts.create, post),
};
```

And then we would use the `api` module, relying only on its contract:

```TypeScript
// ...
await client.createUser(user);
// ...
await client.createPost(post);
```

> **Clarification**
>
> In a formal definition, contracts are pre- and post-conditions in the form of prescribed verifiable specifications. In practice, I've rarely seen contracts in this form, only in the form of fixed guarantees instead.
>
> Guarantees aren't necessarily a type signature or an interface. They can be sound or written agreements, DTOs, message formats, etc. The important thing is that these agreements *should declare and fixate the behavior* of parts of the system toward each other.

Different modules can fulfill the same "promises." So if we rely on "promises," implementation becomes easier to change, e.g., during testing:

TypeScript

```
// Describe the "contract" of using storage.
// In the interface, we specify which method can be used,
// what it takes as an argument and what it returns as a result:

interface SyncStorage {
  save(value: string): void;
}

// In the argument of `saveToStorage`, specify not a concrete entity,
// but "something that implements the `SyncStorage` interface":

function saveToStorage(value: string, storage: SyncStorage) {
  if (value) storage.save(value);
}

// ...

describe("when given a non-empty value", () => {
  // In tests, we describe the storage mock
  // as "something that implements the `SyncStorage` interface":
  const mock: SyncStorage = { save: jest.fn() };

  it("should save it into the given storage", () => {
    // Then, during the tests, we can replace
    // storage implementation with the mock:
    saveToStorage("Hello World!", mock);
    expect(mock.save).toHaveBeenCalled();
  });
});
```

This way, we even can replace one algorithm or piece of the application with another at the run time:

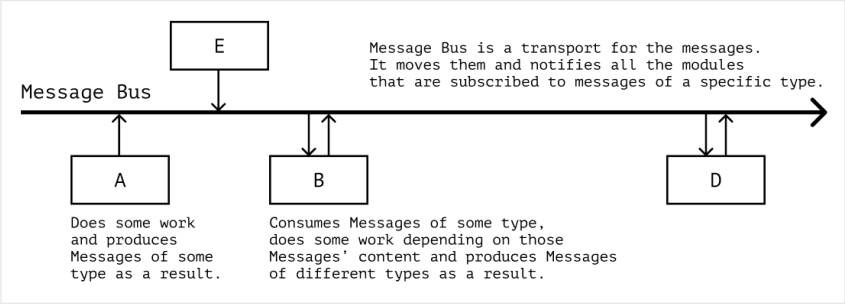
```JavaScript
// Save settings to cookies or local storage
// depending on the user settings:

const storage = preferences.useCookie ? cookieAdapter : localStorageAdapter;
const saveCurrentTheme = () => saveToStorage(THEME, storage);

// While `cookieAdapter` and `localStorageAdapter` both implement `SyncStorage`,
// we can use either of them in the `saveToStorage` function.
```

> **By the way** 🔁
>
> The idea of the module replacement is the basis of the "Strategy" pattern and dependency injection.🔗🔗🔗

Events and Messages

The lower the coupling, the more module interaction resembles sending messages. Server-client communication via REST is an example of such communication.🔗 The client and server know nothing about each other's structure and communicate only by a pre-defined contract—messages of a certain kind with data inside.

Messages can be sent either directly from one module to another via a public API or a special entity—a *message bus*. In the second case, the modules know nothing about each other and are coupled only through the message bus:



Communication comes down to sending and receiving messages from the bus

> **Clarification** 📧
>
> There's a difference between a "message bus," a "message queue," an "event bus," and a "message broker."🔗🔗🔗 However, it isn't critical for this chapter, so I didn't focus on a particular term. I use "message bus" in the text as a general synonym for all those, even though it's not "technically correct."

> **By the way** 📆
>
> Communication via events can be called "perfect communication" between modules because it only couples modules via message structure and sending protocols. However, setting up such communication is often resource-intensive and can be an overhead for small projects.

Sending events and messages is usually associated with microservice architecture, but their benefits can be used in regular applications as well. If the application is large, and it's necessary to build communication between its parts without coupling, a message bus can help to solve this problem.

We can use the "Observer" pattern as a primitive message bus.🔗 In this pattern, modules subscribe to certain kinds of messages and react to them when they appear:

TypeScript

```
// The observer will react to messages:

type Observer = (message: Message) => void;

// The observable allows observers to subscribe to the messages
// and notifies them about all new incoming messages:

type Observable = {
  subscribe: (listener: Observer) => void;
  notifyAll: (message: Message) => void;
};

// The implementation here is an object with two methods
// and a list of listeners:

const listeners = [];
const bus: Observable = {
  subscribe: (listener) => listeners.push(listener),
  notifyAll: (message) => listeners.forEach((listener) => listener(message)),
};

// The observer knows that it'll be given an object
// of type `Message` as an argument—this is the contract.
// From the message kind, it'll know if it should handle the message:

const onUserUpdated = ({ type, payload }) => {
  if (type === "updateUser") {
    // If the type is correct, react to the message.

    const [firstName, lastName] = payload;
    // ...
  }
};

// We can subscribe the observer to messages
// with the `subscribe` method:

bus.subscribe(onUserUpdated);

// When a new message arrives,
// the observable will notify the subscribed observers:

bus.notifyAll({
  type: "updateUser",
  payload: ["Alex", "Bespoyasov"],
});
```

Clarification 🌊

In production, you probably won't need to write your own implementation of "Observer." There are many solutions to work with this pattern, such as RxJS.

Fully decoupled communication isn't always needed but can be useful when *different parts of the system should respond to the same messages*, but we don't want to increase the coupling between those parts.

Dependencies

Speaking of coupling and module integration, it's worth mentioning dependency management. By *dependencies*, we'll mean any code that ours use for simplicity. For example, in the function `randomInt`, we use the method `Math.random` in addition to the two arguments—this is a dependency:

```TypeScript
function randomInt(min, max) {
  return Math.random() * (max - min) + min;
}
```

> **By the way** 👻
>
> The `Math` dependency in the example above is *implicit*, because `Math` is used directly in the function's body and isn't designated as an argument. Such implicit dependencies increase the coupling. If we try to test the `randomInt` function, we must make a global mock of the `Math` object.

We can manage dependencies in different ways. It depends on the paradigm and style of the code. However, it's usually convenient to separate the dependencies which produce effects from the rest. This separation helps to bring the code to the Impureim-styled structure, which we discussed in the chapter on side effects earlier. Let's look at examples of such refactoring in code written in different paradigms.

Object Composition

In object-oriented programming, the unit of composition is the object. Objects can mix data and actions (state and methods), so it's usually more challenging to compose objects than functions. In particular, most design patterns and SOLID principles address the problems of object composition.🔗

> **By the way** 👀
>
> It's worth noting that OOP code *can* be written to avoid these problems by separating data and actions. Although, in functional programming, it's the paradigm itself that pushes this separation, while in OOP, we have to make an effort to keep it in mind.

For example, let's look at the code of a finance management app:

```TypeScript
class BudgetManager {
  constructor(private settings: BudgetSettings, private budget: Budget) {}

  // The main problem with the code is the CQS violation: here, effects are mixe
  // This class simultaneously validates the data and updates the budget value..
  checkIncome(record: Record): MoneyAmount | boolean {
    if (record.createdAt > this.budget.endsAt) return false;

    const saving = record.amount * this.settings.piggyBankFraction;
    this.budget.topUp(record.amount - saving);

    return saving;
  }
}

// ...But this isn't visible on the higher level of the composition.
// We won't be able to tell that there's any effect,
// until we look inside the `BudgetManager' code.
class AddIncomeCommandHandler {
  constructor(private manager: BudgetManager, private piggyBank: PiggyBank) {}

  execute({ record }: AddSpendingCommand) {
    const saving = this.manager.checkIncome(record);

    if (!saving) return false;
    this.piggyBank.add(saving);
  }
}
```

By the way 💉

In the example, I imply that we use *Dependency Injection, DI* through class constructors.🔗 We won't discuss it separately, but I'll leave some links with more info about it.🔗🔗

In the example above, because of the CQS violation, it's not clear to us *how many* effects actually occur when the `execute` method is called. We saw 2, but there's no guarantee that `this.budget.topUp` doesn't change anything other than the `budget` object.

The composition of side effects negates the point of abstraction: the more effects there are, the more overall state we have to keep in mind—it's hard to work with. If side effect composition can be avoided, it's better to avoid it.

Instead, we could extract the data transformations and push the effects to the edges of the application. It would keep effects and logic separate:

TypeScript

```
// Extract validation into a separate entity.
// This class will only handle validation.
class AddIncomeValidator {
  constructor(private budget: Budget) {}

  // If necessary, the `canAddIncome` method can be made completely pure,
  // if we pass the value of `endsAt` as an argument.
  canAddIncome(record: Record) {
    return record.createdAt <= this.budget.endsAt;
  }
}

// At the top level, we separate logic and effects.
// We strive for the Impureim sandwich we talked about earlier:
// - Impure effects for getting data (for example, working with the database on
// - Pure data transformation logic (domain functions, creating entities);
// - Impure effects for saving data (or displaying it on the screen).
class AddIncomeCommandHandler {
  constructor(
    // The same technique will allow us to see the problems with coupling.
    // If the class accumulates too many dependencies,
    // we should probably think about improving its design.
    private settings: BudgetSettings,
    private validator: AddIncomeValidator,
    private budget: Budget,
    private piggyBank: PiggyBank
  ) {}

  execute({ record }: AddSpendingCommand) {
    // Validation:
    if (!this.validator.validate(record)) return false;

    // Pure(-ish because of injected settings) logic.
    // It can be extracted into a separate module,
    // but I decided to keep it here for clarity:
    const saving = record.amount * this.settings.piggyBankFraction;
    const income = record.amount - saving;

    // Effects of saving the data:
    this.budget.topUp(income);
    this.piggyBank.add(saving);
  }
}
```

Separate Data and Behavior

The next step could be to separate data from behavior. The `budget` and `piggyBank` objects would become "data containers"—*entities* in DDD terms,[🔗][🔗] and data transformation would be handled by "services":

```TypeScript
class AddIncomeCommandHandler {
  constructor(
    private settings: BudgetSettings,
    private validator: AddIncomeValidator,
    private budgetRepository: BudgetUpdater,
    private piggyBankRepository: PiggyBankUpdater
  ) {}

  // The `budget` and `piggyBank` objects now contain no behavior, only data:
  execute({ record, budget, piggyBank }: AddSpendingCommand) {
    if (!this.validator.validate(record, budget)) return false;

    // Data transformation, business logic:
    const saving = record.amount * this.settings.piggyBankFraction;
    const income = record.amount - saving;

    // Updated data objects:
    const newBudget = new Budget({ ...budget, income });
    const newPiggyBank = new PiggyBank({ ...piggyBank, saving });

    // “Services” for effects with saving the data:
    this.budgetRepository.update(newBudget);
    this.piggyBankRepository.update(newPiggyBank);
  }
}
```

Separating data and behavior helps to separate code that changes quickly (behavior) from code that changes slowly (data). In this way, changes *take fewer files*, and this limits their spread across the codebase.

Interface Segregation Principle

When describing behavior, we can use CQS, as an “integration linter.” For example, if we describe a service as a set of commands, a type signature difference can indicate a CQS violation:

```TypeScript
interface BudgetUpdater {
  updateBalance(balance: MoneyAmount): void;
  recalculateDuration(date: TimeStamp): void;

  // Oops! This method returns something.
  // It's a query, not a command!
  currentBalance(): MoneyAmount;
}
```

Then we can apply the *Interface Segregation Principle, ISP* and decompose the problem:

```TypeScript
// Extract the query into a separate “reading data service”:
interface BudgetSource {
  currentBalance(): MoneyAmount;
}

// While the older service only contains
// commands for changing the data now:
interface BudgetUpdater {
  updateBalance(balance: MoneyAmount): void;
  recalculateDuration(date: TimeStamp): void;
}
```

> **By the way** 👓
>
> The separation of data reading and writing services is one of the ways of applying the ISP. It depends on the task whether to separate the services at the class level but separating them at the interface level helps to express the difference in their intent.

Functional Composition

In projects with “slightly more functional” code, we also can find “dependency injection” made by partial application of functions. The benefit of this “injection” is that it makes *implicit* dependencies *explicit*.

For example, take a look at the `listingQuery` function. It outputs a list of Markdown files from a specified folder:

```TypeScript
const listingQuery = (query) => {
  return fs
    .readdirSync(query)
    .filter((fileName) => fileName.endsWith(".md"))
    .map((fileName) => fileName.replace(".md", ""));
};

const projectList = listingQuery("projects");
```

It implicitly depends on the `fs` module, which gives access to the filesystem. In general, this isn't bad, but such a function is inconvenient to test.[🔗] The tests would require a global mock for `fs`.

With the partial application, we can create a “factory function.” It would take `dependencies` as an argument and return `listingQuery` function as a result:

TypeScript

```
/**
 * 1. Inject the `system` “service”;
 * 2. Pass the actual arguments;
 * 3. Use the `system` “service”;
 *    to get the desired effects.
 */
const createListingQuery =
  ({ system }) =>
  (query) =>
    system
      .readdirSync(query)
      .filter((fileName) => fileName.endsWith(".mdx"))
      .map((fileName) => fileName.replace(".mdx", ""));

// When using it, we would first create the function
// with “injected” `system` service:
const listingQuery = createListingQuery({ system: fs });

// ...And then would use that function:
const projectList = listingQuery("projects");
```

This approach isn't very "functional," but it's okay to work with if we don't have trouble "injecting" dependencies for each such factory, and using them doesn't cause problems with the shared state and its effects.

Dependency Rejection

In "more hardcore" functional programming, changing state and producing side effects isn't common. The concept of "dependencies" in its usual sense doesn't really fit there. Instead of "dependencies" that "expose methods" and effects, FP offers the functional core in an imperative shell.

> **By the way** 🥷
>
> I like what Mark Seemann calls this concept—*dependency rejection*.🔗 It's like we move away from the concept of dependencies in general and try to solve the problem differently.

In this approach, all the work with the state is pushed to the edges of the application. It means that we can read and write (or render) data only at the beginning and end of the module. All work between these points is based on pure data transformations.

That is, we first get all the data we need from the "impure" world, pass it as arguments to the chain of transformations, and then save the result:

```TypeScript
// Extract the function with
// the “File names to Post names” transformation:
const listingQuery = (fileNames) =>
  // Notice, there's no dependency on `fs`,
  // we work only with the pure data here:
  fileNames
    .filter((fileName) => fileName.endsWith(".mdx"))
    .map((fileName) => fileName.replace(".mdx", ""));

// “Composition” now is a separate function:
// - It first calls the effect to read the data.
// - Then, it runs it through a chain of transformations.
// - At the end, it returns the result or calls the effect to save the data.
const listingQueryComposition = (query) => {
  // Inside, we follow the Impureim sandwich structure.
  //
  // 1. Read data effect.
  //   (Because of the effects, the composition context
  //    and the composition function itself are considered “impure.”)
  const files = fs.readdirSync(query);

  // 2. Data transformation logic in the form of a sequence of pure functions.
  return listingQuery(files);

  // 3. Write (save / render) data effect.
  //   (Here, we return the result from the function.
  //    But if we weren't, we would call an effect to saving the result.)
};
```

At first glance, it seems to become worse: tests for the function now need global mocks, and with “dependency injection,” we could just replace the services with stubs. However, in this concept, unit tests should test only the functional core—the `listingQuery` function.

The `listingQuery` function is pure, so the tests for it wouldn't require any mocks at all. As for composition, in simple cases, we may skip testing it since it just “gathers” the functionality together. In more complex cases, we should use integration or E2E tests.

When using integration tests, such a composition forces the “Ports-Adapters” architecture, which also helps reduce the number of mocks, making the tests less “fragile.”

> **By the way** 🔌
>
> We'll talk more about the Ports-Adapter architecture in a separate chapter later.

We'll still need the mocks, for example, for testing the adapters. But in this case, we'd write only one adapter for each service, meaning we'd only need to mock that service once.

```TypeScript
interface System {
  readDirectory(directory: DirectoryPath): List<FileName>;
}

const createAdapter = (service): System => ({
  readDirectory(directory) {
    /*...*/
  },
});

//  test.js

describe("when asked to read the given directory", () => {
  it("should trigger the `readdirSync` method on the service", () => {
    const mock = { readdirSync: jest.fn() };
    const adapter = createAdapter(mock);

    adapter.readDirectory("testdir");
    expect(mock.readdirSync).toHaveBeenCalledWith("testdir");
  });
});
```

In the case of "dependency injection," we'd have to mock the service *for each function* where the service is used.

Other Dependency Management Options

Dependency rejection isn't always the best option. If a project is heavy on I/O operations, then "dependency injection" through the partial application is probably a better fit.

However, even if we're going to "inject" services, it'll be much easier if we separate logic and effects first. So in functional code, separating logic and effects is the first refactoring worth doing.

> **By the way** 🦄
>
> There are also various advanced dependency management techniques that are used in FP.[🔗][🔗] Although in JavaScript, I've only seen them being used once.
>
> A good overview of all the techniques for working with dependencies in FP is described in the article series "Six approaches to dependency injection" by Scott Wlaschin.[🔗] I highly recommend at least the first three articles.

Integrity and Consistency

In stateful applications, we should also keep an eye on the *integrity and consistency* of the data. These properties ensure that the user sees and works with the application in a valid state.

> **Read more** 📚
>
> Scott Wlaschin has written about this in detail in "Domain Modeling Made Functional" in the section about aggregates and data consistency.

Aggregates

Data consistency is easier to achieve if we use immutable data structures and ensure adequate encapsulation.

Consider an example. Imagine that in the code of an online store, the shopping cart must always have the correct `total`:

```TypeScript
// cart.ts
type Cart = {
  items: ProductList;
  total: MoneyAmount;
};

function totalPrice(products: ProductList) {
  return products.reduce(
    (tally, { price, count }) => tally + price * amount,
    0
  );
}

function createCart(products: productList): Cart {
  return {
    items: products,
    total: totalPrice(products),
  };
}
```

By improperly changing the cart object, we can make the data inconsistent. Suppose some extraneous code added a new product to the list:

```TypeScript
// The outside code doesn't know that after adding a product
// it must also recalculate the total price:
userCart.products.push(appleJuice);

// Now the `cart.total` field shows an incorrect value,
// because it wasn't recalculated after adding the product.
```

A piece of data that must be updated "as a whole" is an *aggregate*. Data immutability can help keep aggregates consistent. It forces the updating of an aggregate to happen *starting at the aggregate root*:

```TypeScript
// cart.ts
// ...

function addProduct(cart: Cart, product: Product): Cart {
  const products = [...cart.products, product];
  const total = totalPrice(products);
  return { products, total };
}

// ...

addProduct(userCart, appleJuice);
```

The `addProduct` function guarantees consistency because it knows what data to update and from where to update it to keep it valid. Correct updates are its area of responsibility. Aggregate is its area of influence.

Input Prevalidation

Aggregates and immutability help keep data valid and consistent *inside* a part of the application. But to *prevent* invalid data from getting there, we need prevalidation.

> **Read more** 👀
>
> In DDD terms, we'd call such an application part a bounded context. We discussed this topic in more detail earlier in the chapter on the functional pipeline.

With prevalidation, in the `Cart` component, instead of ad-hoc checks on the required fields:

```JavaScript
function Cart({ items }) {
  return (
    !!items && (
      <ul>
        {items.map((item) =>
          item && item.product ? (
            <li key={item.id}>
              {item.product?.name ?? "—"}: {item.product?.price} ×{" "}
              {item.product?.count ?? 0}
            </li>
          ) : null
        )}
      </ul>
    )
  );
}
```

…We would first validate the data and handle potential errors:

```JavaScript
function hasCorruptedItem(item) {
  return !item || !item.product;
}

function validateCart(cart) {
  if (!cart || !cart.items) return Result.failure("EMPTY_CART");
  if (cart.items.some(hasCorruptedItem))
    return Result.failure("CORRUPTED_ITEM");

  // There's also an option to use panics instead of `Result`.
  // We talked more about error handling in the previous chapter.

  // An alternative to errors could be a “fallback default” cart object as a res
  // but it seems to me that a violation of the API contract is a sufficient rea

  return cart;
}
```

...And inside the component, we'd use *already valid and tested* data:

```JavaScript
// ...

function Cart({ items }) {
  return (
    <ul>
      {items.map(({ id, product }) => (
        <li key={id}>
          {product.name}: {product.price} × {product.count}
        </li>
      ))}
    </ul>
  );
}
```

---

1. Coupling, Wikipedia, https://en.wikipedia.org/wiki/Coupling_(computer_programming)
2. Separation of Concerns, Wikipedia, https://en.wikipedia.org/wiki/Separation_of_concerns
3. Cohesion, Wikipedia, https://en.wikipedia.org/wiki/Cohesion_(computer_science)
4. Feature Envy, Refactoring Guru, https://refactoring.guru/smells/feature-envy
5. “Design By Contract”, c2.com, https://wiki.c2.com/?DesignByContract
6. Strategy Pattern, Refactoring Guru, https://refactoring.guru/design-patterns/strategy
7. “Inversion of Control Containers and the Dependency Injection pattern” by Martin Fowler, https://martinfowler.com/articles/injection.html
8. “Dependency Injection with TypeScript in Practice” by Alex Bespoyasov, https://bespoyasov.me/blog/di-ts-in-practice/
9. REpresentational State Transfer, REST, https://restfulapi.net
10. Message Broker, Microsoft Docs, https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648849(v=pandp.10)
11. Message Bus, Microsoft Docs, https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff647328(v=pandp.10)
12. Message Queue, Wikipedia, https://en.wikipedia.org/wiki/Message_queue
13. Observer Pattern, Refactoring Guru, https://refactoring.guru/design-patterns/observer

14. RxJS, Reactive Extensions Library for JavaScript, https://rxjs.dev

15. The Principles of OOD, Robert C. Martin, http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

16. "Dependency Injection in .NET" by Mark Seemann, https://www.goodreads.com/book/show/9407722-dependency-injection-in-net

17. "Domain-Driven Design" by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design

18. "Evans Classification" by Martin Fowler, https://martinfowler.com/bliki/EvansClassification.html

19. "Functional architecture: The pits of success" by Mark Seemann, https://youtu.be/US8QG9I1XW0

20. "Command-Query Separation" by Martin Fowler, https://martinfowler.com/bliki/CommandQuerySeparation.html

21. "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing

22. "Dependency Rejection" by Mark Seemann, https://blog.ploeh.dk/2017/02/02/dependency-rejection/

23. "Six approaches to dependency injection" by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/dependencies/

24. "Dependency Injection Using the Reader Monad" by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/dependencies-3/

25. "Dependency Interpretation" by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/dependencies-4/

26. "Domain Modeling Made Functional" by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional

14

Generics, Inheritance, and Composition

During refactoring, we may discover "hidden patterns" in different parts of code. These can be similar algorithms or function details that can be generalized.

Generalization can both simplify the code and make it more complex. In this chapter, we'll talk about how to understand when it's worth generalizing similar parts of code and when it isn't. We'll also discuss hierarchies and inheritance and explain why composition is preferable to inheritance.

Generic Algorithms

A generic algorithm is an extension of the ideas of abstraction and code deduplication.

If several functions work "by the same scheme," we can formalize this "work scheme" as a set of operations. These operations would describe the function in an abstract, "generic" way without referring to specific variables in the code.

For example, instead of manually going through the elements of an array:

JavaScript

```
const items = [1, 2, 3, 4, 5];

for (const i of items) {
  console.log(i);
}
// 1 2 3 4 5

for (const i of items) {
  console.log(i - 1);
}
// 0 1 2 3 4
```

...We can extract the "work scheme." Formalized, it'd sound like this: "iterate over an array, apply a specified function to each element." In the example below, the `forEach` method encapsulates this "scheme":

```JavaScript
const printNumber = (i) => console.log(i);
const printNumberMinusOne = (i) => console.log(i - 1);

// Iterate over `items`, apply the `printNumber` function to each element:
items.forEach(printNumber);
// 1 2 3 4 5

// Iterate over `items`, apply the `printNumberMinusOne` function to each elemer
items.forEach(printNumberMinusOne);
// 0 1 2 3 4
```

The generic algorithm doesn't refer to specific variables and functions. Instead, it expects them as parameters:

> Iterate over a *specified* array; apply a *specified* function to each element ”

Such an algorithm relies on arguments' *properties*. An array is iterable; a function is callable and accepts an array element as an argument. The combination of these properties is the algorithm *contract*. The basis of generalized programming is identifying such "work schemes" and defining their contracts.

To understand how generic algorithms can be helpful and how to identify them, let's look at a financial management application. In the code below, there are two functions for counting expenses and incomes throughout the history of records:

```TypeScript
// The type describes the history of spending and income in the application:
type RecordHistory = List<Entry>;

// The record in the history contains its kind, creation date, and amount:
type EntryKind = "Spend" | "Income";
type Entry = {
  type: EntryKind;
  created: TimeStamp;
  amount: MoneyAmount;
};

// Calculates how much is spent in total:
function calculateTotalSpent(history: RecordHistory) {
  return history.reduce(
    (total, { type, amount }) => total + (type === "Spend" ? amount : 0),
    0
  );
}

// Calculates how much is added in total:
function calculateTotalAdded(history: RecordHistory) {
  let total = 0;
  for (const { type, amount } of history) {
    if (type === "Income") total += amount;
  }
  return total;
}
```

Let's say we now want to calculate a total for a certain period, for example, today. When adding it, we should check if the new function has any common features with the existing ones:

```TypeScript
// In the `calculateSpentToday` function, you can see the “work scheme”,
// similar to the algorithms in `calculateTotalSpent` and `calculateTotalAdded`:
// - Receive a history array
// - Filter the necessary records
// - Calculate the sum

function calculateSpentToday(history: RecordHistory) {
  return history.reduce(
    (total, { type, created }) =>
      total +
      (type === "Spend" && created >= today && created < tomorrow ? amount : 0),
    0
  );
}
```

Note that each function boils down to two tasks: filter the history and summarize the `amount` field in the filtered records. We can extract these tasks and generalize them:

```TypeScript
type HistorySegment = List<Entry>;
type EntryPredicate = (record: Entry) => boolean;

// Filtering can now be used separately.
// The `keepOnly` function can filter the history of records
// by _any_ criterion that implements the `EntryPredicate` type:

const keepOnly = (
  history: RecordHistory,
  criterion: EntryPredicate
): HistorySegment => history.filter(criterion);

// Summation can now also be used separately.
// We can pass _any_ fragment of history to the `totalOf` function,
// and it will calculate the amount of spending or income in it:

const totalOf = (history: HistorySegment): MoneyAmount =>
  history.reduce((total, { amount }) => total + amount);
```

Then we can combine the original algorithms from these two functions—this will be an implementation of the generalized algorithm. All that's *different* is going to be used as *parameters* of this generalized algorithm:

```TypeScript
// Only the filter criteria differ; everything else is the same:
const isIncome: EntryPredicate = ({ type }) => type === "Income";
const isSpend: EntryPredicate = ({ type }) => type === "Spend";
const madeToday: EntryPredicate = ({ created }) =>
  created >= today && created < tomorrow;

// The filtering criterion is passed as a “parameter” for `keepOnly`:
const added = keepOnly(history, isIncome);
const spent = keepOnly(history, isSpend);

// The sum can then be calculated for any fragment of history:
totalOf(spent);
totalOf(added);
totalOf(keepOnly(spent, madeToday));
```

Generalization is useful when we're entirely sure of the "work scheme." If several pieces of code are the same or only slightly different, generalization can help reduce duplication.

But generalization can also make the code more complicated. If we suspect that the "scheme" may change in the future, it's probably too early to generalize. The heuristics here are the same as when dealing with code duplication. Until we're sure of our code knowledge, it's better to hold off on generalizing.

> **By the way** 💡
>
> We have discussed the heuristics of working with code duplication and the lack of information about the code base in more detail in one of the previous chapters.

Generic Types

Sometimes we can find similar details not only in algorithms but in data types. A *type* is a set of values with some specific properties.🔗🔗 Similar types can be combined into a generic one, but we should follow the rule:

> **Generalize only when we're sure there won't be any exceptions ❗**

Data is harder to change than code. Careless type generalization can weaken typing and force us to add unnecessary checks to the code. Let's go back to the `Entry` type from the snippet above for an example:

```TypeScript
type EntryKind = "Spend" | "Income";
type Entry = {
  type: EntryKind;
  created: TimeStamp;
  amount: MoneyAmount;
};
```

If we know that only records with these properties can be found in the history, then the type describes the domain well. But if there's a possibility that there may be other records in history, then we may have problems with this type.

For example, let's say the history can show user comments that don't contain `amount` but contain text:

```TypeScript
type EntryKind = "Spend" | "Income" | "Comment";
type Entry = {
  type: EntryKind;
  created: TimeStamp;

  // The generalization weakens the `amount` field,
  // and with it the entire `Entry` type:
  amount?: MoneyAmount;

  // Here, a new field appears.
  // It "may or may not exist" —
  // this also weakens the type.
  content?: TextContent;
};
```

There's a contradiction in the `Entry` type. Its `amount` and `content` fields are optional, so the type "allows" entries without amount and text. But this reflects the domain incorrectly: a comment *must* contain text, and spending records and incomes *must* have amounts. Without this condition, the data in the history of records is invalid.

The problem is that the `Entry` type with optional fields tries to mix *different* entities and data states. We can check this by creating a React component to display the sum of the record from the history on the screen:

```
type RecordProps = {
  record: Entry;
};

// We pass an `Entry` type record to the component,
// to output the amount of spending or income:
const Record = ({ record }: RecordProps) => {
  // But internally, we have to filter this data.
  // If the record is a comment, the render has to be skipped,
  // because comments don't contain amounts:
  if (record.type === "Comment") return null;

  const sign = isIncome(record) ? "+" : "-";

  // And here, we also have to tell the compiler,
  // that `amount` is "definitely there, we checked!"
  return `${sign} ${record.amount!}`;
};
```

The comment rendering in this component doesn't really fit. To fix this problem, we first need to understand what we know about the domain:

- Can there only be comments, or can other text messages appear?
- Can there be new record types that have the `amount` field?
- Can there be new record types that will have completely different fields?

We can't always answer these questions. When we lack knowledge of the domain or program requirements, it's too early to generalize the types. In such cases, it is preferable to *compose* types instead of generalizing:

TypeScript

```
// We split the type into several.
// Unnecessary fields are gone,
// data states are separated more clearly,
// and static typing is more helpful this way.
type Spend = { type: "Spend"; created: TimeStamp; amount: MoneyAmount };
type Income = { type: "Income"; created: TimeStamp; amount: MoneyAmount };
type Comment = { type: "Comment"; created: TimeStamp; content: TextContent };

// Yup, we “wrote more code,” but in the early stages of the project,
// it's more important for us to understand the domain
// and grasp the app essence and the relationships between entities.

// Atomic types like the ones above are more flexible,
// because we can compose them by different properties
// that are important to us in a particular situation.

// For example, in the `FinanceEntry` type below
// we compose only the records that _contain an amount_.
// We can walk through the lists of such records with the `amount` adder:
type FinanceEntry = Spend | Income;

// In the type `MessageEntry` we compose records _containing text_.
// Right now, there's only one such type, so `MessageEntry` is just a type alias
// but if there are more records with text, we can extend this type further.
type MessageEntry = Comment;

// The most general type `Entry` is represented as a choice of _all_ possible op
// With these records, we can do only what we can do with _any_ record:
type Entry = FinanceEntry | MessageEntry;

// For example, `Entry` can be sorted by the `created` date,
// since the `created` field is guaranteed for all records:
const sortByDate = (a: Entry, b: Entry) => a.created - b.created;
```

After refactoring, the `Record` component no longer needs extra checks:

TypeScript

```
// In the props, we don't specify the `Entry` type but `FinanceEntry`.
// By definition, it has the `amount` field, so we don't need
// any additional checks during the rendering of the component anymore.

type RecordProps = {
  record: FinanceEntry;
};

const Record = ({ record }: RecordProps) => {
  const sign = isIncome(record) ? "+" : "−";
  return `${sign} ${record.amount}`;
};
```

> **By the way** 👀
>
> Even if we consider TypeScript's "bad typing," this is still useful. If we design an app to avoid premature generalizations, we'll save the code from unnecessary checks.
>
> Of course, there's no guarantee that we won't get the wrong data type in the component's props in production. But with proper alert monitoring and error handling, we can quickly find and fix it.

The basic idea of composition is not to generalize *early*. Type composition is easier to extend as application requirements become more complex. For example, assume we needed to add overdraft records and system text messages. Let's add the `Overdraft` and `Warning` types:

TypeScript

```
type Spend = { type: "Spend"; created: TimeStamp; amount: MoneyAmount };
type Income = { type: "Income"; created: TimeStamp; amount: MoneyAmount };
type Overdraft = { type: "Overdraft"; created: TimeStamp; amount: MoneyAmount };

type Comment = { type: "Comment"; created: TimeStamp; content: TextContent };
type Warning = { type: "Warning"; created: TimeStamp; content: TextContent }; //

// In the unions, we only need to add a new variant
// to each place that needs to be expanded:
type FinanceEntry = Spend | Income | Overdraft;
type MessageEntry = Comment | Warning;
type Entry = FinanceEntry | MessageEntry;

// If necessary, we can recompose the types from scratch,
// to compose them by other properties.
```

Knowing enough about the domain allows us to see patterns and generalize types. (We know enough when the code of the entities has stopped changing.) But this isn't a necessary step:

TypeScript

```
// If we're sure of the type structure now,
// we can generalize the types.

type FinanceEntry = {
  type: "Spend" | "Income" | "Overdraft";
  created: TimeStamp;
  amount: MoneyAmount;
};

type MessageEntry = {
  type: "Comment" | "Warning";
  created: TimeStamp;
  content: TextContent;
};
```

Inheritance and Composition

In the example above, it's easy to start using *generic types* for describing `Entry`. The motivation might be that the generic type would allow determining the record kind "on the fly":

```TypeScript
type FinanceEntryKind = "Spend" | "Income" | "Overdraft";
type MessageEntryKind = "Comment" | "Warning";
type EntryKind = FinanceEntryKind | MessageEntryKind;

// Type fields and behavior will then be defined
// by the type-argument `TKind`:

type Entry<TKind extends EntryKind> = {
  type: TKind;
  created: TimeStamp;
  amount: TKind extends FinanceEntry ? MoneyAmount : never;
  content: TKind extends MessageEntryKind ? TextContent : never;
};

type Income = Entry<"Income">;
type Comment = Entry<"Comment">;
```

But it's best not to rush with generics, either. Generics are like "blueprints" for types. To use them, we have to make sure that the structure of the type doesn't change.

> **Be careful** 🚧
>
> This section is one big IMHO. I could be totally wrong. Stay alert, and don't believe me without question.

Generics are fine for cases where we know the type structure but don't know its details. For example, in the code above, we had a type `EntryPredicate`:

```TypeScript
type EntryPredicate = (record: Entry) => boolean;
```

A predicate is *by definition* a function that returns a boolean value, so we know the structure exactly:

```TypeScript
type SomethingPredicate = (x: Something) => boolean;
```

If, for some reason, we need to create *multiple* predicates with different arguments, which we *don't know in advance*, the generic would describe such a "blueprint" perfectly:

```TypeScript
// Predicate with an “abstract parameter”:
type Predicate<T> = (x: T) => boolean;

// Predicates with “specific parameters”:
type EntryPredicate = Predicate<Entry>;
type HistoryPredicate = Predicate<History>;
type WhateverPredicate = Predicate<Whatever>;
```

Here we're sure that the *structure of the type is known and won't change*. We can pass any type argument, and it won't affect the type structure or its behavior. In `Entry<TKind>`, we can't give such a guarantee, at least in the early stages of design.

Inheritance

Since we're talking about composition and entities' "blueprints", let's talk about OOP and inheritance. In OOP-style projects, it's better to avoid deep inheritance hierarchies:

```TypeScript
class Task {
  public start(): void {}
}

class AdvancedTask extends Task {
  public configure(settings: TaskSettings): void {}
}

class UserTask extends AdvancedTask {
  public definedBy: User;
}

// Class `UserTask` is the 3rd in the inheritance chain:
// Task -> AdvancedTask -> UserTask
```

Deep hierarchies are brittle. They claim to know the domain perfectly, but the model cannot describe the world *perfectly*, so sooner or later, the hierarchy will divert from the real world. If we haven't provided some functionality in the base class, the broken hierarchy will force us to add it to it or redefine the inheritance chain.

Instead of inheritance, it's preferable to use composition. In OOP code, composition usually means the implementation of interfaces:

TypeScript

```
// Declare several interfaces,
// each of them describes a cohesive set of features:

interface SimpleTask {
  start(): void;
}

interface Configurable {
  configure<TSettings>(settings: TSettings): void;
}

interface UserDefined {
  definedBy: User;
}

// A class can implement several interfaces,
// bypassing unnecessary inheritance steps:

class UserTask implements SimpleTask, Configurable, UserDefined {}

// Then, when adding a new feature, it'll be enough to
// extend the list of interfaces with the new one and implement it:

interface Cancellable {
  stop(): void;
}

class UserTask
  implements SimpleTask, Restartable, Configurable, UserDefined, Cancellable {}

// In the case of inheritance, we would have to
// change one of the base classes
// or change the inheritance structure.
```

> **However** 📝
>
> We're not talking about the abstract class implementation here.🔗 It, on the contrary, can be helpful. But even with abstract classes, avoiding hierarchies deeper than 1-2 levels is better.

In JavaScript, however, there's one use case where a deeper inheritance can be useful. If in a project, we use panics to handle errors but want to avoid verbose checks with `instanceof` for *each* error type, we can use "error layer" hierarchies:

```TypeScript
// Application error base class:
class AppError extends Error {}

// Error classes divided by “app layers”:
// API layer and its subclasses per API error.
class ApiError extends AppError {}
class NotFound extends ApiError {}
class BadRequest extends ApiError {}

// Validation layer and subclasses per validation error.
class ValidationError extends AppError {}
class InvalidUserDto extends ValidationError {}
class MissingPostData extends ValidationError {}

// Then, when handling we can reduce the number of checks,
// by replacing the checks for every possible type:
if (e instanceof NotFound) {
} else if (e instanceof BadRequest) {
} else if (e instanceof InvalidUserDto) {
} else if (e instanceof MissingPostData) {
}

// ...With checks for the app layer errors:
if (e instanceof ApiError) {
} else if (e instanceof ValidationError) {
}
```

This way, we can reduce the number of `instanceof` checks, but it'll only work if the error handling is the same for *all* errors from the same layer.

The Liskov Substitution Principle

Previously, we gave an example with a component that had excessive checks:

```TypeScript
type Entry = Spend | Income | Comment;
type RecordProps = { record: Entry };

const Record = ({ record }: RecordProps) => {
  if (record.type === "Comment") return null; // Filters out comments.

  const sign = isIncome(record) ? "+" : "-";
  return `${sign} ${record.amount}`;
};
```

When we see such conditions, we should check if the Liskov substitution principle is violated.[🔗][🔗] This principle in Martin's formulation sounds like this:

> Functions that use references to the base type must be able to use its subtypes without knowing it[🔗]

In practice, the `Entry` type describes *any* history record. It doesn't make a difference between a comment, spending, or an income. Therefore, we can only pass it to a function that is *ready to work with any record*.

That is, if a function works only with expenses or incomes, but *not comments*, then the `Entry` type cannot be passed to such a function. It can't because the function may want to read the `amount` field on the record, but the comment doesn't have this field. So to prevent errors, we have to *filter out* the comments before using the function.

Conversely, if the function can work with any record and only uses the `created` field, for example, it can be passed arguments of the `Entry` type since the `created` field exists on all the `Entry` "subtypes."

> **Read more** 🔬
>
> For a better understanding of the LSP, it's worth diving into the concept of variance,[1][2][3] but that's a big separate topic. We'll skip it now, but I'll leave some references to the subject in the source list.

To avoid violating the substitution principle, we have to pass the "smallest common type" to the `Record` component:

```TypeScript
type FinanceEntry = Income | Spend;
type RecordProps = { record: FinanceEntry };

const Record = ({ record }: RecordProps) => {
  // No additional checks are required.
  // The `FinanceEntry` type for sure contains
  // everything the function can work with.

  const sign = isIncome(record) ? "+" : "-";
  return `${sign} ${record.amount}`;
};
```

The substitution principle helps to see what types can be composed together. We can use it as an "integration linter," highlighting premature generalizations and incorrect abstractions.

It also helps to find places for *correct* generalizations. For example, when we split the problem of `total` calculation into filtering and summarizing, we extracted a function that can filter not only by type but generally by whatever:

```TypeScript
type EntryPredicate = (record: Entry) => boolean;
type HistorySegment = List<Entry>;

type HistoryFilter = (
  history: RecordHistory,
  criterion: EntryPredicate
) => HistorySegment;

// As filter criteria, we can use not only these functions:
const isIncome = ({ type }) => type === "Income";
const isSpend = ({ type }) => type === "Spend";

// But also these:
const beforeToday = ({ created }) => created < today;
const madeToday = ({ created }) => created >= today && created < tomorrow;

// And these:
const addedToday = (record) => isIncome(record) && madeToday(record);
const spentBeforeToday = (record) => isSpend(record) && beforeToday(record);
```

In `HistoryFilter` we can now pass *any* implementation of type `EntryPredicate`. This composition flexibility is the main benefit of the substitution principle. We can search for such places using the LSP and refactor them in the code first.

> **Simplification 🚧**
>
> We won't discuss how the substitution principle relates to pre- and postconditions and how those should behave. But I'll leave some links that I can recommend you to read about this topic.🔗

---

1. "Functional Design Patterns" by Scott Wlaschin, https://youtu.be/srQt1NAHYC0

2. Types and Typeclasses, Learn You Haskell, http://learnyouahaskell.com/types-and-typeclasses

3. Generics in TypeScript, TypeScript Docs https://www.typescriptlang.org/docs/handbook/2/generics.html

4. Abstract Classes and Members, TypeScript Handbook, https://www.typescriptlang.org/docs/handbook/2/classes.html#abstract-classes-and-members

5. "A behavioral notion of subtyping" by Barbara H. Liskov, Jeannette M. Wing, https://dl.acm.org/doi/10.1145/197320.197383

6. The Principles of OOD, Robert C. Martin, http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

7. The Liskov Substitution Principle, https://web.archive.org/web/20151128004108/http://www.objectmentor.com/resources/articles/lsp.pdf

8. Covariance and Contravariance, Wikipedia, https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)

9. "Why variance matters" by Ted Kaminski https://www.tedinski.com/2018/06/26/variance.html

10. "The Ins and Outs of Generic Variance in Kotlin" by Dave Leeds, https://typealias.com/guides/ins-and-outs-of-generic-variance/

11. "Design By Contract", c2.com, https://wiki.c2.com/?DesignByContract

15

App Architecture

Poor architecture can make successful refactoring difficult. Tight coupling between code parts and uncontrolled dependencies can prevent us from finding "seams" in the code and isolating the changes. Non-obvious interactions between the modules can make the code confusing and difficult to read.

In this chapter, we'll talk about simplifying refactoring when dealing with a big chunk of an application. We'll discuss the notion of architecture, its goals, and its benefits. We'll talk about the application domain model and its interaction with the outside world. We'll also discuss the difference between business logic and UI logic.

> **Caution** 🚧
>
> Most of my experience comes from developing user applications with rich domain models. So I'll talk about such an average application.
>
> The development of other apps may differ; hence the approaches to the architecture may vary. Even in user applications, enough edge cases may force us to write the code differently. Please, keep this in mind when reading the chapter.

Not About File Structure

Architecture is the interaction of system parts, the top level of application detail, and the identification and prioritization of app requirements.🔗🔗

There is no one-size-fits-all solution in application design because each project's requirements are unique. Two similar applications can be designed differently because of their constraints and requirements.

> **Clarification** 🔬
>
> By requirements, we'll mean functional, system, business, and other requirements.🔗🔗🔗
>
> We won't investigate all types of requirements, how to collect, analyze, and prioritize them—that's a separate big topic. Instead, I'll leave links to books and articles describing how and why to do it.🔗🔗🔗🔗🔗

The techniques in this chapter aren't a set of mandatory practices. Instead, it's a palette from which we can choose the most suitable tool for a particular task. There can be nothing "mandatory" in architecture; each project is a unique set of constraints and compromises, the balance of which depends on many factors.

We emphasize the notion of requirements to focus on the idea:

> **App design doesn't equal project file structure !**

The file structure is just a way to explore the code and open files in the IDE conveniently. Much more important is what requirements are critical for the app and how to make sure that the code organization doesn't interfere with the development.

The requirements give us an understanding of how the system must behave under different conditions. This understanding hints at the details of code organization:

- How program parts should interact with each other.
- How modules should depend on each other.
- What level of code coupling is acceptable.

The goals of good architecture are to:

- Not interfere with development and not block changes to the program;
- Reduce the number of decisions we have to make at the moment;
- Standardize engineering practices and the process of adding features;
- Be flexible and leave room for future changes.

Not every application needs sophisticated architecture. Spending time improving the architecture of a prototype that will be abandoned makes no sense. However, if an application needs to live a long time, and we don't want the code organization to slow down with the development, we can try to put some time into architectural improvements.

> **However** 💰
>
> There's no guarantee that these attempts will improve things. Sometimes it is easier to rewrite the application from scratch, taking into account the experience we gained over the project's lifetime. But we'll discuss the "refactor or rewrite" decision separately in a later chapter.

> **Clarification** 🛠️
>
> In this chapter, we won't discuss "how to build an architecture from scratch" or "what to do in really bad cases." Instead, we'll discuss tricks that can relatively cheaply help fix the most common problems in an "average" application.

Business Workflow Modeling

The main value of an application is in its business logic. Business logic is the processes that bring profit. The description of such processes in the code is called the *domain model*.

When refactoring an application, we should consider how clearly and unambiguously this model is defined. Can different developers tell from a piece of code what it's about? How accurately can they do this? How much variation will there be in their opinions?

The clearer the domain model is stated, the more unambiguous the developers will perceive it. A clear model will help them communicate more effectively with each other and the business when discussing tasks and project requirements.

Ubiquitous Language

In the chapter on variable names, we mentioned the *ubiquitous language*.🔗🔗 It's the set of terms used by "business people" when describing business workflows.

When refactoring, we should ensure that the terms in the code are consistent with this language. The ubiquitous language reduces controversies in the code because it creates unambiguous relationships between a term and an entity that the term describes.

In the example below, the code doesn't follow the ubiquitous language and uses different terms for the user of the online store:

```
const isMerchant = (user) => user.role === "seller";
const SellerPanel = ({ user }) => isMerchant(user) && <main>{/*...*/}</main>;
```

In the snippet above, it's not clear from the context how the terms "seller" and "merchant" differ. Is there a difference between the entities they describe? If so, what's the difference?

If the terms mean the same thing, choosing and keeping only one of them is better. Preferably, choose the one used by the product owners or stakeholders.

```
// Product owners use the term “seller,” so should we:

const isSeller = (user) => user.role === "seller";
const SellerPanel = ({ user }) => isSeller(user) && <main>{/*...*/}</main>;
```

If the terms mean different things, it's a reason to check whether the code correctly reflects the domain. For example, if the term “merchant” has something to do with the user's *company*, it should raise the question of whether the `SellerPanel` component really needs to check the *user* role.

Domain Model

As mentioned earlier, the domain model is data transformations that reflect business workflows. It's convenient to represent domain logic as separate functions, which we can combine into chains of data transformations.

> **Read more** ⛓
>
> We talked more about data transformation chains in the chapter on functional pipelines.

To see why it's beneficial, consider an example. Let's say we have an online auction site function in front of us. The function creates a new auction object, but it tries to put in it *everything that might happen* to the auction during its lifetime:

JavaScript

```
function composeAuction(user, from, to, products, startPrice, invited) {
  return {
    created: true,
    author: user,
    timeRange: { from, to },
    lots: products.map(lotFromProduct),
    price: startPrice,
    participants: invited.filter(accepted),
    winners: [],
    bids: [],
    bestBid: null,
    open: false,
    expired: false,
    cancelled: false,
    closedLots: null,
  };
}
```

The problem with the `composeAuction` function is that it doesn't differentiate between the different stages of the auction lifecycle. The freshly created object is in an *undetermined* state. Its structure doesn't unambiguously reflect the lifecycle stage it is in.

The created object has excessive, mutually exclusive, or contradictory fields. For example, as long as the auction hasn't started, it probably doesn't make sense to add bid history or the current lot. Or, the `open` and `expired` flags are mutually exclusive, but because they're in the object simultaneously, it seems they're not.

The structure of the created object is unfriendly to the reader. Without additional context, we can't understand the rules by which the object's fields work and change. We need documentation to understand how such an auction object lives and evolves.

When refactoring, we should remember to represent the domain model as a set of distinct operations in the code. We can first identify all the possible states the data goes through and then describe the transformations that will bring the data to those states. For example:

JavaScript

```
// First, we create a new auction:
function createAuction(user, from, to, lots, startPrice) {
  return {
    author: user,
    timeRange: { from, to },
    lots: products.map(lotFromProduct),
    price: startPrice,

    // The `status` field will help validate the object
    // and check what operations can be performed on it:
    status: STATUS.created,

    // Different lifecycle states will be _unambiguously_ marked by this field.
    // We can't create an auction that has different statuses at the same time,
    // it's either `STATUS.created`, or `STATUS.expired`, or something else.
  };
}

// Suppose inviting participants is a separate workflow,
// then, we'll have a separate function for it:
function inviteParticipants(auction, participants) {
  return {
    ...auction,
    participants,
    status: STATUS.inviting,
  };
}

// This function will check if an auction can start:
function canStart(auction) {
  const moment = Date.now();
  const { from, to } = auction.timeRange;
  return moment >= from && moment <= to;
}

// When the auction starts, it contains only the participants,
// who have accepted the invitation:
function startAuction(auction) {
  return {
    ...auction,
    participants: auction.participants.filter(accepted),
    currentLot: auction.lots.find(available),
    status: STATUS.active,
  };
}

// Each new bid will be appended to the history.
// (The implementation might defer depending on the concurrency requirements.)
function addBid(auction, bid) {
  return { ...auction, bids: [...auction.bids, bid] };
}

// When the auction expires, it changes the status:
function expireAuction(auction) {
  const { currentLot, ...rest } = auction;
```

```
  return {
    ...rest,
    closedLots: auction.lots.filter(isClosed),
    status: STATUS.expired,
  };
}

// When defining the winners, check who won at least one lot:
function defineWinners(auction) {
  return { ...auction, winners: participants.filter(ownsLot) };
}

// ...And so on for all the business workflows.
```

Functions from the domain model should be able to "play" business workflows from beginning to end, transforming data. If we can do this, we've wholly and adequately reflected the domain in the code.

Yes, there's more code now. But at the same time, there's more *information* about the domain. We express more knowledge about how the domain works and what data is involved in the business workflows. There are also more *constraints* that the application has to consider. For example, we explicitly say that there can be no winners in an ongoing auction, and that's why the `winners` field doesn't exist in the auction until it's finished.

The more information about the domain we express in the code, the more errors we'll find *at the design stage*. The more inconsistencies we resolve in the design before starting to code, the less inconsistency will be there in the code.

In languages with static typing, when refactoring, we can also explicitly describe data states as types:

TypeScript

```
type CreatedAuction = {
  author: User;
  timeRange: LimitedTimeFrame;
  lots: ReadonlyList<Lot>;
  price: MonetaryValue;
  status: "created";
};

type PendingAuction = {}; // ...
type ActiveAuction = {}; // ...
type ExpiredAuction = {}; // ...
type FinishedAuction = {}; // ...

// When using `Result` to handle errors,
// we can also describe all potential problems as types:

type ExpiredInvitation = "...";
type CantAddParticipantsAfterStart = "...";
type CantModifyExpiredAuction = "...";
// ...

type DomainError =
  | ExpiredInvitation
  | CantAddParticipantsAfterStart
  | CantModifyExpiredAuction;
// ...
```

This way, we'll notice the objects and processes that don't contain enough data or contain unnecessary data more quickly.

Dependency Direction

The domain model is the most critical part of the application. It describes the key features of the project and carries its business value. The rest of the code must serve it and help connect it to the outside world.

Application use cases should be "built around" the domain transformations. We can achieve this with the Impureim sandwich code style.[🔗] In this kind of organization, we first get the data from impure sources, then run it through a chain of transformations and then save it or render it on the screen.

> **In detail** 🥪
>
> In more detail, we discussed Impureim-sandwich and the "functional core in an imperative shell" approach in the chapter on side effects.

Interaction with the World

When refactoring a large chunk of an application, we should pay attention to the coupling between the app and the outside world. The coupling can be explicit when we use a third-party service directly in our code. But also, the coupling can be implicit.

For example, in an online store, the server requires an object with the `products` field in the client's request to create an order. Let's say the client code uses some object that contains the `products` field. It's tempting to use this object directly when sending the order:

```JavaScript
function createOrder(user, products) {
  return { user, products };
}

// ...

async function sendOrder(order) {
  const response = await fetch("/api/orders/", {
    method: "POST",
    body: JSON.stringify(order),
  });
}

// ...
const order = createOrder(currentUser, productList);
await sendOrder(order);
```

It looks handy, but it creates a coupling point with the server. Now, we can't change the structure of the `order` object inside `createOrder` because that would break compatibility with the server.

Such coupling can be no problem for small projects that rarely change data structures or APIs. But if we care about app scalability or API flexibility, or if we often change object fields in the client code, this will become a problem.

In such situations, we can add an *Anti-Corruption Layer, ACL*, a set of transformations responsible for compatibility between the API and our code.

> **Be careful 🚧**
>
> Anti-Corruption Layer should not be confused with Access Control List. Both concepts are abbreviated as ACL but differ in meaning.

The Anti-Corruption Layer can be a single-function adapter, like `toServerOrder` in the example below:

```JavaScript
function toServerOrder(clientOrder) {
  // const serverOrder = ...

  // Here go the transformations required
  // for creating a compatible data structure.

  return serverOrder;
}

async function sendOrder(order) {
  const dto = toServerOrder(order);
  // ...Prepare the object first
  // and then send it to the server.
}
```

...Or it can be a separate module. For example, suppose the frontend has to work with several incompatible API versions simultaneously. In that case, such a module can contain the logic for mapping the correct data structures to a particular API version.

```JavaScript
function toServerOrder(clientOrder, config) {
  const dataAdapter = orderAdapters[config.currentApiVersion];
  const serverOrder = dataAdapter(clientOrder);
  return serverOrder;
}
```

It frees the frontend from constantly checking the data against the server structure *inside the client application*. We now have a single place where the data is being prepared before sending. The rest of the code is free to change data structures and object fields how it wants. The client becomes *uncoupled* from the server.

With static typing, all data conversions can be expressed explicitly with types, so that the difference between structures is more noticeable:

```TypeScript
type Order = {
  user: User;
  products: List<Product>;
};

type ServerOrderDto = {
  userId: EntityId<User>;
  orderItems: List<OrderLine>;
};

type ServerOrderSelector = (clientOrder: Order) => ServerOrderDto;
```

The anti-corruption layer can increase code complexity though. It's usually worth adding if we assume that the data or API might change. For example, it's helpful in analytics, logging, interaction with the user's device storage, etc.

> **By the way** 📜
>
> To understand where we might need an anti-corruption layer, it's handy to use the repository history. We can collect information from the repository about what project code changed and how often. With this data, we can assume what might change in the future. These stats can help us avoid huge code rewrites and compatibility problems in the future.
>
> The technique of "metadata mining" is well described in "Your Code as a Crime Scene" by Adam Tornhill.🔗

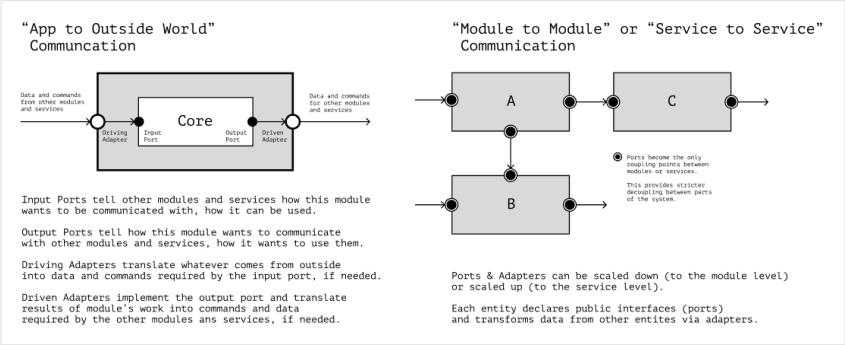
Ports and Adapters

Since we already mentioned adapters, the "Ports and Adapters" architecture pattern is worth discussing.🔗🔗

An *adapter* is an entity that converts one module's incompatible interface to another's requirements. A *port* is a specification of how a module wants to be communicated.

> **For example** 💬
>
> A module declares *ports* to signal other modules how they can communicate with it. If module A wants to talk to module B, but their ports are incompatible, module A creates an *adapter*. The adapter makes the interfaces compatible and communication possible.

With the "Ports and Adapters" architectural style, the module communication comes down to connecting the ports of some modules with the adapters of others. In this way, we make the modules unaware of each other's structure, which reduces the coupling between them.



When using "Ports and Adapters," module interaction comes down to connecting adapters to ports of other modules

Using "Ports and Adapters" automatically forces the "Impureim sandwich" code style because it pushes "interaction with the world" to the edges of the application.

It also helps reduce the number of mocks in tests. If we use *a single* adapter for network requests, we need to test *only* it instead of mocking every network request. It reduces the number of tests to update after code changes, lowering the test-induced damage.

Finally, ports and adapters limit the propagation of changes within the module. The adapter serves as a barrier where the change should stop preventing it from propagating throughout the code base.

> **However** 👀
>
> Not every project may need this architectural style. In simple applications that aren't going to scale, ports and adapters can add extra work with no apparent benefit.

All Kinds of Architecture

We're not going to study all the ways to design apps. Instead, I'll leave some links to articles and books which I find the most useful.

For the frontend, though, it's worth separately mentioning Feature-Sliced. It's a methodology for frontend application design that tries to collect developer experience and practices not only from the frontend but from the backend as well. The people behind it are doing a great job, which in my opinion, isn't appreciated enough.

UI Logic

Developing user interfaces is rather a chaotic task. On average, the UI changes faster and more often than the domain code, but the user needs to see consistent and valid data in the interface. To keep this true, when working with user interfaces, we can use the rule:

> **Always keep UI logic and business logic separate ❗**

Even if the data in the UI looks the same as the domain model data, it's better to think of them as *different* data. The UI changes faster than the domain model, so sooner or later, these data sets will start to differ. If we don't distinguish between the business and UI logic, we can miss this moment, and domain objects will end up containing data from UI.

For example, let's imagine we need to render a shopping cart. In the UI, it has two views: compact and expanded. A flag that determines the view may accidentally "leak" into the domain model:

```JavaScript
const cart = {
  products: [chocolateBar, sodaCan, teslaCar],
  user: someUser,

  // Only needed in the UI:
  isExpanded: true,
};
```

The problem with this "leaking" is that the `cart` object now contains data from different levels of abstraction and different tasks. This means that testing the application with this data and preparing it to be sent to the server will be more difficult. Also, changes in the UI now may cause unnecessary changes to the entire application.

Finally, if we keep ignoring data separation, the UI data can "metastasize" to other objects as well:

```JavaScript
const cart = {
  // "Zoom in" to the user object...
  user: {
    name: "Pippi Longstocking",
    role: "buyer",

    // ...And more fields are related only to the UI:
    theme: SITE_THEME.light,
  },

  products: [chocolateBar, sodaCan, teslaCar],
  isExpanded: true,
};
```

To solve these problems, it's worth separating the UI logic from the domain logic:

```JavaScript
// The separation can be done like this:

const cart = {
  user: someUser,
  products: [chocolateBar, sodaCan, teslaCar],
};

const cartView = {
  isExpanded: true,
};

// Or like this:

const cartUi = {
  model: cart,
  view: cartView,
};

// Or like this:

const cartUi = {
  cart,
  ...cartView,
};

// Or even like this,
// if `cart` and `cartView`
// are created separately from each other:

const cartUi = {
  ...cart,
  ...cartView,
};
```

How exactly to separate the objects depends on the task. The main idea is that the business logic data *should, in principle, be separated* from the presentation layer data. We want to decouple the UI from the domain and allow them to evolve independently of each other.

Reactivity

When it comes to UI logic, it's worth mentioning reactivity. *Reactive interfaces* instantly map changes from UI to the data model and back.[🔗] With reactive interfaces, the temptation to merge domain data with UI data is even higher because reactivity creates the illusion that these data sets are identical.

When working with reactive interfaces, it's convenient to use the MVVM pattern.[🔗🔗] In this pattern, the view-model works as a mediator between UI and business logic, separating them and not allowing them to mix.

In addition, it's useful to use tools for *reactive data mapping* with selectors. These can be used to "compile" the final data for the presentation layer. The Effector library handles this

very well:

```TypeScript
// We can separately declare domain logic types:
export type ConverterState = {
  unit: Unit;
  value: TimeStamp;
};

// ...And separately declare the types of data that the UI should display:
export type UiRepresentation = {
  date: StringRepresentation;
  stamp: NumberRepresentation;
  open: boolean;
};

// We can keep these states and their logic separate and independent:
const $converter = createStore<ConverterState>({ unit, value });
const $isOpen = createStore<boolean>(false);

// And for the UI, we can map domain data to the UI:
const $representation = combine<UiRepresentation>(
  $converter,
  $isOpen,
  (converter, isOpen) => ({
    stamp: toMilliseconds(converter.unit, converter.value),
    date: toDateString(converter.value),
    open: isOpen,
  })
);
```

As a result, the business logic becomes isolated, and the data gets into the UI through data selectors. It decouples UI from the rest of the code, making it easier to evolve independently.

Testing

From the architecture point of view, we can evaluate the result of the refactoring by how easy it is to test the code. We'll borrow a heuristic for it from "The Grand Unified Theory of Clean Architecture and Test Pyramid":

- If we can test business logic with unit tests;
- And we can test adapters with integration tests;
- And we can test application use cases with E2E tests...

...Then the architecture works. The code is sufficiently decoupled, the UI doesn't get mixed with the business logic, and it takes less time and resources to test and maintain the tests.

1. "Software Architecture in Practice" by L. Bass, P. Clements, R. Kazman, https://www.goodreads.com/book/show/70143.Software_Architecture_in_Practice

2. "Enterprise Integration Patterns" by Gregor Hohpe, https://www.goodreads.com/book/show/85012.Enterprise_Integration_Patterns

3. “Designing Data-Intensive Applications” by Martin Kleppmann https://dataintensive.net

4. “Clean Architecture” by Robert C. Martin, https://www.goodreads.com/book/show/18043011-clean-architecture

5. “What I’ve Learned From Failure” by Reg Braithwaite, https://leanpub.com/shippingsoftware/read

6. “Domain Modeling Made Functional” by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional

7. Software Requirements, Wikipedia, https://en.wikipedia.org/wiki/Software_requirements

8. List of System Quality Attributes, Wikipedia, https://en.wikipedia.org/wiki/List_of_system_quality_attributes

9. “Domain-Driven Design” by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design

10. “Ubiquitous Language” by Martin Fowler, https://martinfowler.com/bliki/UbiquitousLanguage.html

11. “Impureim Sandwich” by Mark Seemann, https://blog.ploeh.dk/2020/03/02/impureim-sandwich/

12. Anti-corruption Layer Pattern, Microsoft Docs, https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer

13. Access Control List, Wikipedia, https://en.wikipedia.org/wiki/Access-control_list

14. “Your Code As a Crime Scene” by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene

15. “Functional Architecture is Ports and Adapters” by Mark Seemann, https://blog.ploeh.dk/2016/03/18/functional-architecture-is-ports-and-adapters/

16. “Ports & Adapters Architecture” by Herberto Graça, https://herbertograca.com/2017/09/14/ports-adapters-architecture/

17. “Test-Induced Design Damage” by David Heinemeier Hansson, https://dhh.dk/2014/test-induced-design-damage.html

18. “The Software Architecture Chronicles” by Herberto Graça, https://herbertograca.com/2017/07/03/the-software-architecture-chronicles/

19. “Functional architecture: The pits of success” by Mark Seemann, https://youtu.be/US8QG9I1XW0

20. Feature-Sliced Design, Architectural methodology for frontend projects, https://feature-sliced.design

21. Reactive programming, Wikipedia, https://en.wikipedia.org/wiki/Reactive_programming

22. Model-View-ViewModel, Wikipedia, https://en.wikipedia.org/wiki/Model–view–viewmodel

23. View Models, Reactive UI, https://www.reactiveui.net/docs/handbook/view-models/

24. Effector, Business logic with ease, https://effector.dev

25. “The Grand Unified Theory of Clean Architecture and Test Pyramid” by Guilherme Ferreira, https://youtu.be/gHSpj2zM9Nw

26. “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing

16

Declarative Style

In the chapter about abstraction, we discussed how to decompose tasks and why separating the intent from implementation in code is beneficial. We looked at how to improve code readability and emphasize details that are important at a particular moment.

In this chapter, we'll extend the ideas of abstraction and talk about declarative code style. We'll discuss the notion of declarative code, its benefits, and its advantages compared to imperative-style code.

Readability

To better understand the benefits of declarative style, let's first discuss the difference between declarative and imperative code. Consider the two snippets below:

```JavaScript
// 1.

function keepEvenNumbers(array) {
  const result = [];

  for (const x of array) {
    if (x % 2 === 0) {
      result.push(x);
    }
  }

  return result;
}

// 2.

function keepEvenNumbers(array) {
  return array.filter((x) => x % 2 === 0);
}
```

Both functions filter the given array of numbers, keeping only even ones. The difference between them is how they do it. The first function describes *how* to solve the problem as a set of instructions:

- Create an empty array `result`.
- Iterate over the `array` argument.
- For each element, check if it's even.
- If yes, add it to `result`.

On the other hand, the second function describes *what* needs to be done. It focuses on the filtering *criteria*, not the details of the filtering algorithm.

This is the difference between the imperative and declarative styles. Declarative code describes *what* to do, while imperative code describes *how* to do it.

Imperative style is often harder to read because it mixes intent and implementation details. The declarative style, on the contrary, encourages us to decompose tasks and split the code by levels of abstraction. The names of functions and variables in declarative code convey more information, making the code easier to read.

For example, look at the `validate` function in the code snippet below. The function body contains too many details, and its name says little about its purpose. This code is hard to understand at a glance:

```JavaScript
function validate(user, cart) {
  return (
    !!cart.items.length &&
    user.account >= cart.items.reduce((tally, item) => tally + item.price, 0)
  );
}
```

To solve this problem, we can decompose it and split function steps by levels of abstraction. For example, we can extract validation checks into separate functions, the names of which will reflect the purpose of those checks:

```JavaScript
// cart.js
function isEmpty(cart) {
  return !cart.products.length;
}

function totalPriceOf(cart) {
  return cart.items.reduce((tally, item) => tally + item.price, 0);
}

// user.js
function canAffordSpending(user, amount) {
  return user.account >= amount;
}
```

The extracted functions' names are now expressed in terms more appropriate to the current level of abstraction. The names now better describe the functions' meaning. They help control the reader's attention when using these functions within `validate`:

```
// order.js

function validate(user, cart) {
  return !isEmpty(cart) && canAffordSpending(user, totalPriceOf(cart));
}
```

> **In detail** 💡
>
> We talked more about the levels of abstraction, switching between them, and controlling the reader's attention in the chapter on abstraction.

Also, the extracted functions' names now carry some information about the purpose of the `validate` function. We can replace the name `validate` with a more informative one, for example, `canMakeOrder`. Then the function code will turn into "text," similar to a regular sentence:

```
// order.js

function canMakeOrder(user, cart) {
  const orderPrice = totalPriceOf(cart);
  return !isEmpty(cart) && canAffordSpending(user, orderPrice);
}

// The (user) (canMakeOrder) IF the (cart) (!isEmpty)
// AND they (canAffordSpending) (orderPrice) of money.
```

Declarative code is similar to how people talk to each other in real life, so it's easier to understand. It communicates valuable information but doesn't overload the reader with unnecessary details. Such "communication" is polite, more casual, and less tiresome.

> **However** 🐣
>
> In smaller projects, thorough decomposition may not be that important. The less code, the less tiresome it is. The need for decomposition depends on how complex it is for the team to read the code and work with the current code base.

Reliability

The following section is controversial and subjective, but in my experience, it's easier to make accidental mistakes in imperative code. Partly because, in imperative code, we have to

think about “the goal” and “how to achieve it” simultaneously, but also because imperative code often contains more lines and statistically is more likely to contain an error.

For example, let’s look at the function `selectOperation` that chooses a mathematical operation by the given key:

```JavaScript
function selectOperation(kind) {
  let operation = null;
  switch (kind) {
    case "log":
      operation = (x, base) => Math.log(x) / Math.log(base);
    case "root":
      operation = (x, root) => x ** -root;
    default:
      operation = (x) => x;
  }
  return operation;
}
```

Every `case` block of this function misses the `break` statement. As a result the `operation` variable will always be equal to `(x) => x`. Such an error is relatively easy to spot in a small function, but if there’s a lot of code, it’s much easier to miss.

We can improve the code by using `return` inside the `case` blocks:

```JavaScript
function selectOperation(kind) {
  switch (kind) {
    case "log":
      return (x, base) => Math.log(x) / Math.log(base);
    case "pow":
      return (x, power) => x ** power;
    default:
      return (x) => x;
  }
}
```

However, this doesn’t solve the problem with accidental errors but only hides them. In the snippet above, for example, we may forget to add `return`, and the function will work incorrectly. We can get rid of the problem by making the selection declarative:

```JavaScript
const log = (x, base) => Math.log(x) / Math.log(base);
const pow = (x, power) => x ** power;
const id = (x) => x;

function selectOperation(kind) {
  const operations = { log, pow, id };
  return operations[kind] ?? operations.id;
}
```

In the code above, we delegate the "selection" to the *language*. We specify an object with data and a selection criterion, and the interpreter finds the value in the object according to the specified key. We don't care how that choice is made. We only care about its result. That's the essence of declarative style.

> **Matter of taste** 🍕
>
> I also like the last snippet for aesthetic reasons. Selecting from an object by key looks like a more natural solution to this problem, while code with `switch` seems noisy and verbose.

Extensibility

Extending imperative code is often more difficult than extending declarative code.

In imperative code, when adding a new feature, we not only have to figure out *what* to add but also *where* and *how* to add it. On the other hand, extending the functionality of declarative code often comes down to updating the "algorithm parameters."

> **By the way** 🖥
>
> I borrowed the following example from a lecture by Timur Shemsedinov on declarative style and metaprogramming. 🔗 Highly recommended to check out.

For example, let's compare two implementations of the `parseDuration` function. The function converts a formatted string with a time period into the number of milliseconds in that period. In the first version, the algorithm is implemented imperatively:

JavaScript

```
/** @example parseDuration('1h 25m 16s') === 5_116_000 */
function parseDuration(stringRepresentation) {
  const s = stringRepresentation;
  if (typeof s !== "string") return 0;

  const seconds = s.match(/(d+)s/);
  const minutes = s.match(/(d+)m/);
  const hours = s.match(/(d+)h/);

  let duration = 0;
  if (seconds) duration += +seconds[1] * 1000;
  if (minutes) duration += +minutes[1] * 1000 * 60;
  if (hours) duration += +hours[1] * 1000 * 60 * 60;
  return duration;
}
```

...In the second one—declaratively:

```JavaScript
// All information about the supported string format
// is located in the `MULTIPLIER` object:
const MULTIPLIER = {
  s: 1000,
  m: 1000 * 60,
  h: 1000 * 60 * 60,
};

// Algorithm steps are now separate functions:
const sumDurations = (sum, [value, unit]) => sum + value * MULTIPLIER[unit];
const hasValidValue = ([value]) => !Number.isNaN(value);

const parseComponent = (component) => {
  const value = +component.slice(0, -1);
  const unit = component.slice(-1);
  return [value, unit];
};

/** @example parseDuration('1h 25m 16s') === 5_116_000 */
function parseDuration(stringRepresentation) {
  if (typeof stringRepresentation !== "string") return 0;

  const components = stringRepresentation.split(" ");
  return components
    .map(parseComponent)
    .filter(hasValidValue)
    .reduce(sumDurations, 0);
}
```

The second snippet is easier to extend because it *divides code that changes rarely from the code that changes frequently*. If we, for example, want to extend the string format with days and weeks, we *only* need to update the `MULTIPLIER` object. The rest of the function code will remain unchanged.

It's like we extract the "algorithm parameters" into the `MULTIPLIER` object, separating them from the "logic" of the function. The supported string format is now more explicit, and the algorithm is more flexible because it can work with more different values from this object. "Parameters" are no longer "inlined" in the algorithm.

> **By the way** ⛔
>
> This ability to add similar functionality without changing the existing code is the goal of the Open-Closed principle from SOLID.🔗

The string format extension now comes down to adding new fields to the `MULTIPLIER` object. The code of the algorithm itself won't change:

```JavaScript
const MULTIPLIER = {
  s: 1000,
  m: 1000 * 60,
  h: 1000 * 60 * 60,

  // Added days and weeks:
  d: 1000 * 60 * 60 * 24,
  w: 1000 * 60 * 60 * 24 * 7,
};

// Since the rest of the function code is the same,
// the probability of an accidental error is lower.
// Besides, by using the data in the `MULTIPLIER` object
// it's easier to generate test data for the function automatically.
```

In the first implementation, we'd need to change the code of the entire `parseDuration` function:

```JavaScript
function parseDuration(stringRepresentation) {
  const s = stringRepresentation;
  if (typeof s !== "string") return 0;

  const seconds = s.match(/(d+)s/);
  const minutes = s.match(/(d+)m/);
  const hours = s.match(/(d+)h/);
  const days = s.match(/(d+)d/);
  const weeks = s.match(/(w+)d/);

  let duration = 0;
  if (seconds) duration += +seconds[1] * 1000;
  if (minutes) duration += +minutes[1] * 1000 * 60;
  if (hours) duration += +hours[1] * 1000 * 60 * 60;
  if (days) duration += +days[1] * 1000 * 60 * 60 * 24;
  if (weeks) duration += +weeks[1] * 1000 * 60 * 60 * 24 * 7;
  return duration;
}

// Speaking of how easy it is to make an accidental error in imperative code:
// Have you noticed a typo in the regular expression for `weeks`? :-)
```

> **However** 👀
>
> Extracted "parameters" are suitable for extending code with *similar* functionality. If we need to add new behavior to the algorithm itself, it may not help.

It's worth remembering that such "metaprogramming" isn't always needed, because *generalized functions can be more complex*.

As a rule, declarative generalization is useful when we notice a part of a function that "changes too often" compared to the rest of the code. Such functionality can be made

declarative. It'll reduce the probability of accidental errors when updating it.

Configurability

We extracted the "algorithm parameters" in a separate object in the previous example. We motivated this because code and its settings change at different rates, so keeping them separate is better.

In fact, this is one of the well-known rules that 12-Factor Apps recommend.[🔗] This rule can be described as:

> **Always keep configs separate from code !**

Settings and configs, on average, change more frequently than the code they configure. When configs are inlined in the code, changing them is more dangerous and difficult than when they're separate.

For instance, a hardcoded configuration makes it hard to change the program's environment. For example, consider an app that should run in test and production environments. If the deployment settings or third-party services configuration is hardcoded, the environment switch requires updating the configs manually in the code.

> **By the way 🖌**
>
> This property helps to distinguish configs from the rest of the code. If the value of a variable depends on the environment, it's definitely a part of the configuration.

For example, suppose the base URL of the API in the code below should change for different environments. In the `fetchUser` function though, the base URL is hardcoded into the body of the function:

```JavaScript
async function fetchUser(id) {
  const response = await fetch(`https://api.our-app.com/v1/users/${id}`);
  const data = await response.json();
  return data.user;
}

// Calls to `fetchUser` access a specific API version: `api.our-app.com`.
// We need to update the function code to change the environment.
```

It's convenient to extract the configuration by following the *Transformation Priority Premise, TPP*.[🔗] First, we should extract configs to local variables and then to environment variables or configuration files:

```JavaScript
// Step 1: extract configs to local variables.
const baseUrl = "https://api.our-app.com";
const apiVersion = "v1";

async function fetchUser(id) {
  const response = await fetch(`${baseUrl}/${apiVersion}/users/${id}`);
  const data = await response.json();
  return data.user;
}

// Step 2: extract them into configs
// (environment variables or configuration modules).
// The goal is to create a _distinct_ separation
// between the code and the configs.
import { networkConfig } from "@config";

async function fetchUser(id) {
  const response = await fetch(`${networkConfig.apiRoot}/users/${id}`);
  const data = await response.json();
  return data.user;
}
```

State Machines

As mentioned in the previous chapter, business logic and UI logic are best kept separate. The business logic code should be responsible for the business workflows and associated data transformations. The UI-logic code should be responsible for rendering the user interface.

The UI logic can be complex. For example, it might describe the behavior of interdependent components or dynamic interface that depends on many conditions.

To keep the code of complex UI logic readable, we can represent the UI as a finite set of its states. Each such state describes the interface visible to the user and the conditions under which it's rendered on the screen.

The interaction with the UI can then be described as a network of such states. If the number of states is limited, we can call such an interaction a *Finite State Machine, FSM.*

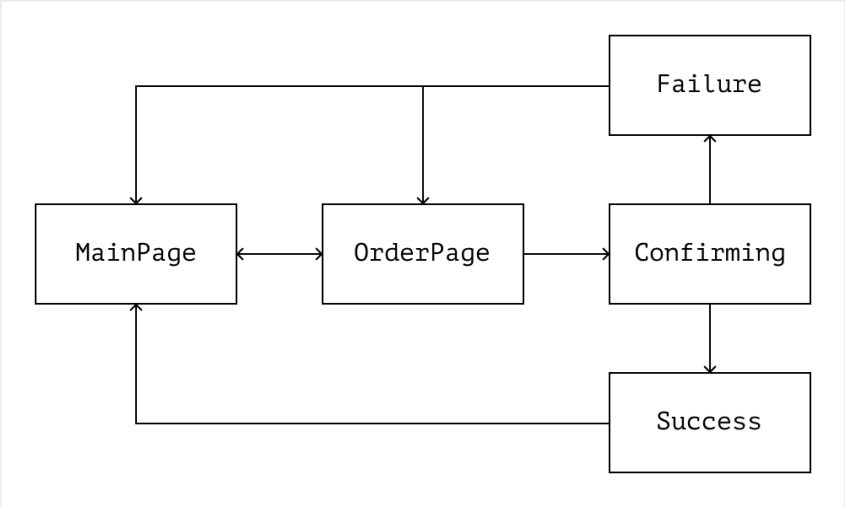
> **By the way** 🤖
>
> Finite State Machine is a mathematical concept, but it's useful for describing UI as a function of the data state. It helps make UI more deterministic.

The basic idea of an FSM is *limited sets* of states and *unambiguous rules for transitions* between them. For example, the UI of an online store during order placement can be expressed as a such set of states and transitions:

"Checkout Use Case":

Current State	Allowed Transitions
`OrderPage`	`MainPage`, `Confirming`
`Confirming`	`Success`, `Failure`
`Success`	`MainPage`
`Failure`	`MainPage`, `OrderPage`

We can represent this table as a diagram of transitions between states:



Online store state transitions diagram

But we can also represent it *in code* as a collection of states and transitions:

```JavaScript
const fsm = createMachine({
  states: {
    main: {}, // Transitions from `MainPage`...
    order: {}, // From `OrderPage`...
    confirming: {}, // From `Confirming`...
    success: {}, // From `Success`...
    failure: {}, // From `Failure`...
  },
});
```

Then, we can describe the states in terms of components:

```JavaScript
// Represents the `OrderPage` state:
const ConfirmOrder = () => (
  <form onSubmit={fsm.to("confirming")}>{/*...*/}</form>
);

// Represents the `Success` state:
const OrderConfirmed = () => (
  <>
    Order Confirmed.
    <a href={fsm.to("main")}>Back to main page</a>
  </>
);

// Represents the `Failure` state:
const OrderError = () => (
  <>
    Couldn't confirm the order.
    <a href={fsm.to("main")}>Back to main page</a>
    <a href={fsm.to("order")}>Try again</a>
  </>
);

// Represents the `Confirming` state:
const Confirming = () => "Loading...";
```

Then the app entry point can use the FSM to decide what to render on the screen:

```JavaScript
// Selects the component based on the state:
function Checkout() {
  const [state] = useMachine(fsm);

  return state.match
    .with("confirming", Confirming)
    .with("success", OrderConfirmed)
    .with("failure", OrderError)
    .orElse(ConfirmOrder);
}
```

A benefit of an FSM is that it allows going from one state *only to a specific set of following states*. It's impossible to go to states outside this list. FSMs make the UI more declarative and deterministic.

Also, using FSM, it's easier to separate UI data from business logic data. In general, the FSM states have identifiers that relate them to the result on the screen. From these identifiers, we can always separate the data that relates only to the UI and not mix it with the business logic data.

> **Tools** 🧰
>
> The FSM and the logic of transitions between states can be described using different tools. In the example above, I used a fictional library. However, as a good example of an FSM UI library, I can suggest XState.🔗

Drawbacks

As we mentioned above, the declarative code style has disadvantages.

Code Complexity

Generalizations can make code more complex. A declarative "facade" can hide an overly complex abstraction that's hard for other developers to understand.

After refactoring, we should always check to see if the code changes have benefited. If the code has become harder to read or maintain, it's better to roll back the changes.

When we doubt support complexity, we can request a code review from more developers than usual. This way, we'll know if the code is still readable and easy to maintain.

Performance

Imperative code tends to be more efficient. If performance is more important than readability, we can sacrifice declarative style.

In doing so, it can be useful to isolate imperative code from the rest. For example, if we need some efficient algorithm in an application, we can implement it in a function:

```JavaScript
function mergeTrees(treeA, treeB) {
  // ...Efficient algorithm implementation.
}
```

...But implement the rest of the app declaratively:

```JavaScript
function mergeCompanyDepartments(departmentIdA, departmentIdB) {
  return mergeTrees(
    extractDepartment(departmentIdA),
    extractDepartment(departmentIdB)
  );
}
```

In this way, we bring the imperative implementation "down a level," isolating it from the rest of the code, making the function name a declarative description of the entire algorithm.

1. “Your Code As a Crime Scene” by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene

2. “Metaprogramming with JavaScript Examples” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Metaprogramming

3. The Principles of OOD, Robert C. Martin, http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

4. 12 Factor Apps, https://12factor.net

5. Transformation Priority Premise, Wikipedia https://en.wikipedia.org/wiki/Transformation_Priority_Premise

6. Finite-State Machine, Wikipedia, https://en.wikipedia.org/wiki/Finite-state_machine

7. “Application State Management with Finite State Machines” by Alex Bespoyasov, https://bespoyasov.me/blog/fsm-to-the-rescue/

8. JavaScript and TypeScript finite state machines and statecharts, XState, https://github.com/statelyai/xstate

17

Static Typing

In previous chapters, we mainly focused on common language features like variables, functions, and modules when talking about readability. We occasionally paid attention to types but didn't dive into the details of working with them.

However, static typing can also be a tool for writing expressive code. We can use types and interfaces to convey additional information to the reader or to make the app design more explicit.

This chapter will discuss how to express more domain knowledge through types and make invalid data transformations unrepresentable. We'll look at how to use ubiquitous language in type signatures and track down errors in API design using types.

> **Before we start** 💬
>
> We won't discuss if one *should* use static typing or not. Instead, we'll focus on how to use types as refactoring tools.
>
> Static typing is a controversial topic, not everyone likes it, and it's okay. You can skip this chapter if your team doesn't use or like typing.

Ubiquitous Language

In "Domain Modeling Made Functional," Scott Wlaschin warns readers against primitive obsession.🔗🔗 He suggests using types to describe the domain to avoid it.

> **Be careful** 🚨
>
> The idea of replacing primitives with domain types can be controversial to the team. Discuss this idea with other developers before applying it.

Data types and function signatures can carry information about the task the code is solving. They can reflect the task context, the interaction between entities, and even the business workflow model as a whole. When used thoughtfully, types can even become an alternative to documentation:

```TypeScript
// Primitive types don't reflect the context,
// they lack the details of the domain:

type Account = {
  date: string;
  user: number;
  value: number;
};

// Domain types help describe entity relations
// and how they work together:

type Account = {
  date: DateTimeIso;
  user: UserId;
  value: MoneyAmount;
};
```

When type names use domain terms, they form a part of the *ubiquitous language*. It's the language used by product owners and people directly related to business workflows.

> **Read more** 👀
>
> We talked more about ubiquitous language in entity names and architecture chapters.

The benefit of ubiquitous language is its *unambiguity*. If the whole team, including the non-developers, uses the same terms, there's less chance of "translation loss." It makes it easier to spot bugs and errors in the domain model at the early stages.

Domain Modeling

Data transformations are conveniently expressed through functional types in many statically typed languages. A set of such types can describe the business workflows—model the domain. The cost of an error in such a model is lower than in the workflow implementation. That is the main benefit.

Types help to build a top-level understanding of how the system works. In such a model, we can see the interaction of its parts, the module contracts, and the data used. But it also shows design errors and inconsistencies between the model and the real world.

Errors in types are easier to fix than errors in implementation. With types, it's possible to design an application *before* we start implementing it. We write a draft, look at the flaws in

the model, fix them, and recheck the model:

```TypeScript
// Describe the data used in the application.
// Indicate different states it goes through
// at different stages of the app life cycle:

type CreatedOrder = {
  createdAt: TimeStamp;
  user: UserId;
  items: ProductList;
};

type ValidatedOrder = {
  /*...*/
};

type DiscountedOrder = {
  /*...*/
};

type Order = CreatedOrder | ValidatedOrder | DiscountedOrder;

// Design the domain workflows
// that transform the data:

type CreateOrder = (user: UserId, items: ProductList) => CreatedOrder;
type ValidateOrder = (order: CreatedOrder) => ValidatedOrder;
type ApplyDiscount = (order: ValidatedOrder, value: Price) => DiscountedOrder;

// If we notice an error in any of the types,
// (for example, the designed workflow is different from what happens in reality
// we can quickly and relatively cheaply fix the model:

type ApplyDiscount = (
  order: ValidatedOrder,
  coupon: DiscountCoupon
) => DiscountedOrder;
```

During refactoring, such checks help to detect workflows that don't meet the project requirements or violate the domain constraints:

TypeScript

```
type Divider = (a: number, b: number) => number;
const divide: Divider = (a, b) => a / b;

// Should `divide` take zero as a second argument?
// If yes, how to handle the division by zero?
// Should we return the container from this function?

// We can answer these questions beforehand by enriching
// the domain model with additional data and constraints:

type RealNumber = // ...Represents any number.
type NaturalNumber = // ...Represents integers bigger than 0.

type Divider =  (a: NaturalNumber, b: NaturalNumber) => RealNumber;
const divide: Divider = (a, b) => a / b;

// Now from the `Divider` type,
// we can see that the division by zero
// should be handled by the calling side.
```

Types in TypeScript

In TypeScript, we have several ways to model the domain and create domain types:🔗🔗🔗

- Type aliases
- Classes
- Type branding

> **By the way** 🧵
>
> Because of TypeScript's structural typing,🔗 the usability and applicability of each option in the list may differ.
>
> In this book, we won't dive into the nuances of TypeScript's type system. Instead, we'll focus primarily on using types as a refactoring tool.
>
> However, for a better understanding of the constraints of structural typing and how it can affect the code, I'll leave some links on the subject in the sources.🔗🔗

The easiest but unreliable way to create domain types is to use type aliases.🔗 They're convenient to give primitive types informative names but challenging to convey the *constraints* of the domain. For example, such code is quite valid *syntactically* but not from the *domain point of view*:

TypeScript

```
// For example, a type alias can give the primitive a helpful name,
// which reflects the type meaning according to the domain:
type RealNumber = number;
type NaturalNumber = number;

// But it won't force the domain constraints:
const x: RealNumber = -1;
const y: NaturalNumber = x;

// Oops!
// -1 isn't a natural number.
```

It's difficult to reflect the domain constraints and validate assigned values in the type alias. So there's no guarantee that the argument of type `NaturalNumber` will be a natural number:

TypeScript

```
function divide(a: NaturalNumber, b: NaturalNumber): RealNumber {
  return a / b;
}

// Compiler's happy but, at the runtime, there's an error:
divide(1, 0);
```

So if we need to distinguish between types or enforce value validation, we have to use classes or branded types:

TypeScript

```
// When using classes, we can add validation
// to the class constructor:

class NaturalNumber {
  constructor(value) {
    if (value <= 0 || Math.floor(value) !== value) {
      throw new Error("The value must be a positive integer.");
    }

    this.value = value;
  }
}

// Then it'll be impossible to create an incorrect value:

new NaturalNumber(-1); // Error!
new NaturalNumber(42); // NaturalNumber

// But the classes are pretty verbose
// and aren't convenient to use as wrappers over a primitive.
// It takes a lot of code to create “numbers” via `new NaturalNumber(42)`
// and somehow implement arithmetic operations with these values.

// The second option is to use type branding:

type Tagged<T, S> = T & { __tag: S };
type NaturalNumber = Tagged<number, "natural">;

// And create values only via factory functions.
// The validation then can be placed in those functions:

function naturalFrom(value: number): NaturalNumber {
  if (value <= 0 || Math.floor(value) !== value) {
    throw new Error("The value must be a positive integer.");
  }

  return value as NaturalNumber;
}

naturalFrom(-1); // Error!
naturalFrom(42); // NaturalNumber
```

The problem with classes and type branding is that we must watch if they're used correctly. We'll need to write linter rules for their use or search for mistakes during code reviews. This approach isn't reliable.

It's hard to recommend a particular method here. It all depends on the project and the needs of the team. However, we can say that for *purely descriptive* purposes, even the type aliases work quite fine. Often a domain model built with type aliases is enough to find previously undetected design errors.

Model and Reality

Business workflows transform data from one state to another. Types can help to fixate these states and explicitly name them. When each step of the transformation has a name, it's easier for us to reason about the whole process and find errors in its logic.

In the example below, the `sendRecoverLink` function accepts an object of type `User` as an argument. This type has a `verified` flag, but there are no rules explaining *when and why* this flag becomes `true`:

TypeScript

```
type User = {
  id: string;
  verified?: boolean;
};

async function sendRecoverLink(user: User) {
  if (!user.verified) return false;
  await api.recoverPassword(user.id);
}
```

With the current `User` type implementation, the `sendRecoverLink` function accepts data that is invalid half the time. We can prevent developers from passing invalid data by making it more difficult at the type level.

User verification is probably a separate business workflow that *results in* a verified user object. This causal relationship can be expressed directly in types after we separate the types of verified and unverified users:

```TypeScript
// Describe the states of verified
// and unverified users, as different types:

type CreatedUser = { name: string };
type VerifiedUser = { name: string; verified: true };

type User = CreatedUser | VerifiedUser;

// Show the causality in the functional type of the verification process.
// Describe what steps the data goes through and how the user becomes verified:

type VerifyUser = (user: CreatedUser) => Promise<VerifiedUser>;

// Fixate restrictions on password recovery.
// Only a verified user should be able to restore the password:

type RecoverPassword = (user: VerifiedUser) => Promise<void>;

// Now, sending a recovery link to an unverified user becomes impossible.
// The invalid data transformation becomes unrepresentable in the code:

const sendRecoverLink: RecoverPassword = async (user) => {
  await api.recoverPassword(user.id);
};

sendRecoverLink(unverifiedUser); // Error!
```

Again, in TypeScript, it's challenging to achieve bullet-proof "unrepresentability" of invalid transformations. In other typed languages, it may be much easier. But even just showing different data states as types helps to notice errors in the business logic at the design stage.

It's easier to notice such errors in types because branching in types is much more verbose and difficult, unlike in implementation. It forces us to describe workflow types declaratively and linearly. If we notice ambiguity in the described model, we can act on it.

Violation of Agreements

Explicit types help with detecting violations of agreements or project rules. For example, the `sendRecoverLink` function from the previous example violates CQS:

```TypeScript
async function sendRecoverLink(user: User) {
  if (!user.verified) return false;
  await api.recoverPassword(user.id);
}

type RecoverPassword = (user: User) => Promise<false | void>;

// Here, `false` is an attribute of a query,
// while `void` is an attribute of a command.
```

Types draw attention to such contradictions. We can further improve the function, for example, by using the `try*` pattern in its name:

```TypeScript
async function trySendRecoverLink(user: User) {
  if (!user.verified) return false;
  await api.recoverPassword(user.id);
}

type TryRecoverPassword = (user: User) => Promise<false | void>;

// The `try*` pattern in the name explicitly says that the function
// is still a command but may sometimes return' false.
```

This way, we at least make the expectations of the function more explicit. But it's better, of course, to go further and refactor the function according to CQS.

> **In detail** 💡
>
> We discussed CQS and separating logic from effects in more detail in the chapter on side effects.

API Design

During refactoring, static typing can help to check how clear the API of a module or function is. For example, from the signature of a function, we can check the clarity of its API by "xxxing out" its name and the names of its arguments:

```TypeScript
function getPostContents(user: number, post: string): Promise<string> {}
// ->
function xxx(xxx: number, xxx: string): Promise<string> {}
```

If the function signature makes little sense, we can improve it until we see the purpose of the function:

```TypeScript
function xxx(xxx: number, xxx: string): Promise<string> {}
// ->
function xxx(xxx: UserId, xxx: PostSlug): Promise<string> {}
// ->
function xxx(xxx: UserId, xxx: PostSlug): Promise<PostContents> {}

// number -> UserId: the first argument is the user ID;
// string -> PostSlug: the second argument is the publication URL;
// string -> PostContents: the result is the content of the publication.

// From the signature, the function's work becomes clearer:
// we query the content of a particular post using a key
// built from the user ID and the publication URL.
```

Informative signatures make the function meaning clearer and the information density of the code higher. Such signatures carry part of the task context so that we can convey additional knowledge to the reader in the names of functions and arguments:

```TypeScript
function fetchPost(authorId: UserId, post: PostSlug): Promise<PostContents> {}

// getPostContents -> fetchPost: means that the data is requested over the netwc
// userId -> authorId: tells how exactly the user is associated with these posts
```

> **By the way** 📚
>
> Mark Seemann describes the “xxxing out” technique in more detail in “Code That Fits in Your Head.”🔗

We can use the same rule to check if the code follows engineering practices required in the project, for example, the CQS principle:

```TypeScript
class PostReader {
  constructor(private postSource: PostStorage) {}

  getPost(id) {
    this.contents = this.postSource.fetchPost(id);
  }
}

// The signature of the `getPost` method is more like a command than a query:

type GetPost = (id: PostId) => void;

// Perhaps we should rework the API or choose another name for the method:

class PostReader {
  // ...

  readPost(id: PostId): void {
    this.contents = this.postSource.fetchPost(id);
  }
}
```

> **By the way ❌**
>
> The "xxxing out" technique is more helpful in designing public APIs to make them more informative. For non-public functions, it may be a bit less critical.

---

1. "Domain Modeling Made Functional" by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional

2. "Primitive Obsession", Refactoring Guru, https://refactoring.guru/smells/primitive-obsession

3. "Domain-Driven Design" by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design

4. "Ubiquitous Language" by Martin Fowler, https://martinfowler.com/bliki/UbiquitousLanguage.html

5. More on Functions, TypeScript Documentation, https://www.typescriptlang.org/docs/handbook/2/functions.html

6. Type Aliases, TypeScript Handbook, https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#type-aliases

7. "Branding and Type-Tagging" by Kevin B. Greene, https://medium.com/@KevinBGreene/surviving-the-typescript-ecosystem-branding-and-type-tagging-6cf6e516523d

8. Factory Method, Refactoring Guru, https://refactoring.guru/design-patterns/factory-method/typescript/example

9. Type Compatibility, TypeScript Documentation, https://www.typescriptlang.org/docs/handbook/type-compatibility.html

10. Nominal & Structural Typing, Flow Documentation, https://flow.org/en/docs/lang/nominal-structural/

11. "Code That Fits in Your Head" by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head

18

Refactoring Test Code

Tests help us refactor application code by indicating errors we may have made. But the tests are code, too, so we must refactor them from time to time.

In this chapter, we'll talk about how not to break tests during refactoring, how to keep their reliability, and what to pay attention to when searching for problems with the test code.

"Tests" for Tests

A reliable test fails when the code doesn't work as expected. Tests "cover our back" during the refactoring of application code because they will catch an error in the app behavior. The application code, on the other hand, covers the tests because it helps check if they fail for the right reasons.

When tests change along with the application code, we can't check if everything works *as before*. The updated tests may contain bugs or check something *different* from the original functionality, and we might not even notice it.

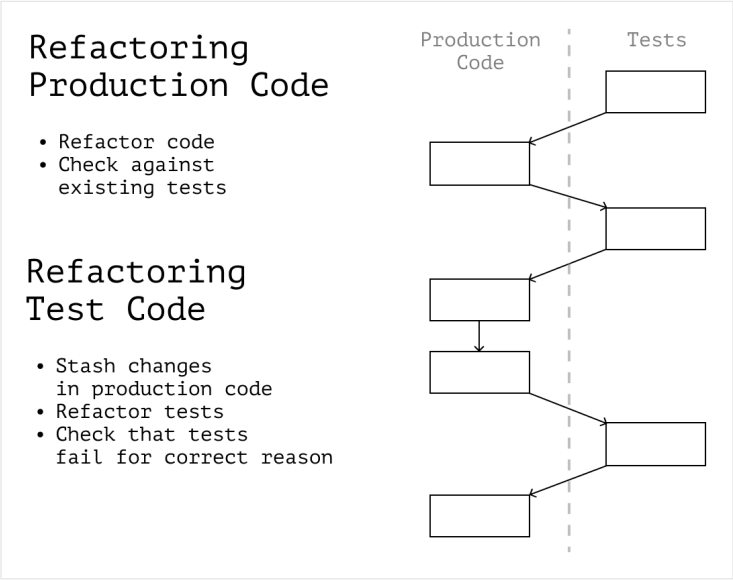
As a result, we might start trusting tests that don't work or work incorrectly. To avoid this, while refactoring test code, we should follow the rule:

> **Alternate refactoring tests and the application code. Avoid doing it at the same time !**

If, during refactoring, we realize that we need to refactor the test code, we should:

- Stash the application code changes from the last commit (using `git stash`)
- Refactor the tests
- Check that they fail for the specified reasons
- Commit test changes
- Unstash the application code changes
- Continue refactoring

This way, we turn the refactoring into "ping-ponging" between refactoring the tests and the application code. They support and cover each other, ensuring the overall program behavior stays the same.



When we change the code, the behavior is captured by the tests. When we change the tests, the application code captures the behavior

This technique doesn't guarantee that we won't make any mistakes but reduces their probability. With it, there's always at least one part (either tests or code) that *hasn't changed since the last moment everything worked*. So we're more confident that everything works as before.

> **By the way** 📚
>
> In "Code That Fits in Your Head,"🔗 Mark Seemann recommends doing the same thing. Besides the technique itself in the book, he also describes how to avoid weakening tests during refactoring.

"Brittle" Tests

Sometimes tests feel "brittle" and unreliable. Most of the time, this happens because of *mocks*. Mocks are demanding about their internal structure, the order and manner in which they're called, and the structure of results they return. In some cases, an application can become "over-mocked"—when mocks replace almost all modules in tests.

In such cases, any changes to the application code, even the smallest ones, result in many updates to the test code. The tests require more resources to support and slow down the

development. This effect is called *test-induced damage*.

Unlike mocks, *stubs and simple test data* help write more change-resistant tests. They're more straightforward in use and forgive far more than they demand. They help us avoid test-induced damage and spend less time updating test code.

> **In detail** 🤓
>
> The difference between stubs, mocks, and other fake objects, is well described by Microsoft. We'll use their terminology in this chapter.

To make the tests less brittle, we can use this heuristic:

> **Use fewer mocks. Make stubs and test data simpler !**

For example, to reduce the number of mocks, we can organize business logic to test it without mocks. It isn't easy to achieve when the logic is mixed with various effects. So it's better to keep effects separated and describe the logic in the form of pure functions.

Pure functions are intrinsically testable. They don't require a fancy test infrastructure and only need the test data and the expected result to be tested.

Let's look at the difference between code where logic and effects are mixed and code where they're separated. Notice how brittle their tests seem:

```JavaScript
// In this function, the logic and effects are mixed:

function fetchPostList(kind) {
  const directory = path.resolve("content", kind);
  const onlyMdx = fs.readDirSync(directory).filter((f) => f.endsWith(".mdx"));
  const postNames = onlyMdx.map((f) => f.replace(/.mdx$/, ""));
  return postNames;
}

// For a unit test of such a function, we need to mock `fs`.
// We need to describe the work of the used method,
// specify the results of that method,
// reset the mock after the test:

it("should return a list of post names with the given kind", () => {
  jest.spyOn(fs, "readDirSync").mockImplementation(() => testFileList);
  const result = fetchPostList("blogPost");
  expect(result).toEqual(expected);
  jest.restoreAllMocks();
});
```

In the second case, the logic and the effects are separated. The data transformation can be tested using only stubs and test data:

```TypeScript
function namesFromFiles(fileList) {
  return fileList
    .filter((f) => f.endsWith(".mdx"))
    .map((f) => f.replace(/.mdx$/, ""));
}

// To test the function, it's enough
// to only have test data and the expected result:

it("should convert file list into a list of post names", () => {
  const result = namesFromFiles(testList);
  expect(result).toEqual(expected);
});
```

The test structure becomes simpler, and updating the test data doesn't take a lot of resources. With this code organization, we can completely abandon static test data and generate it automatically based on predefined properties. Such testing is sometimes called *property-based*.

> **By the way** 🧪
>
> It's usually convenient to use additional tools to generate test data in property-based tests.🔗 In the JavaScript ecosystem, libraries like faker.js,🔗 create objects with random data according to predefined properties.

The effects we separated earlier can be tested by integration tests or E2E tests. Depending on how we organize the work with dependencies, it may be enough to test only adapters to them. In most cases, the complexity of mocks in such tests will be lower.

For example, in tests of such an adapter for `fs`, it's enough to check that the correct method has been called with the required argument:

```TypeScript
function postsByType(kind) {
  const directory = path.resolve("content", kind);
  const fileList = fs.readDirSync(directory);
  return fileList;
}

// We don't need to mock the whole service implementation anymore,
// it's enough just to expose the API similar to the service interface.
// This kind of mock is much more resistant to changes in application code
// and causes less test-induced damage.

describe("when called with a post kind", () => {
  it("should read file list from the correct directory", () => {
    const spy = jest.spyOn(fs, "readDirSync");
    postsByType("blogPost");
    expect(spy).toHaveBeenCalledWith("/content/blogPost/");
  });
});
```

> **In detail** 🧩
>
> More about dependency and effect organization strategies we discussed in the chapters on architecture and side effects.

Then the `fetchPostList` function now becomes a "composition" of logic with effects:

```TypeScript
function fetchPostList(kind) {
  // Read Effect:
  const fileList = postsByType(kind);

  // Pure Logic:
  return namesFromFiles(fileList);
}
```

Such a function may no longer need to be tested by unit tests. Since it combines the functionality of different modules (units), we can think about integration or E2E testing.

Test Duplicates

The test-induced damage slows down development because, after each code change, we have to update the tests. One reason for this slowdown can be tests that check the same functionality multiple times.

Ideally, we want only *one* test to be responsible for a particular unit of behavior. When there're more, we start spending unnecessary time updating them. The more duplicates, the greater the "time tax."

For example, if we wrote an additional unit test for the `fetchPostList` function in the example above, it would most likely be redundant and duplicate the tests of the `postsByType` and `namesFromFiles` functions. Then for every change to those functions, we would need to update not one but two tests.

The duplicate tests could hint at one of several problems:

1. There may indeed be duplication in the application code. It's a reason to perform a review and reduce the duplication in the code.
2. The modules' responsibilities aren't clearly divided, so tests of one module check something already checked by others. It's a signal to define the concerns of different modules more clearly or reconsider the testing strategy to avoid overlapping.

> **Clarification** 🧪
>
> Tests of *different kinds* can sometimes overlap for reliability. An integration test may take over some of the functionality tested by unit tests if it's more convenient to test the application.
>
> I try to keep the number of such overlaps to a minimum, but testing strategies may differ from project to project, so it's difficult to give general recommendations here.

Never-Failing Tests

A test should be responsible for a specific problem and *must fail* when it occurs. If the test never fails, it's harmful: it has no value but takes resources for support. Such a test should be removed or rewritten to fail when the specified problem occurs.

> **By the way** 🙃
>
> Most often, I've encountered never-failing tests in over-mocked systems, where the infrastructure and test arrangement consisted almost entirely of calling mocks. Such tests often pass the result of one mock to another and end up testing nothing.

Tests for Simple Functions

When choosing what and how to test, we should compare the benefits of the test and its costs. For example, we can pay attention to the cyclomatic complexity of the function this test covers.

If the complexity of the function equals one and the test brings more additional work than real benefit, we can abandon the test.[🔗] For example, a separate unit test for the `fullName` function may be unnecessary:

```JavaScript
const fullName = (user) => `${user.firstName} ${user.lastName}`;
```

> **Clarification 🚧**
>
> We're not saying that simple functions don't need tests. The decision whether to test or not depends on the specific situation. The main idea is that if a test brings more costs than benefits, we should consider its necessity.

Regressions

There are cases when simple functions still *need* to be tested though, for example, if a function once had a regression. Regressions pay attention not to potential but actual bugs in the code, which *could and once did happen*.

Anything that comes up in a regression should be covered with tests. If the test seems too simple, and someone might find it useless and delete it, we can add a note in the comment with a link to the regression:

```JavaScript
/**
 * @regression JIRA-420: Users had full names in an incorrect format where the l
 * @see https://some-project.atlassian.com/...
 */
describe("when called with a user object", () => {
  it("should return a full name representation with first name at start", () =>
    const name = fullName(42);
    expect(name).toEqual(expected);
  });
});
```

---

1. "Code That Fits in Your Head" by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
2. "Test-Induced Design Damage" by David Heinemeier Hansson, https://dhh.dk/2014/test-induced-design-damage.html
3. "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing
4. Unit testing best practices with .NET Core and .NET Standard, Microsoft Docs, https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices
5. Faker, Generate fake (but realistic) data for testing and development, https://fakerjs.dev

19

Comments and Documentation

Usually, refactoring only affects the application code and its tests. But sometimes, we'll need to improve comments in the code, project documentation, and development process during refactoring.

In this chapter, we'll talk about how and why to refactor these things. We'll discuss how to look for conflicting sources of information about the project and what to pay attention to in the development process.

Sources of Truth

Work on a project consists of various tasks. We write code and test it, discuss constraints and the domain, clarify the project requirements, etc. In the process, we produce code, documentation, project plans, backlogs, and more.

From the *application* point of view, the only important part is the code. The code directly affects the program execution, and everything else doesn't.

> **By the way** 💬
>
> In JavaScript, some tools use JSDoc comments as signatures,🔗 but these are more *types* than comments. By comments, we'll mean those that don't affect program execution.

On the other hand, developers consider the code and everything "around it." We pay attention to comments, documentation, and code reviews to better understand a particular task and the whole project.

Usually, everything "around the code" gets out of date faster than the code itself. Because of this, information from different sources can be contradictory, and we might doubt what to trust. For example, in the snippet below, the comment conflicts with the function signature:

```JavaScript
/**
 * @param {User} Current user object.
 * @param {Product[]} List of products for the order.
 * @return {Order}
 */
function createOrder(userId, products) {
  /**
   * In the annotation, the first argument is typed as `User`
   * but the implementation seems to use `UserId`.
   * This contradiction questions how the code should really work:
   * - What's correct: the code or the annotation?
   * - What has changed last? Why has that decision been made?
   * - What does the documentation say? What do the product owners say?
   */
}
```

The snippet above offers us two contradictory statements. Such contradictions make it hard to read the code and slow down development. We can't be sure we understand the code correctly until we resolve those contradictions. Such a resolution can take effort and time.

To avoid such problems, we can conduct reviews of comments, documentation, and other sources of information about the project. During these reviews, we should identify the contradictions and resolve them.

Comments

We can use several tools and heuristics to "refactor" comments:

Correct or Delete False Comments

Comments can lie and tell the reader incorrect information. Sometimes the lies can be blatant:

```JavaScript
/** @obsolete Not used anymore across the code base. */
function isEmpty(cart) {}

// ...

// The comment above lies the function _is_ used.
if (!isEmpty(cart)) {
}
```

However, sometimes the lie can be less noticeable. In this case, to see if a suspicious comment is false, we should try to disprove its statement.

To do so, we can ask for help from other developers or documentation. For example, we can find developers who know exactly how the function should work and ask them about the suspicious comment.

When there's no way to contact the team or use documentation, we can conduct a series of experiments on the suspicious code. We'll need to cover this code with tests and observe how they behave when we change the function.

If we're convinced that the comment is false, we should delete it or rewrite it to remove all contradictions. It's important to note this step in the commit message. It's helpful to describe why we suspected the comment was false, and what exactly helped reveal the lie.

> **By the way** 💡
>
> Just like other refactoring techniques, comment refactoring is better done in small steps and separate commits. It helps to describe the context of the problem and the purpose of the changes in the commit message.

Clarify Vague Comments

For vague, inaccurate, or ambiguous comments, we should add details. We can add examples of function calls, links to specific pull requests, tasks, or issues in the task tracker. For example, such a comment isn't constructive:

```JavaScript
// Custom sorting function:
function sort(a, b) {}
```

The fact that the `sort` function "sorts stuff" is clear from its name. Instead, it'd be better to describe what sorting algorithm is used, add examples of how it works, and links to a detailed description of the algorithm:

```JavaScript
/**
 * Implements the most efficient sorting algorithm
 * by comparing electron movement in the circuitry.
 * @see https://wiki.our-project.app/sorting/electron-movement-sort/
 * @example ...
 *
 * @param {Sortable} a
 * @param {Sortable} b
 * @return {CompareResult}
 */
function superFastSort(a, b) {}
```

Inline Small Details in the Code

Comments containing small details can be "inlined" in the types and names of variables, functions, or methods:

```TypeScript
// Fetches post contents by the author's ID.
async function getPost(user) {}

// We can express it like this ↓
async function fetchPostContents(authorId) {}

// Or even like this using types ↓
async function fetchPost(authorId: UserId): Promise<PostContents> {}
```

This technique doesn't always work. It can make the variable or function name too long. In such cases, it's better not to apply the changes.

Describe Context Instead of "Rephrasing Names"

Some comments are unhelpful because they only rephrase the name of the commented entity with different words:

```JavaScript
// Compares strings.
function compareString(a, b) {}
```

In such cases, again, it'd be more beneficial to put what isn't known to the developers in the comment and describe the task's context.

For example, for the `compareString` function, it's better to indicate *why* we use our own implementation of the comparison function. The reason behind the function will convey more details about the problem and the project as a whole:

```JavaScript
/**
 * Implements compare for our limited alphabet with custom diacritic rules.
 * - Required for the correct handling of ...
 * - Justified as a part of R&D in ...
 * @see https://wiki.our-project.app/sorting/custom-diacritic-comparer/
 *
 * @example compareString('a', 'ä') === -1
 * @example compareString('a', 't') === -1
 */
function compareString(a, b) {}
```

Turn TODOs and FIXMEs into Tasks

Sometimes comments contain `TODO` and `FIXME` tags describing bugs or flaws in the code. It's useful to turn the contents of such comments into tickets in the project's task tracker.

Unlike comments, tasks in the tracker are visible to other developers and the rest of the team. By the number of such tasks, we can evaluate the state of the project and estimate the accumulated technical debt.

The description of the created tasks is worth specifying how and why the code works now and what we want to change. Links to the created tasks can be left in a comment next to the code. Then a comment like this:

```JavaScript
// TODO: improve post-conditions.
function ensureMessageSent() {}
```

...Will turn into:

```JavaScript
/**
 * For now, it's unclear what criteria to use for the verification.
 * Update post-conditions when the criteria are known:
 * @see https://our-team.tasktracker.com/our-product/task-42
 */
function ensureMessageSent() {}
```

The most effective strategy for this is to conduct regular reviews, look for tags, and turn them into tasks. Then developers can still create `TODO` and `FIXME` tags in the code while writing code to "stay focused on a task." But then, during regular reviews, these comments will be converted to tickets, keeping the number of such tags under control.

Documentation

The project documentation keeps the history of the application development and the reasons for the technical decisions made. It's useful because it answers developers' questions, but its problem is that it becomes obsolete faster than the code.

The documentation isn't usually integrated into the code but exists separately. Because of this, we have to update it *in addition to changes in the code*. This separation makes it harder to remember to update documentation after the code. It increases the number of discrepancies between the docs and the code.

Updating documentation and code requires a *process*. To keep the documentation from becoming obsolete, we should have a regular task to review it. Once every certain period of time (say, a month), we compare differences between documentation and code and fix them.

Regular tasks limit the size of discrepancies because they don't last long and don't have time to accumulate. When developers periodically "clean up" discrepancies, their negative effect is less.

> **By the way** 📽
>
> To make regular reviews less excessive, we should store in the docs mostly code-agnostic information, which is more related to the domain and the business and is unlikely to change as quickly as the code.
>
> On the other hand, storing architecture-important decisions closer to the code can be more useful. If these decisions affect the organization of the code or the way the application works, it should be easy for developers to refer to them without wasting time.
>
> One approach to dealing with such decisions is known as *Architectural Decision Records, ADR*.🔗

Knowledge Accessibility

It's also helpful to make knowledge about the project and the subject area as *available to the development team* as possible. The easier and quicker developers can find and refer to a piece of knowledge, the more accessible it is.

When developers know about the subject area, they make fewer errors in the application code and find inconsistencies in the project earlier. Accessible knowledge makes development more convenient and thus speeds it up.

Knowledge can be stored in different ways: in documentation, repository metadata, code comments, or the code itself. The accessibility of different options varies. For example, information in the code is most accessible because the code is always at the developer's fingertips. Repository metadata or documentation is less accessible because it exists separately and has to be accessed differently.

To make information more accessible, we can move it "closer to the code" during regular reviews:

- Turn details from comments into variables, functions, and types;
- Turn notes from code reviews into code, documentation, or commit messages;
- Turn insights from "conversations at the cooler" into documentation or issues in the repo or task tracker.

This technique may not work if the project already has a process governing documentation or repository working strategies. But in projects "without a process," the idea of knowledge accessibility can be helpful.

---

1. JSDoc Reference, TypeScript Documentation https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html
2. Architectural Decision Records, ADR, https://adr.github.io

20

Refactoring as a Process

In the previous chapters, we primarily focused on the technical details of refactoring. However, this is only a part of improving the code in a project.

In this chapter, we'll talk about how to find time for refactoring and perform it regularly. We'll discuss how to overcome "friction" in the project and start refactoring even if the code base is large and has a lot of legacy code.

Refactor or Rewrite

When we look at a big chunk of legacy code, the first thought is, "it's easier just to rewrite it from scratch." Sometimes it's true, but it isn't always possible to adequately assess the state of the code at a glance.

Before we decide whether to refactor or rewrite, we should evaluate three things: team resources, the benefits of our solution, and its risks.

These parameters won't give us a direct answer to the question "What to do?" but often, this assessment alone is enough to decide. However, even if it isn't enough, it'll tell us what to investigate before we make a final decision.

Resources

By *resources*, we'll mean the time, knowledge, and experience the team has at its disposal. These resources can be "spent" on refactoring or rewriting code to improve it.

Available Time

To estimate the developers' free time, we can look back at the past experience with the project. We need to assess how much time the team spent on code improvements in the past and how often there were "empty holes" in the project schedule.

This assessment shows how the team spent time on development in the past. It helps predict how much time we'll have in the future.

This prediction may seem an understatement because we may want to spend more time improving the code. However, it reflects the development patterns that the team is used to. These patterns are helpful to consider because even if we agree to spend more time on refactoring, the development will fall back to familiar patterns without a change in the process.

In addition, we should always be aware of "unpredicted problems" with legacy code that will take extra time to solve.

Accumulated Knowledge

Accumulated knowledge about the project can be assessed by the amount of documentation, valuable comments in the code, the quality of the commit history, the availability of the developers who wrote the code, and the clarity of the code itself.

Knowledge is difficult to quantify, so we can use a qualitative assessment. The more contradictions we find between different sources of information, the worse we can consider the quality of the accumulated knowledge. Conversely, the clearer the code and easier it is to find the developers who wrote it, the higher the quality of expertise.

Experience

By experience, we mean the developers' familiarity with this project and proficiency in other projects, languages, and paradigms—the more varied the experience, the less time we'll spend developing poor non-working solutions.

> **By the way** 🤷
>
> It's also worth considering the experience of outside consultants and experts if we want them to participate, but it's harder to do objectively.

Benefits and Risks

We should also define the benefits and risks of refactoring or rewriting the code. They will directly depend on the work processes and the project's inner structure, so we might need internal research.

Project Meta Information

The project's meta information reflects the process of working on the project and what parts of the code are the most important. All the necessary meta information is already there if

the team uses a version control system. For example, the commit history can show us:

- What is a crucial part of the project—what files were updated with the new features and bug fixes the most;
- What implicit coupling exists between project parts—what files were modified along with other files;
- What parts of the project had the most bugs—what commits were related to bug fixes, etc.

Meta information analysis can tell us which parts of the project are the most complex or beneficial from a business perspective.

> **Read more** 📚
>
> Adam Tornhill wrote more about this assessment in the book "Your Code as a Crime Scene."🔗

If we don't have a version control system, we can try to collect indirect indicators: release notes, tech support logs, etc. However, it'll be much harder to conclude from them.

Estimates

If we decide to refactor a particular piece of code, we'll need to evaluate the size of the task and the required time.

To understand how much time a piece of code can take to refactor, we should first point out the places that raise questions. To find out what problems we're dealing with, we can lay out those questions in the following table:

	Simple	**Difficult**
Clear	Research is unnecessary, clear how to solve, and takes minimal time.	Clear how to solve but definitely takes a long time.
Unclear	The problem is small and isolated but might need research.	Definitely need research, lots of hidden connections, unfamiliar area.

The amount of time needed directly depends on the proportion of topics in this table. The more complex and unclear questions we have, the harder it is to plan the development iteration.

Tasks from the last table cell are the most difficult to plan. For such problems, we can suggest the team use the "Hypothesis per Iteration" method. Each assumption about the problem would be a separate development iteration in this method. Disproving or confirming that assumption is the goal of the iteration.

Testing one assumption doesn't take as much time as examining the whole problem. Iterations with such checks are easier to plan and conduct. At the same time, with each tested assumption, we understand more about the problem, and sooner or later, it'll move from the lower right cell to another cell in the table. Then we can plan the development more thoroughly.

> **By the way** 👀
>
> Evaluating the scope and size of the task can be helpful even if the team follows the "No Estimates" methodology.🔗 For example, with the evaluation, it is easier to decide when to change the work strategy without wasting extra resources on the task.

Refactoring Large Chunks of Code

If a piece of code is too big and cannot be refactored all at once, it's helpful to follow the "Strangler Fig" method.

The goal of the approach is *not to replace* a piece of code under refactoring but to implement similar functionality *beside* that code. We can develop this copy considering all the knowledge we now have and all the problems we want to avoid. When the copy is done, we replace the old code with it.

> **By the way** 🌳
>
> The name of this approach comes from the analogy to the strangler fig plant.🔗 This plant winds around a tree, hardens, and when the tree dies, it stays in the shape of that tree. This method of refactoring does the same. We first "cover" a feature with new code and remove the old code inside after it's ready.

When working with the "Strangler Fig" approach, it's important to frequently merge the results into the main branch of the repo. This technique will help prevent merge conflicts in the version control system and keep the refactoring process from taking too much time.

Frequency and Hygiene

In general, it's better to refactor as often as possible. It's most convenient to refactor the code right after making changes to it. The best option is if we can come back to the code after a short rest to take a fresh look at it. This kind of review helps us identify poor solutions in the code sooner.

When working with legacy code, we should refactor the code *before* fixing bugs in it or adding new features. After refactoring, the code will be clearer, better covered by tests, and more pleasant to work with. Thus, we'll reduce the time needed to fix a bug or add a feature.

Finally, when touching any code, it's worth remembering the "Boy-Scout Rule":

> **Leave the code behind in a better state than we found it !**

Metrics

Although refactoring relies on subjective measures such as beauty and readability, we can still use quantitative metrics to assess the changes' quality.

As the basis, we'll take the metrics from the talk "Where does bad code come from?" and expand the list. As a result, we'll get a list of 7 metrics, each of which answers the question, "How long does it take to XXX?"

- **S**earch, how long it takes to find the required place in the code.
- **W**rite, to write a new feature and cover it with tests.
- **A**gree, to get developers to agree that "code is good enough."
- **R**ead, to read and understand what a piece of code does.
- **M**odify, to modify the code to fit the new requirements.
- **E**xecute, to execute/build/deploy a piece of code.
- **D**ebug, to find and fix a bug.

If we measure these metrics before and after refactoring, we can compare the quality of the changes we've made. Some metrics are still quite subjective but can be expressed in numbers. A quantitative description of characteristics will help us to notice a trend of improvement or deterioration when estimating code quality.

---

1. "Your Code As a Crime Scene" by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene
2. "No Estimates" by Allen Holub, https://youtu.be/QVBlnCTu9Ms
3. "Strangler Fig Application" by Martin Fowler https://martinfowler.com/bliki/StranglerFigApplication.html
4. "Opportunistic Refactoring" by Martin Fowler https://martinfowler.com/bliki/OpportunisticRefactoring.html
5. "Code That Fits in Your Head" by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
6. "Where Does Bad Code Come From?" https://youtu.be/7YpFGkG-u1w

21

Afterword

Thank you for reading this book! I hope you find it helpful, and I would love to hear your feedback and ideas.

Errata and Feedback

If you found a typo or an error, please, open an issue in the project's repository or send them by email. I'll also be glad to hear your ideas and code snippets that can make the examples in the book more descriptive.

You can find previous fixes and updates in the commit history of the project's repository.

For Publishers

This book is a large draft of a book on refactoring that I would like to write in the future.

In the future book, I would like to combine fragmented code examples into a single application. It would help to show how to refactor not various pieces of code but whole projects. In my opinion, it would be more demonstrative and helpful.

Refactoring such a project would cover all the techniques mentioned in this book combined under the shared context of a single application. It would make it possible to see how the different refactoring techniques support each other and help work toward a common result.

If you'd like to publish such a book, please, contact me by mail. I'd be happy to discuss the details!

For Translators

This book is available in 2 languages: Russian and English. If you are interested in translating it into other languages, please, contact me by mail.

Acknowledgements

Thanks to Maksim Ivanov (@satansdeer),[3] Nick Lopin (@nlopin),[4] and Artem Samofalov (@dex157)[5] for help in proofreading and fact-checking!

Thanks to all project contributors for finding typos, improving phrasing, and other help with the text![6]

Extras

In the appendices to this book, you'll find a list of links to books, articles, talks, and tools that I refer to in my work and can recommend.

You'll also find a list of shortened recommendations for finding and fixing code problems based on the topics of the previous chapters.

---

1. Refactor Like a Superhero, Book Repository, https://github.com/bespoyasov/refactor-like-a-superhero-online-book
2. Contact Email, bespoyasov@me.com
3. Maksim Ivanov, https://github.com/satansdeer
4. Nikolai Lopin, https://github.com/nlopin
5. Artem Samofalov, https://github.com/dex157
6. Project Contributors, https://github.com/bespoyasov/refactor-like-a-superhero-online-book/graphs/contributors

22

Cheat Sheet on Refactoring Techniques

This appendix contains a list of short recommendations for finding and fixing problems with the code.

The recommendations are based on the topics from the previous chapters. Before applying them, it's worth reading the chapters to assess whether these techniques fit your project, style, and coding habits.

It isn't a list of strict rules. Instead, it's a set of tips you can consider.

Searching for Problems

- Search for code smells in the project.
- Pay attention to feelings like "hard to read, change, test."

Before Refactoring

- Define the boundaries of the changes and find the "seams" in the code.
- Cover the selected part of the code with tests.
- Describe as many edge cases as possible with tests.
- Set up automatic test runner and code linter.
- Mark the compiler and linter warnings as errors.

During Refactoring

- Use small steps and atomic commits.
- Create compact but detailed pull requests.
- Apply just one refactoring technique at a time.
- Fix bugs and add features separately from refactoring.
- Respect the transformation priority premise.
- Do not mix refactoring of tests and application code.

- Apply the automated refactoring tools in IDE.
- Look at diffs to find inconsistencies before committing changes.
- Run all changes through tests, even the smallest ones.

Where to Start

- Implement automatic code formatting.
- Test the code with linters and static analyzers.
- Replace self-written helpers with language and environment features.

Improving Names

- Clarify obscure names.
- Decipher undocumented abbreviations.
- Decompose entities with long names.
- Use different names for different entities.
- Use ubiquitous language.
- Fix lying names.

Dealing with Code Duplication

- Separate duplication from the lack of information about the system.
- Conduct regular reviews of duplicated code.
- Extract repetitive data into variables.
- Extract repetitive actions in functions.

Abstraction as a Tool

- Describe intent in function names.
- Structure code to give information that is needed to the reader at a particular moment.
- Remember the limits of humans' working memory; limit the number of entities in the function.
- Split information by levels of abstraction.
- Decompose tasks based on the data used in them.
- Make sure the function has only one reason to change.
- Ensure that the module guarantees the validity of its data itself.

Linear Code Execution

- Explicitly express the data states in the application.
- Make it difficult to pass invalid data in code.

- Validate the data before use.
- Use selectors to “prepare” data for different use cases.
- Set a limit on the cyclomatic and cognitive complexity of the code.
- “Straighten” the conditions:
    - Use the early return;
    - Apply de Morgan’s laws;
    - Use predicates for dynamic conditions;
    - Use design patterns (e.g. Strategy, Null-object);
    - Apply pattern-matching when possible.

Working with Side Effects

- Use pure functions more often.
- Aim for referential transparency for easier debugging.
- Treat data and code as immutable and stateless by default.
- Move side effects to the edges of the module, function, or use case.
- Test the simplicity of code with tests: the easier it is to write tests, the simpler the code is.
- Write adapters to reduce coupling and create fewer mocks in tests.
- Separate effects into commands and queries.
- Identify CQS violations by type signatures and function names.
- Use different models to read and write data when appropriate.

Handling Errors

- Identify different types of errors in the application.
- Centralize error handling.
- Get to error handling as early as possible.
- Try to express possible errors in the function signature.
- Check input data before using it.
- Use logging and analytics in the “last resort” handlers.
- Use decorators for composing cross-cutting concerns.

Integrating Application Parts

- Keep coupling low and cohesion high.
- Identify cohesion by input data, output data, and code dependencies.
- Designate module guarantees through its public API.
- Make module communication less coupled:
    - Use messages or events when appropriate;
    - Apply patterns (e.g., Observer).

- Limit the number of module dependencies.
- Compose data transformations instead of side effects.
- Separate logic from side effects.
- Separate data from behavior.

Working with Generics

- Do not rush to generalize.
- Compose complex types from simple types instead of using inheritance.
- Use generics when you are sure that the type structure will not change.
- Pay attention to conditions; they can indicate a violation of the Liskov substitution principle.

Architecture and Communication with the World

- Use ubiquitous language to model the domain.
- Express the stages of the application data lifecycle in their states.
- Try not to depend directly on third-party code.
- Add an anti-corruption layer where the data format or API may change.
- Separate UI logic from business logic.
- Separate configuration from code.

Static Typing as a Tool

- Use types to describe the domain and the business workflows.
- Make invalid data states unrepresentable in types.
- Identify CQS violations by method and function signatures.
- Apply the "xxxing" technique to increase the information density of signatures and function names.

Refactoring Tests

- Do not mix refactoring tests and application code.
- Use more stubs and test data and fewer mocks.
- Get rid of test duplicates and never-failing tests.

Refactoring Documentation and Comments

- Get rid of conflicts between the code and documentation.
- Clarify vague comments: add context, examples, and reasons behind the current implementation.
- Conduct regular reviews of comments, documentation, and project issues.

- Evaluate resources, benefits, and risks when choosing between refactoring or rewriting from scratch.
- Use project meta information from the version control system to analyze project health.
- Use the “Strangler Fig” technique when refactoring large chunks of code.
- Refactor code regularly.
- Remember the “Boy-Scout Rule.”
- Rely on measurable metrics when evaluating improvements.

23

Sources and References

My opinion of modern development and good code has been influenced by other people's work. In this appendix, I have prepared a list of books, articles, talks, methodologies, and studies that I find most useful and refer to in my own work.

Each section in this list has a title similar to the chapter topics. If you are interested in the topic of a particular chapter and want to know more, the sections in the list will help you navigate more easily among the links.

The links can repeat. I decided it was more important to compile a complete list for each topic than to get rid of duplicates. I hope you find this useful.

Notion of Refactoring and Code Quality

What refactoring is and why it is needed. What an uncontrollably increasing complexity of a project can lead to. How to define "bad" and "good" code.

Books

- "The Art of Readable Code" by Dustin Boswell, Trevor Foucher, https://www.goodreads.com/book/show/8677004-the-art-of-readable-code
- "Beyond Legacy Code" by David Scott Bernstein, https://www.goodreads.com/book/show/26088456-beyond-legacy-code
- "The Black Swan" by Nassim Nicholas Taleb, https://www.goodreads.com/book/show/242472.The_Black_Swan
- "Clean Code" by Robert C. Martin, https://www.goodreads.com/book/show/3735293-clean-code
- "Refactoring" by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936.Refactoring
- "Refactoring", 2nd edition, by Martin Fowler, https://www.goodreads.com/book/show/35135772-refactoring

- “Refactoring JavaScript” by Evan Burchard, https://www.goodreads.com/book/show/39331294-refactoring-javascript
- “Software Design: Cognitive Aspect” by Françoise Détienne, https://www.goodreads.com/book/show/3104497-software-design-cognitive-aspect
- “Working Effectively with Legacy Code” by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Talks and Video

- “7 Ineffective Coding Habits of Many Programmers” by Kevlin Henney, https://youtu.be/ZsHMHuklIJY
- “Log4J & JNDI Exploit: Why So Bad?” https://youtu.be/Opqgwn8TdlM
- “Preventing the Collapse of Civilization” by Jonathan Blow, https://youtu.be/pW-SOdj4Kkk
- “Where Does Bad Code Come From?” https://youtu.be/7YpFGkG-u1w

Blog Posts, Studies, and Examples

- “Beauty Is in Simplicity”, by Jørn Ølmheim, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_05/
- Code Readability Testing, an Empirical Study, https://www.researchgate.net/publication/299412540_Code_Readability_Testing_an_Empi
- Evaluating Code Readability and Legibility: An Examination of Human-centric Studies, https://github.com/reydne/code-comprehension-review
- “The Human Cost of Tech Debt”, by Erik Dietrich, https://daedtech.com/human-cost-tech-debt/
- How Readable Code Is, https://howreadable.com
- “Refactoring, Changing the Code without Changing its External Behavior” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Refactoring
- “Technical Debt”, DevIQ, https://deviq.com/terms/technical-debt

Related Concepts

- Bus factor, Wikipedia, https://en.wikipedia.org/wiki/Bus_factor
- Entropy, Wikipedia, https://en.wikipedia.org/wiki/Entropy
- Murphy’s Law, Wikipedia, https://en.wikipedia.org/wiki/Murphy%27s_law

Tools

- Code Smells, Refactoring Guru, https://refactoring.guru/refactoring/smells
- Refactoring Techniques, Refactoring Guru, https://refactoring.guru/refactoring/techniques

Before Start

What to look for before refactoring. How to prepare the code for changes in order to simplify the work. How to secure the work with future changes.

Books

- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Debug It!: Find, Repair, and Prevent Bugs in Your Code” by Paul Butcher, https://www.goodreads.com/book/show/6770868-debug-it
- “Refactoring” by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936.Refactoring
- “Thinking, Fast and Slow” by Daniel Kahneman, https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow
- “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing
- “Willpower Doesn’t Work” by Benjamin P. Hardy, https://www.goodreads.com/book/show/35604684-willpower-doesn-t-work
- “Working Effectively with Legacy Code” by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code
- “Your Code As a Crime Scene” by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene

Talks and Video

- “The Grand Unified Theory of Clean Architecture and Test Pyramid” by Guilherme Ferreira, https://youtu.be/gHSpj2zM9Nw
- “How do you prepare before tackling a problem?” by Fun-fun-function, https://youtu.be/mF-tVjXbO8Y

Blog Posts and Examples

- “Before You Refactor” by Rajith Attapattu, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_06/
- “How to ask good questions” by Julia Evans, https://jvns.ca/blog/good-questions/
- “Read Code” by Karianne Berg, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_70/
- “Refactoring, Changing the Code without Changing its External Behavior” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Refactoring
- “TDD: What, How, and Why” by Alex Bespoyasov, https://bespoyasov.me/blog/tdd-what-how-and-why/

Related Concepts

- Agile Software Development, https://en.wikipedia.org/wiki/Agile_software_development
- Working memory, Capacity, Wikipedia, https://en.wikipedia.org/wiki/Working_memory#Capacity

Tools

- Extreme Programming, http://www.extremeprogramming.org
- Test-Driven Development, TDD, https://martinfowler.com/bliki/TestDrivenDevelopment.html
- Tools for Better Thinking, https://untools.co

During Refactoring

What to avoid during refactoring, how to make the process easier. How to isolate changes and make sure no other code is broken. How to stay within resource budget and keep changes small.

Books

- “Beyond Legacy Code” by David Scott Bernstein, https://www.goodreads.com/book/show/26088456-beyond-legacy-code
- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Jedi Techniques” by Maxim Dorofeev, Translated Summary, https://bespoyasov.me/blog/jedi-techniques/
- “Refactoring” by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936.Refactoring
- “Refactoring”, 2nd edition, by Martin Fowler, https://www.goodreads.com/book/show/35135772-refactoring
- “Refactoring JavaScript” by Evan Burchard, https://www.goodreads.com/book/show/39331294-refactoring-javascript
- “Rules of Work Communication” by M. Ilyahov and L. Sarycheva, Translated Summary, https://bespoyasov.me/blog/rules-of-work-communication/
- “Thinking, Fast and Slow” by Daniel Kahneman, https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow
- “Working Effectively with Legacy Code” by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Talks and Video

- “7 Ineffective Coding Habits of Many Programmers” by Kevlin Henney, https://youtu.be/ZsHMHukIUY

- “All the Little Things” by Sandi Metz, https://youtu.be/8bZh5LMaSmE
- “Where Does Bad Code Come From?” https://youtu.be/7YpFGkG-u1w

Blog Posts and Examples

- “Antipatterns as a Worst Practices” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Antipatterns
- “Convenience Is not an -ility” by Gregor Hohpe, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_19/
- “Deploy Early and Often” by Steve Berczuk, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_20/
- “Don’t Push: Automate Instead” by Alex Bespoyasov, https://bespoyasov.me/blog/do-not-push-automate-instead/
- “How to Get Your Code Reviewed Faster” by Artem Sapegin, https://blog.sapegin.me/all/faster-code-reviews/
- “Use Git Tactically” by Mark Seeman, https://stackoverflow.blog/2022/04/06/use-git-tactically/
- “Write Better Commits, Build Better Projects” by Victoria Dye, https://github.blog/2022-06-30-write-better-commits-build-better-projects/

Related Concepts

- Agile Software Development, https://en.wikipedia.org/wiki/Agile_software_development
- Atomic Commit, Wikipedia https://en.wikipedia.org/wiki/Atomic_commit
- Continuous Integration, Wikipedia, https://en.wikipedia.org/wiki/Continuous_integration
- Decomposition, Wikipedia, https://en.wikipedia.org/wiki/Decomposition_(computer_science)
- Transformation Priority Premise, Wikipedia https://en.wikipedia.org/wiki/Transformation_Priority_Premise

Tools

- Continuous Integration, Wikipedia, https://en.wikipedia.org/wiki/Continuous_integration
- Getting Started, About Version Control, https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control
- Oh Shit, Git!?! https://ohshitgit.com
- Transformation Priority Premise, Wikipedia https://en.wikipedia.org/wiki/Transformation_Priority_Premise
- Use binary search to find the commit that introduced a bug, https://git-scm.com/docs/git-bisect

Formatters, Linters, and Language Features

How to use all the capabilities of automated code refactoring tools and analyzers. Reasons to know the peculiarities of the language and environment in which the code is executed. How to benefit from automation. How consistency helps you to solve problems more quickly.

Books

- “The Art of Readable Code” by Dustin Boswell, Trevor Foucher, https://www.goodreads.com/book/show/8677004-the-art-of-readable-code
- “Autopilot: The Art & Science of Doing Nothing” by Andrew Smart, https://www.goodreads.com/book/show/18053732-autopilot
- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Refactoring JavaScript” by Evan Burchard, https://www.goodreads.com/book/show/39331294-refactoring-javascript
- “Thinking, Fast and Slow” by Daniel Kahneman, https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow

Talks and Video

- “7 Ineffective Coding Habits of Many Programmers” by Kevlin Henney, https://youtu.be/ZsHMHukIUY
- “Refactoring with Cognitive Complexity” by G. Ann Campbell, https://youtu.be/el9OKGrqU6o

Blog Posts and Studies

- “Automate Your Coding Standard” by Filip van Laenen, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_04/
- “Automatic semicolon insertion in JavaScript” by Dr. Axel Rauschmayer, https://2ality.com/2011/05/semicolon-insertion.html
- “Code Layout Matters” by Steve Freeman, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_13/
- How Readable Code Is, https://howreadable.com
- “How to Convert HTML Form Field Values to a JSON Object” by Jason Lengstorf, https://www.learnwithjason.dev/blog/get-form-values-as-json
- “Know Your IDE” by Heinz Kabutz, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_45/
- “Know Your Next Commit” by Dan Bergh Johnsson, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_47/
- “Use Git Tactically” by Mark Seeman, https://stackoverflow.blog/2022/04/06/use-git-tactically/

- “Use the Right Algorithm and Data Structure” by JC van Winkel, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_89/
- “Why robots should format our code for us”, https://blog.sapegin.me/all/prettier/

Tools

- Can I Use, support tables for web, https://caniuse.com
- List of EcmaScript Proposals, https://proposals.es
- Prettier, an opinionated code formatter, https://prettier.io
- Refactoring Source Code in VSCode, https://code.visualstudio.com/docs/editor/refactoring

Naming

Why naming is important, how variable and function names affect code perception and development speed. Why terminology synchronization improves team collaboration. How to identify “bad” and “good” names. What to do with lying names.

Books

- “Clean Code” by Robert C. Martin, https://www.goodreads.com/book/show/3735293-clean-code
- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Domain Modeling Made Functional” by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional
- “Domain-Driven Design” by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design
- “Refactoring” by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936.Refactoring
- “Refactoring”, 2nd edition, by Martin Fowler, https://www.goodreads.com/book/show/35135772-refactoring
- “Software Design: Cognitive Aspect” by Françoise Détienne, https://www.goodreads.com/book/show/3104497-software-design-cognitive-aspect
- “Thinking, Fast and Slow” by Daniel Kahneman, https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow
- “Working Effectively with Legacy Code” by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Talks and Video

- “7 Ineffective Coding Habits of Many Programmers” by Kevlin Henney, https://youtu.be/ZsHMHuklUY
- “All the Little Things” by Sandi Metz, https://youtu.be/8bZh5LMaSmE
- “Evolutionary Software Architectures” by Neal Ford, https://youtu.be/CglSFhwbI3s
- “Transforming Code into Beautiful, Idiomatic Python” https://youtu.be/OSGv2VnC0go

Blog Posts, Studies, and Examples

- “Antipatterns as a Worst Practices” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Antipatterns
- “Code in the Language of the Domain” by Dan North, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_11/
- “Coding with Clarity” by Brandon Gregory, https://alistapart.com/article/coding-with-clarity/
- Evaluating Code Readability and Legibility: An Examination of Human-centric Studies, https://github.com/reydne/code-comprehension-review/blob/master/list-papers/AllPhasesMergedPapers-Part1.md
- “Give it five minutes” by Jason Fried, https://signalvnoise.com/posts/3124-give-it-five-minutes
- “How to ask good questions” by Julia Evans, https://jvns.ca/blog/good-questions/
- “Read Code” by Karianne Berg, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_70/
- “Refactoring, Changing the Code without Changing its External Behavior” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Refactoring
- “Stop using isLoading booleans” by Kent C. Dodds, https://kentcdodds.com/blog/stop-using-isloading-booleans

Related Concepts

- Abstraction layer, Wikipedia, https://en.wikipedia.org/wiki/Abstraction_layer
- Bus factor, Wikipedia, https://en.wikipedia.org/wiki/Bus_factor
- Declarative Programming, Wikipedia, https://en.wikipedia.org/wiki/Declarative_programming
- Decomposition, Wikipedia, https://en.wikipedia.org/wiki/Decomposition_(computer_science)
- Entropy, Wikipedia, https://en.wikipedia.org/wiki/Entropy
- Post Hoc Ergo Propter Hoc, Wikipedia, https://en.wikipedia.org/wiki/Post_hoc_ergo_propter_hoc

Tools

- Conventional Commits, https://www.conventionalcommits.org/en/v1.0.0/
- Naming Cheat Sheet, https://github.com/kettanaito/naming-cheatsheet

- Refactoring Source Code in VSCode, https://code.visualstudio.com/docs/editor/refactoring
- Refactoring Techniques, Refactoring Guru, https://refactoring.guru/refactoring/techniques
- Ubiquitous Language, https://martinfowler.com/bliki/UbiquitousLanguage.html

Code Duplication

How to distinguish between code duplication and lack of knowledge about the system. Why and how to use duplication as a tool. The benefits of regular code audits and how to get into the habit of doing them.

Books

- "The Art of Readable Code" by Dustin Boswell, Trevor Foucher, https://www.goodreads.com/book/show/8677004-the-art-of-readable-code
- "Jedi Techniques" by Maxim Dorofeev, Translated Summary, https://bespoyasov.me/blog/jedi-techniques/
- "Refactoring" by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936.Refactoring
- "Your Code As a Crime Scene" by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene
- "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Blog Posts and Examples

- "Antipatterns as a Worst Practices" by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Antipatterns
- "Beware the Share" by Udi Dahan, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_07/
- "Copypaste in Code" by Alex Bespoyasov, https://bespoyasov.me/blog/copy-paste/
- "How to ask good questions" by Julia Evans, https://jvns.ca/blog/good-questions/
- "Refactoring, Changing the Code without Changing its External Behavior" by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Refactoring
- "The Single Responsibility Principle" by Robert C. Martin, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_76/
- "When You SHOULD Duplicate Code" by Marius Bongarts, https://medium.com/@mariusbongarts/when-you-should-duplicate-code-b0d747bc1c67

Related Concepts

- Don't Repeat Yourself, Wikipedia https://ru.wikipedia.org/wiki/Don't_repeat_yourself
- Separation of Concerns, Wikipedia, https://en.wikipedia.org/wiki/Separation_of_concerns

Tools

- Copy/paste detector, jscpd, https://www.npmjs.com/package/jscpd
- Detect copy-pasted and structurally similar code, jsinspect, https://github.com/danielstjules/jsinspect
- Duplicate Code Smell, Refactoring Guru, https://refactoring.guru/smells/duplicate-code

Abstraction and Separation of Concerns

How and why to use abstraction. Reasons to separate intention from implementation and consider the limits of the working memory of the human brain. How to give the reader information about the system in a controlled way. How to efficiently decompose complex tasks into simpler ones. How to make sure the application data is always in the correct state.

Books

- "And Suddenly the Inventor Appeared: Triz, the Theory of Inventive Problem Solving" by Genrich Altshuller, https://www.goodreads.com/book/show/161916.And_Suddenly_the_Inventor_Appeared
- "Code That Fits in Your Head" by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- "Designing Data-Intensive Applications" by Martin Kleppmann https://dataintensive.net
- "Domain Modeling Made Functional" by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional
- "The Humane Interface" by Jef Raskin, https://www.goodreads.com/book/show/344726.The_Humane_Interface
- "Thinking, Fast and Slow" by Daniel Kahneman, https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow
- "Software Design: Cognitive Aspect" by Françoise Détienne, https://www.goodreads.com/book/show/3104497-software-design-cognitive-aspect
- "Structure and Interpretation of Computer Programs" by Harold Abelson, Gerald Jay Sussman, Julie Sussman, https://www.goodreads.com/book/show/43713.Structure_and_Interpretation_of_Computer
- "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

- "You Don't Know JS Yet: Scope & Closures" by Kyle Simpson, https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/scope-closures/ch8.md
- "Your Code As a Crime Scene" by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene

Talks and Video

- "7 Ineffective Coding Habits of Many Programmers" by Kevlin Henney, https://youtu.be/ZsHMHuklUY
- "All the Little Things" by Sandi Metz, https://youtu.be/8bZh5LMaSmE
- "Category Theory in Life" by Eugenia Cheng, https://youtu.be/ho7oagHeqNc
- "Design, Composition, and Performance" by Rich Hickey, https://youtu.be/MCZ3YgeEUPg
- "Functional Design Patterns" by Scott Wlaschin, https://youtu.be/srQt1NAHYC0

Blog Posts, Studies, and Examples

- "Antipatterns as a Worst Practices" by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Antipatterns
- "Climbing the infinite ladder of abstraction" by Alexis King, https://lexi-lambda.github.io/blog/2016/08/11/climbing-the-infinite-ladder-of-abstraction/
- "Coupling, Cohesion & Connascence" by Khalil Stemmler, https://khalilstemmler.com/wiki/coupling-cohesion-connascence/
- "Declarative Data Validation with Rule-Based Approach and Functional Programming" by Alex Bespoyasov, https://bespoyasov.me/blog/declarative-rule-based-validation/
- "Maintain a Single Layer of Abstraction at a Time" by Khalil Stemmler, https://khalilstemmler.com/articles/oop-design-principles/maintain-a-single-layer-of-abstraction/
- "Missing Abstraction" by Alex Bespoyasov, https://bespoyasov.me/blog/missing-abstraction/
- "Out of the Tar Pit", by Ben Moseley and Peter Marks, https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf
- "The Power of Composition" by Scott Wlaschin, https://fsharpforfunandprofit.com/composition/
- "Prefer Domain-Specific Types to Primitive Types" by Einar Landre, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_65/

Related Concepts

- Abstraction, Wikipedia, https://en.wikipedia.org/wiki/Abstraction
- Abstraction layer, Wikipedia, https://en.wikipedia.org/wiki/Abstraction_layer
- Cohesion, Wikipedia, https://en.wikipedia.org/wiki/Cohesion_(computer_science)
- Coupling, Wikipedia, https://en.wikipedia.org/wiki/Coupling_(computer_programming)
- Decomposition, Wikipedia, https://en.wikipedia.org/wiki/Decomposition_(computer_science)

- Encapsulation, Wikipedia, https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)
- Separation of Concerns, Wikipedia, https://en.wikipedia.org/wiki/Separation_of_concerns
- Working memory, Capacity, Wikipedia, https://en.wikipedia.org/wiki/Working_memory#Capacity

Tools

- Single Responsibility Principle, Principles of OOD, http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- Tiny Types in TypeScript, https://janmolak.com/tiny-types-in-typescript-4680177f026e
- Tools for Better Thinking, https://untools.co
- What is Connascence? https://connascence.io
- TRIZ, Wikipedia, https://en.wikipedia.org/wiki/TRIZ

Functional Pipeline and Linear Code Execution

How and why to express data states of business workflows in code. The benefits of linear code execution are. How to disallow passing invalid data to functions and isolate data validation. The benefits of functional programming are.

Books

- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Domain-Driven Design” by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design
- “Domain Modeling Made Functional” by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional
- “Refactoring JavaScript” by Evan Burchard, https://www.goodreads.com/book/show/39331294-refactoring-javascript
- “Thinking, Fast and Slow” by Daniel Kahneman, https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow
- “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing

Talks and Video

- “CQRS and Event Sourcing” by Greg Young, https://youtu.be/JHGkaShoyNs
- “Is Domain-Driven Design Overrated?” by Stefan Tilkov, https://youtu.be/ZZp9RQEGeqQ
- “Functional architecture: The pits of success” by Mark Seemann, https://youtu.be/US8QG9I1XW0

- “Functional Design Patterns” by Scott Wlaschin, https://youtu.be/srQt1NAHYC0
- “Professor Frisby Introduces Composable Functional JavaScript” by Brian Lonsdorf, https://egghead.io/courses/professor-frisby-introduces-composable-functional-javascript
- “Refactoring with Cognitive Complexity” by G. Ann Campbell, https://youtu.be/el9OKGrqU6o
- “Why Your Architecture is Functional” by Roman Nevolin, RU with Subtitles, https://youtu.be/9s_4wpzENhg

Blog Posts

- “Apply Functional Programming Principles” by Edward Garson, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_02/
- “Constructive vs predicative data” by Hillel Wayne, https://www.hillelwayne.com/post/constructive/
- “Design a microservice domain model”, MSDN, https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/microservice-domain-model
- “Designing with types: Making illegal states unrepresentable” by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/designing-with-types-making-illegal-states-unrepresentable/
- “Functional design is intrinsically testable” by Mark Seemann, https://blog.ploeh.dk/2015/05/07/functional-design-is-intrinsically-testable/
- “Immutability: Making your code predictable” by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/correctness-immutability/
- “Lenses in Functional Programming” by Albert Steckermeier, https://sinusoid.es/misc/lager/lenses.pdf
- “Making Illegal States Unrepresentable” by Hillel Wayne, https://buttondown.email/hillelwayne/archive/making-illegal-states-unrepresentable/
- “Parse, don’t validate” by Alexis King, https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/
- “The Power of Composition” by Scott Wlaschin, https://fsharpforfunandprofit.com/composition/

Related Concepts

- Data Mapper, https://martinfowler.com/eaaCatalog/dataMapper.html
- Data Transfer Object, DTO, Wikipedia, https://en.wikipedia.org/wiki/Data_transfer_object
- Declarative Programming, Wikipedia, https://en.wikipedia.org/wiki/Declarative_programming
- Function Composition, Wikipedia, https://en.wikipedia.org/wiki/Function_composition
- Functional Programming, Wikipedia, https://en.wikipedia.org/wiki/Functional_programming

- Immutable Object, Wikipedia, https://en.wikipedia.org/wiki/Immutable_object
- Predicate, Wikipedia, https://en.wikipedia.org/wiki/Predicate_(mathematical_logic)
- Projection operations (C#), https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/projection-operations
- Pure Functions, Wikipedia, https://en.wikipedia.org/wiki/Pure_function

Tools

- Bounded Context in DDD by Martin Fowler, https://www.martinfowler.com/bliki/BoundedContext.html
- Design Patterns, Refactoring Guru, https://refactoring.guru/design-patterns
- Domain Model, https://martinfowler.com/eaaCatalog/domainModel.html
- Specification for interoperability of common algebraic structures in JavaScript, Fantasy Land, https://github.com/fantasyland/fantasy-land
- Typed functional programming in TypeScript, fp/ts, https://github.com/gcanti/fp-ts
- The library which provides useful monads, interfaces, and lazy iterators, sweet-monads, https://github.com/JSMonk/sweet-monads
- Zen of Python, https://peps.python.org/pep-0020/

Conditions and Complexity

How to organize the conditions to decrease the cognitive load of the code. Metrics to use to measure complexity. How to use automated tools to manage complexity. Reasons to "straighten" code execution and "turn" conditions inside out. How to use Boolean algebra to simplify conditions. Design patterns that can help do this. How to apply functional programming principles to simplify conditions.

Books

- "The Art of Readable Code" by Dustin Boswell, Trevor Foucher, https://www.goodreads.com/book/show/8677004-the-art-of-readable-code
- "Code That Fits in Your Head" by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- "Debug It!: Find, Repair, and Prevent Bugs in Your Code" by Paul Butcher, https://www.goodreads.com/book/show/6770868-debug-it
- "Domain Modeling Made Functional" by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional
- "Refactoring" by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936.Refactoring
- "Refactoring", 2nd edition, by Martin Fowler, https://www.goodreads.com/book/show/35135772-refactoring
- "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing

- “Working Effectively with Legacy Code” by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code
- “Your Code As a Crime Scene” by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene

Talks and Video

- “Functional architecture: The pits of success” by Mark Seemann, https://youtu.be/US8QG9I1XW0
- “Maybe Not” by Rich Hickey, https://youtu.be/YR5WdGrpoug
- “Refactoring with Cognitive Complexity” by G. Ann Campbell, https://youtu.be/el9OKGrqU6o
- “Why Your Architecture is Functional” by Roman Nevolin, RU with Subtitles, https://youtu.be/9s_4wpzENhg

Blog Posts, Studies, and Examples

- “Anti-if, the Missing Patterns” by Joe Wright, https://code.joejag.com/2016/anti-if-the-missing-patterns.html
- “Antipatterns as a Worst Practices” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Antipatterns
- “Application State Management with Finite State Machines” by Alex Bespoyasov, https://bespoyasov.me/blog/fsm-to-the-rescue/
- “Bringing Pattern Matching to TypeScript” by Gabriel Vergnaud, https://dev.to/gvergnaud/bringing-pattern-matching-to-typescript-introducing-ts-pattern-v3-0-o1k
- “Cognitive Complexity. A new way of measuring understandability” by G. Ann Campbell, SonarSource SA, https://www.sonarsource.com/docs/CognitiveComplexity.pdf
- “A conditional sandwich example” by Mark Seemann, https://blog.ploeh.dk/2022/02/14/a-conditional-sandwich-example/
- “Decoupling decisions from effects” by Mark Seemann, https://blog.ploeh.dk/2016/09/26/decoupling-decisions-from-effects/
- “Destroy all `if`s” by John A De Goes, https://degoes.net/articles/destroy-all-ifs
- “Functional design is intrinsically testable” by Mark Seemann, https://blog.ploeh.dk/2015/05/07/functional-design-is-intrinsically-testable/
- “Out of the Tar Pit”, by Ben Moseley and Peter Marks, https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf
- “Parse, don’t validate” by Alexis King, https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/
- “Refactoring, Changing the Code without Changing its External Behavior” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Refactoring
- “Why should you return early?” by Szymon Krajewski https://szymonkrajewski.pl/why-should-you-return-early/

Related Concepts

- “Cognitive Complexity. A new way of measuring understandability” by G. Ann Campbell, SonarSource SA, https://www.sonarsource.com/docs/CognitiveComplexity.pdf
- Control flow graph & cyclomatic complexity for following procedure, Stackoverflow, https://stackoverflow.com/a/2670135/3141337
- Control-Flow Graph, Wikipedia, https://en.wikipedia.org/wiki/Control-flow_graph
- Cyclomatic Complexity, Wikipedia, https://en.wikipedia.org/wiki/Cyclomatic_complexity
- Pattern Matching, Wikipedia, https://en.wikipedia.org/wiki/Pattern_matching

Tools

- `complexity`, ES Lint, https://eslint.org/docs/latest/rules/complexity
- De Morgan’s Laws, Wikipedia, https://en.wikipedia.org/wiki/De_Morgan%27s_laws
- Design Patterns, Refactoring Guru, https://refactoring.guru/design-patterns/
- ECMAScript Pattern Matching Proposal, https://github.com/tc39/proposal-pattern-matching
- `switch-exhaustiveness-check`, ES Lint TypeScript, https://typescript-eslint.io/rules/switch-exhaustiveness-check/
- `ts-pattern`, Library for Pattern Matching in TypeScript, https://github.com/gvergnaud/ts-pattern

Working with Side Effects

Reasons why side effects make the program more complex and unpredictable. How to reduce the number of effects in your code and what to do with effects needed for the application to work. The benefits of pure functions and referential transparency. Options to test effects and reasons to separate logic from effects. The point of dividing code into commands and queries.

Books

- “Clean Architecture” by Robert C. Martin, https://www.goodreads.com/book/show/18043011-clean-architecture
- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Domain Modeling Made Functional” by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional
- “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing
- “Working Effectively with Legacy Code” by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Talks and Video

- “CQRS and Event Sourcing” by Greg Young, https://youtu.be/JHGkaShoyNs
- “Functional architecture: The pits of success” by Mark Seemann, https://youtu.be/US8QG9I1XW0
- “Why Your Architecture is Functional” by Roman Nevolin, RU with Subtitles, https://youtu.be/9s_4wpzENhg

Blog Posts and Studies

- “Command-Query Responsibility Segregation” by Martin Fowler, https://martinfowler.com/bliki/CQRS.html
- “Command-Query Separation” by Martin Fowler, https://martinfowler.com/bliki/CommandQuerySeparation.html
- “Command-Query Separation” by Alex Bespoyasov, https://bespoyasov.me/blog/commands-and-queries/
- “CQS versus server generated IDs” by Mark Seemann, https://blog.ploeh.dk/2014/08/11/cqs-versus-server-generated-ids/
- “Functional architecture is Ports and Adapters” by Mark Seemann, https://blog.ploeh.dk/2016/03/18/functional-architecture-is-ports-and-adapters/
- “Functional Core in Imperative Shell” by Gary Bernhardt, https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell
- “Impureim Sandwich” by Mark Seemann, https://blog.ploeh.dk/2020/03/02/impureim-sandwich/
- “Message Passing Leads to Better Scalability in Parallel Systems” by Russel Winder, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_57/
- “Out of the Tar Pit”, by Ben Moseley and Peter Marks, https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf
- “A pipe operator for JavaScript” by Axel Rauschmayer, https://2ality.com/2022/01/pipe-operator.html

Related Concepts

- Design Patterns, Refactoring Guru, https://refactoring.guru/design-patterns
- Immutable Object, Wikipedia, https://en.wikipedia.org/wiki/Immutable_object
- Pure Functions, Wikipedia, https://en.wikipedia.org/wiki/Pure_function
- Referential Transparency, Haskell Wiki, https://wiki.haskell.org/Referential_transparency

Error Handling

Kinds of errors exist and how they differ. Problems entangled error handling lead to. What to pay attention to when refactoring error handling in JavaScript code. Techniques to use when there are technology, paradigm, or methodology constraints.

Books

- "Code That Fits in Your Head" by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- "Domain Modeling Made Functional" by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional
- "Patterns for Fault Tolerant Software" by Robert Hanmer, https://www.goodreads.com/book/show/3346135-patterns-for-fault-tolerant-software
- "The Pragmatic Programmer" by Andy Hunt, https://www.goodreads.com/book/show/4099.The_Pragmatic_Programmer
- "What I've Learned From Failure" by Reg Braithwaite, https://leanpub.com/shippingsoftware/read

Talks and Video

- "Error handling: doing it right!" by Ruben Bridgewater, https://youtu.be/bJ3glfA-jqo
- "Functional Design Patterns" by Scott Wlaschin, https://youtu.be/srQt1NAHYC0
- "Maybe Not" by Rich Hickey, https://youtu.be/YR5WdGrpoug

Blog Posts

- "Against Railway-Oriented Programming" by by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/against-railway-oriented-programming/
- "Dealing with Unhandled Exceptions", by Alexander Zlatkov https://blog.sessionstack.com/how-javascript-works-exceptions-best-practices-for-synchronous-and-asynchronous-environments-39f66b59f012#ecc9
- "Declarative Data Validation with Rule-Based Approach and Functional Programming" by Alex Bespoyasov, https://bespoyasov.me/blog/declarative-rule-based-validation/
- "Distinguish Business Exceptions from Technical" by Dan Bergh Johnssonm https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_21/
- "Don't Ignore that Error!" by Pete Goodliffe, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_26/
- "The Error Model" by Joe Duffy, http://joeduffyblog.com/2016/02/07/the-error-model/
- "Errors Are Not Exceptions" by swyx, https://www.swyx.io/errors-not-exceptions
- "Functional Error Handling with Express.js and DDD | Enterprise Node.js + TypeScript" by Khalil Stemmler, https://khalilstemmler.com/articles/enterprise-typescript-nodejs/functional-error-handling/

- “GraphQL error handling to the max with Typescript, codegen and fp-ts” by Ghislain Thau, https://www.the-guild.dev/blog/graphql-error-handling-with-fp
- “A Monad in Practicality: First-Class Failures” by Quil, https://robotlolita.me/articles/2013/a-monad-in-practicality-first-class-failures/
- “A mostly complete guide to error handling in JavaScript” by Valentino Gagliardi, https://www.valentinog.com/blog/error/
- “Notification” by Martin Fowler, https://martinfowler.com/eaaDev/Notification.html
- “Parse, don’t validate” by Alexis King, https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/
- “Railway Oriented Programming” by Scott Wlaschin, https://fsharpforfunandprofit.com/rop/
- “Replacing Throwing Exceptions with Notification in Validations” by Martin Fowler, https://martinfowler.com/articles/replaceThrowWithNotification.html
- “Stop catching errors in TypeScript; Use the Either type to make your code predictable” by Anthony Manning-Franklin, https://antman-does-software.com/stop-catching-errors-in-typescript-use-the-either-type-to-make-your-code-predictable
- “Using Results in TypeScript” by Dan Imhoff, https://imhoff.blog/posts/using-results-in-typescript
- “When life gives you lemons, write better error messages” by Jenni Nadler, https://wix-ux.com/when-life-gives-you-lemons-write-better-error-messages-46c5223e1a2f
- “Why should you return early?” by Szymon Krajewski https://szymonkrajewski.pl/why-should-you-return-early/

Related Concepts

- Cross-Cutting Concern, Wikipedia, https://en.wikipedia.org/wiki/Cross-cutting_concern
- Fail-fast, Wikipedia, https://en.wikipedia.org/wiki/Fail-fast

Tools

- Decorator Pattern, Refactoring Guru https://refactoring.guru/design-patterns/decorator
- Error Boundaries in React, https://reactjs.org/docs/error-boundaries.html
- `pipe`, fp/ts, https://gcanti.github.io/fp-ts/modules/function.ts.html#pipe
- `neverthrow`, Type-Safe Errors for JS & TypeScript, https://github.com/supermacro/neverthrow
- `sweet-monads`, Easy-to-use monads implementation with static types definition, https://github.com/JSMonk/sweet-monads
- `switch-exhaustiveness-check`, ES Lint TypeScript, https://typescript-eslint.io/rules/switch-exhaustiveness-check/
- `true-myth`, A library for safer and smarter error- and “nothing”-handling in TypeScript, https://github.com/true-myth/true-myth

Module Integration

Coupling and cohesion. How to divide an application into modules and then compose these modules together. Why and how to decompose tasks. The benefits of contracts and guarantees between modules are. How to decouple modules from each other as much as possible but still leave room for them to communicate. The difference between object and functional composition is. How to manage dependencies. How to achieve data integrity and consistency.

Books

- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Debug It!: Find, Repair, and Prevent Bugs in Your Code” by Paul Butcher, https://www.goodreads.com/book/show/6770868-debug-it
- “Dependency Injection in .NET” by Mark Seemann, https://www.goodreads.com/book/show/9407722-dependency-injection-in-net
- “Domain-Driven Design” by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design
- “Domain Modeling Made Functional” by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional
- “Refactoring” by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936.Refactoring
- “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing
- “Your Code As a Crime Scene” by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene
- “Working Effectively with Legacy Code” by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Talks and Video

- “7 Ineffective Coding Habits of Many Programmers” by Kevlin Henney, https://youtu.be/ZsHMHuklUY
- “All the Little Things” by Sandi Metz, https://youtu.be/8bZh5LMaSmE
- “Functional architecture: The pits of success” by Mark Seemann, https://youtu.be/US8QG9I1XW0
- “Functional Design Patterns” by Scott Wlaschin, https://youtu.be/srQt1NAHYC0
- “Refactoring with Cognitive Complexity” by G. Ann Campbell, https://youtu.be/el9OKGrqU6o
- “Why Your Architecture is Functional” by Roman Nevolin, RU with Subtitles, https://youtu.be/9s_4wpzENhg

Blog Posts and Examples

- "Antipatterns as a Worst Practices" by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Antipatterns
- "Dependency Injection Using the Reader Monad" by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/dependencies-3/
- "Dependency Injection with TypeScript in Practice" by Alex Bespoyasov, https://bespoyasov.me/blog/di-ts-in-practice/
- "Dependency Interpretation" by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/dependencies-4/
- "Dependency Rejection" by Mark Seemann, https://blog.ploeh.dk/2017/02/02/dependency-rejection/
- "Evans Classification" by Martin Fowler, https://martinfowler.com/bliki/EvansClassification.html
- "Inversion of Control Containers and the Dependency Injection pattern" by Martin Fowler, https://martinfowler.com/articles/injection.html
- "Six approaches to dependency injection" by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/dependencies/

Related Concepts

- Cohesion, Wikipedia, https://en.wikipedia.org/wiki/Cohesion_(computer_science)
- Coupling, Wikipedia, https://en.wikipedia.org/wiki/Coupling_(computer_programming)
- Design By Contract, https://wiki.c2.com/?DesignByContract
- Message Broker, https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648849(v=pandp.10)
- Message Bus, https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff647328(v=pandp.10)
- Message Queue, Wikipedia, https://en.wikipedia.org/wiki/Message_queue
- Separation of Concerns, Wikipedia, https://en.wikipedia.org/wiki/Separation_of_concerns

Tools

- Design Patterns, Refactoring Guru, https://refactoring.guru/design-patterns/
- React software design patterns, https://github.com/themithy/react-design-patterns
- Reactive Extensions Library for JavaScript, RxJS, https://rxjs.dev
- REpresentational State Transfer, REST, https://restfulapi.net
- What is Connascence? https://connascence.io

Generics and Hierarchies

How to understand when you need a generalized algorithm or type. Why composition is preferable to inheritance in most cases. How to use the Liskov substitution principle as an

integration linter.

Talks and Video

- "Category Theory in Life" by Eugenia Cheng, https://youtu.be/ho7oagHeqNc
- "The Grand Unified Theory of Clean Architecture and Test Pyramid" by Guilherme Ferreira, https://youtu.be/gHSpj2zM9Nw
- "Functional Design Patterns" by Scott Wlaschin, https://youtu.be/srQt1NAHYC0

Blog Posts

- "A behavioral notion of subtyping" by Barbara H. Liskov, Jeannette M. Wing, https://dl.acm.org/doi/10.1145/197320.197383
- "Coupling, Cohesion & Connascence" by Khalil Stemmler, https://khalilstemmler.com/wiki/coupling-cohesion-connascence/
- "The Ins and Outs of Generic Variance in Kotlin" by Dave Leeds, https://typealias.com/guides/ins-and-outs-of-generic-variance/
- "Maintain a Single Layer of Abstraction at a Time" by Khalil Stemmler, https://khalilstemmler.com/articles/oop-design-principles/maintain-a-single-layer-of-abstraction/
- "The Power of Composition" by Scott Wlaschin, https://fsharpforfunandprofit.com/composition/
- "Why variance matters" by Ted Kaminski https://www.tedinski.com/2018/06/26/variance.html

Related Concepts

- Covariance and Contravariance, Wikipedia, https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)
- Design By Contract, https://wiki.c2.com/?DesignByContract
- Generics in TypeScript, https://www.typescriptlang.org/docs/handbook/2/generics.html
- The Liskov Substitution Principle, https://web.archive.org/web/20151128004108/http://www.objectmentor.com/resources/a
- Types and Typeclasses, http://learnyouahaskell.com/types-and-typeclasses

Application Architecture

How poor architecture can hamper refactoring. How to use ubiquitous language to improve the architecture. How to build interaction with the outside world and manage dependencies. How ports and adapters are useful. Reasons to separate UI-logic from business-logic. How architecture affects testability.

Books

- “Clean Architecture” by Robert C. Martin, https://www.goodreads.com/book/show/18043011-clean-architecture
- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Designing Data-Intensive Applications” by Martin Kleppmann https://dataintensive.net
- “Domain-Driven Design” by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design
- “Domain Modeling Made Functional” by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional
- “Enterprise Integration Patterns” by Gregor Hohpe, https://www.goodreads.com/book/show/85012.Enterprise_Integration_Patterns
- “Software Architecture in Practice” by L. Bass, P. Clements, R. Kazman, https://www.goodreads.com/book/show/70143.Software_Architecture_in_Practice
- “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing
- “What I’ve Learned From Failure” by Reg Braithwaite, https://leanpub.com/shippingsoftware/read
- “Your Code As a Crime Scene” by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene

Talks and Video

- “CQRS and Event Sourcing” by Greg Young, https://youtu.be/JHGkaShoyNs
- “Design, Composition, and Performance” by Rich Hickey, https://youtu.be/MCZ3YgeEUPg
- “Evolutionary Software Architectures” by Neal Ford, https://youtu.be/CgISFhwbI3s
- “Is Domain-Driven Design Overrated?” by Stefan Tilkov, https://youtu.be/ZZp9RQEGeqQ
- “The Grand Unified Theory of Clean Architecture and Test Pyramid” by Guilherme Ferreira, https://youtu.be/gHSpj2zM9Nw
- “Functional architecture: The pits of success” by Mark Seemann, https://youtu.be/US8QG9I1XW0
- “Functional Design Patterns” by Scott Wlaschin, https://youtu.be/srQt1NAHYC0
- “Why Your Architecture is Functional” by Roman Nevolin, RU with Subtitles, https://youtu.be/9s_4wpzENhg

Blog Posts

- “DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together” by Herberto Graça, https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/
- “Functional Architecture is Ports and Adapters” by Mark Seemann, https://blog.ploeh.dk/2016/03/18/functional-architecture-is-ports-and-adapters/

- “Functional Core in Imperative Shell” by Gary Bernhardt, https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell
- “How to ask good questions” by Julia Evans, https://jvns.ca/blog/good-questions/
- “Impureim Sandwich” by Mark Seemann, https://blog.ploeh.dk/2020/03/02/impureim-sandwich/
- “The pedantic checklist for changing your data model in a web application” by Raphael Gaschignard, https://rtpg.co/2021/06/07/changes-checklist.html
- “Ports & Adapters Architecture” by Herberto Graça, https://herbertograca.com/2017/09/14/ports-adapters-architecture/
- “The Software Architecture Chronicles” by Herberto Graça, https://herbertograca.com/2017/07/03/the-software-architecture-chronicles/
- “Test-Induced Design Damage” by David Heinemeier Hansson, https://dhh.dk/2014/test-induced-design-damage.html
- “View Models”, Reactive UI, https://www.reactiveui.net/docs/handbook/view-models/

Related Concepts

- Model-View-ViewModel, Wikipedia, https://en.wikipedia.org/wiki/Model–view–viewmodel
- List of System Quality Attributes, Wikipedia, https://en.wikipedia.org/wiki/List_of_system_quality_attributes
- Reactive Programming, Wikipedia, https://en.wikipedia.org/wiki/Reactive_programming
- Software Requirements, Wikipedia, https://en.wikipedia.org/wiki/Software_requirements
- Ubiquitous Language, https://martinfowler.com/bliki/UbiquitousLanguage.html

Tools

- Anti-corruption Layer Pattern, https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer
- Every Programmer Should Know, Architecture, https://github.com/mtdvio/every-programmer-should-know#architecture
- Feature-Sliced Design, Architectural methodology for frontend projects, https://feature-sliced.design
- Ubiquitous Language, https://martinfowler.com/bliki/UbiquitousLanguage.html

Declarative Style

The benefits of a declarative code style over an imperative one. Situations when to prefer the imperative style. Why finite state machines can be useful in frontend development.

Books

- “Antifragile: Things That Gain from Disorder” by Nassim Nicholas Taleb, https://www.goodreads.com/book/show/13530973-antifragile
- “The Art of Readable Code” by Dustin Boswell, Trevor Foucher, https://www.goodreads.com/book/show/8677004-the-art-of-readable-code
- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Your Code As a Crime Scene” by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene

Talks and Video

- “Design, Composition, and Performance” by Rich Hickey, https://youtu.be/MCZ3YgeEUPg
- “Designing Declarative APIs” by Ilya Birman, https://youtu.be/uCQ3JFuQ7bQ
- “Goodbye, useEffect” by David Khourshid, https://youtu.be/HPoC-k7Rxwo
- “Metaprogramming and Multiparadigm Programming” by Timur Shemsedinov, RU with EN Slides, https://youtu.be/Bo9y4IxdNRY
- “Refactoring with Cognitive Complexity” by G. Ann Campbell, https://youtu.be/el9OKGrqU6o
- “Type Level Programming in TypeScript” by Gabriel Vergnaud, https://youtu.be/vGVvJuazs84

Blog Posts

- “Application State Management with Finite State Machines” by Alex Bespoyasov, https://bespoyasov.me/blog/fsm-to-the-rescue/
- “Declarative Data Validation with Rule-Based Approach and Functional Programming” by Alex Bespoyasov, https://bespoyasov.me/blog/declarative-rule-based-validation/
- “Maintain a Single Layer of Abstraction at a Time” by Khalil Stemmler, https://khalilstemmler.com/articles/oop-design-principles/maintain-a-single-layer-of-abstraction/
- “Prefer Domain-Specific Types to Primitive Types” by Einar Landre, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_65/
- “Simplicity Comes from Reduction” by Paul W. Homer, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_75/

Related Concepts

- Transformation Priority Premise, Wikipedia https://en.wikipedia.org/wiki/Transformation_Priority_Premise
- Finite State Machine, Wikipedia, https://en.wikipedia.org/wiki/Finite-state_machine
- Pattern Matching, Wikipedia, https://en.wikipedia.org/wiki/Pattern_matching

Tools

- 12 Factor Apps, https://12factor.net
- JavaScript and TypeScript finite state machines and statecharts, XState, https://github.com/statelyai/xstate
- Zen of Python, https://peps.python.org/pep-0020/

Static Typing

How to use types to convey more knowledge about the domain. How to make invalid data states unrepresentable in code. How to use types to detect development principles violation.

Books

- "Code That Fits in Your Head" by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- "Domain Modeling Made Functional" by Scott Wlaschin, https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional
- "Domain-Driven Design" by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design

Talks and Video

- "Category Theory in Life" by Eugenia Cheng, https://youtu.be/ho7oagHeqNc
- "Type Level Programming in TypeScript" by Gabriel Vergnaud, https://youtu.be/vGVvJuazs84

Blog Posts

- "Branding and Type-Tagging" by Kevin B. Greene, https://medium.com/@KevinBGreene/surviving-the-typescript-ecosystem-branding-and-type-tagging-6cf6e516523d
- "Bringing Pattern Matching to TypeScript" by Gabriel Vergnaud, https://dev.to/gvergnaud/bringing-pattern-matching-to-typescript-introducing-ts-pattern-v3-0-o1k
- "Code in the Language of the Domain" by Dan North, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_11/
- "Constructive vs predicative data" by Hillel Wayne, https://www.hillelwayne.com/post/constructive/
- "Designing with types: Making illegal states unrepresentable" by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/designing-with-types-making-illegal-states-unrepresentable/
- "Making Illegal States Unrepresentable" by Hillel Wayne, https://buttondown.email/hillelwayne/archive/making-illegal-states-unrepresentable/

- “Prefer Domain-Specific Types to Primitive Types” by Einar Landre, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_65/

Related Concepts

- Nominal & Structural Typing, Flow Documentation, https://flow.org/en/docs/lang/nominal-structural/
- Ubiquitous Language, https://martinfowler.com/bliki/UbiquitousLanguage.html
- Type Aliases, TypeScript Handbook, https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#type-aliases
- Type Compatibility, TypeScript Documentation, https://www.typescriptlang.org/docs/handbook/type-compatibility.html

Tools

- Tiny Types in TypeScript, https://janmolak.com/tiny-types-in-typescript-4680177f026e
- `switch-exhaustiveness-check`, ES Lint TypeScript, https://typescript-eslint.io/rules/switch-exhaustiveness-check/

Refactoring Test Code

How not to break tests during refactoring. What makes tests brittle.” How to find a balance between high coverage and low test-induced damage.

Books

- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing
- “Working Effectively with Legacy Code” by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Talks and Video

- “7 Ineffective Coding Habits of Many Programmers” by Kevlin Henney, https://youtu.be/ZsHMHuklUY

Blog Posts and Examples

- “Choosing properties for property-based testing” by Scott Wlaschin, https://fsharpforfunandprofit.com/posts/property-based-testing-2/

- “Refactoring, Changing the Code without Changing its External Behavior” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Refactoring
- “TDD: What, How, and Why” by Alex Bespoyasov, https://bespoyasov.me/blog/tdd-what-how-and-why/
- “Test-Induced Design Damage” by David Heinemeier Hansson, https://dhh.dk/2014/test-induced-design-damage.html
- “Unit testing best practices with .NET Core and .NET Standard”, MSDN, https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

Tools

- Faker, Generate fake (but realistic) data for testing and development, https://fakerjs.dev
- Test-Driven Development, TDD, https://martinfowler.com/bliki/TestDrivenDevelopment.html
- `git stash`, Stash the changes in a dirty working directory away, https://git-scm.com/docs/git-stash

Everything Around the Code

How to synchronize the sources of information in a project. How and why to make comments more informative. How to increase the benefits of documentation without increasing the cost of maintaining it.

Books

- “Beyond Legacy Code” by David Scott Bernstein, https://www.goodreads.com/book/show/26088456-beyond-legacy-code
- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Envisioning Information” by Edward R. Tufte, https://www.goodreads.com/book/show/17745.Envisioning_Information
- “The Newspaper Designer’s Handbook” by Tim Harrower, https://www.goodreads.com/book/show/61881153-the-newspaper-designer-s-handbook

Blog Posts and Studies

- “Copypaste in Code” by Alex Bespoyasov, https://bespoyasov.me/blog/copy-paste/
- “How to ask good questions” by Julia Evans, https://jvns.ca/blog/good-questions/
- “Information Density and Linguistic Encoding”, by Matthew W. Crocker, Vera Demberg, Elke Teich https://www.researchgate.net/publication/283938800_Information_Density_and_Linguisti

- “Use Git Tactically” by Mark Seeman, https://stackoverflow.blog/2022/04/06/use-git-tactically/
- “When life gives you lemons, write better error messages” by Jenni Nadler, https://wix-ux.com/when-life-gives-you-lemons-write-better-error-messages-46c5223e1a2f
- “Write Better Commits, Build Better Projects” by Victoria Dye, https://github.blog/2022-06-30-write-better-commits-build-better-projects/

Tools

- Architectural Decision Records, ADR, https://adr.github.io
- Every Programmer Should Know, Practices, https://github.com/mtdvio/every-programmer-should-know#practices
- JSDoc Reference, TypeScript Documentation https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html

Refactoring as a Process

How to decide whether to refactor or rewrite the code. The information to collect about the project before the start. How to estimate the time required for a task. Metrics to use to measure the effect of refactoring on the code.

Books

- “The Art of Readable Code” by Dustin Boswell, Trevor Foucher, https://www.goodreads.com/book/show/8677004-the-art-of-readable-code
- “Beyond Legacy Code” by David Scott Bernstein, https://www.goodreads.com/book/show/26088456-beyond-legacy-code
- “Code That Fits in Your Head” by Mark Seemann, https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head
- “Refactoring” by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936.Refactoring
- “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov, https://www.goodreads.com/book/show/48927138-unit-testing
- “Working Effectively with Legacy Code” by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code
- “Write Better Commits, Build Better Projects” by Victoria Dye, https://github.blog/2022-06-30-write-better-commits-build-better-projects/
- “Your Code As a Crime Scene” by Adam Tornhill, https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene

Talks and Video

- “All the Little Things” by Sandi Metz, https://youtu.be/8bZh5LMaSmE
- “Evolutionary Software Architectures” by Neal Ford, https://youtu.be/CglSFhwbI3s

- “No Estimates” by Allen Holub, https://youtu.be/QVBlnCTu9Ms
- “Preventing the Collapse of Civilization” by Jonathan Blow, https://youtu.be/pW-SOdj4Kkk
- “Where Does Bad Code Come From?” https://youtu.be/7YpFGkG-u1w

Blog Posts and Examples

- “Antipatterns as a Worst Practices” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Antipatterns
- “The Boy Scout Rule” by Robert C. Martin, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_08/
- “Strangler Fig Application” by Martin Fowler https://martinfowler.com/bliki/StranglerFigApplication.html
- “Opportunistic Refactoring” by Martin Fowler https://martinfowler.com/bliki/OpportunisticRefactoring.html
- “When to Refactor” by Mark Seemann, https://blog.ploeh.dk/2022/09/19/when-to-refactor/
- “Refactoring, Changing the Code without Changing its External Behavior” by Timur Shemsedinov, https://github.com/HowProgrammingWorks/Refactoring

Tools

- Every Programmer Should Know, Engineering Philosophy, https://github.com/mtdvio/every-programmer-should-know#engineering-philosophy
- Every Programmer Should Know, Practices, https://github.com/mtdvio/every-programmer-should-know#practices