Get Your Hands Dirty on Clean Architecture

A **Hands-on Guide** to
Creating Clean Web Applications
with Code Examples in Java

Tom Hombergs

2ND EDITION

Get Your Hands Dirty on Clean Architecture (2nd edition)

A Hands-on Guide to Creating Clean Web Applications with Code Examples in Java

Tom Hombergs

This book is for sale at http://leanpub.com/get-your-hands-dirty-on-clean-architecture

This version was published on 2023-07-14

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Tom Hombergs by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Interested in a Hands-on discussion on building a Clean/Hexagonal Architecture? Have a look at the book "Get Your Hands Dirty on Clean Architecture" by @TomHombergs.

The suggested hashtag for this book is #get-your-hands-dirty.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#get-your-hands-dirty

Contents

Foreword

It's a developer's paradise: easy to test the domain logic, easy to mock out infrastructure and technology, crystal-clear separation of domain code and technical code, and even migrating from one technology to another seems easy. No more endless discussion about which part of your code you implement this tricky new feature in that the business people need by tomorrow. It's called "clean architecture," and Tom will guide you on your journey toward this.

For a few years, the foundation of clean architecture has been documented under various names (hexagonal architecture, ports and adapters, onion architecture, and clean architecture). The basic idea looks simple: two concentric circles separating domain stuff and technical stuff within the software. Dependencies flow inward, from technology to the domain. Domain classes are not allowed any dependencies upon technical classes.

Too bad that most of the original sources missed out on explaining how packages and code should be organized. Tom's book perfectly fills this gap. He uses an illustrative example to guide you toward a highly maintainable and clear architectural structure.

Do yourself and your development colleagues a favor and give the clean architecture approach a chance. I promise you won't regret it!

Gernot Starke, Cologne, June 2023
Pragmatic software architect since the 1990's, founder of arc42, co-founder of iSAQB, nerd.

Preface

If you have picked up this book, you care about the architecture of the software you're building. You want your software to not only fulfill the customer's explicit requirements but also the hidden requirement of maintainability and your own requirements concerning structure and aesthetics.

It's hard to fulfill these requirements because software projects (or projects in general, for that matter) usually don't go as planned. Managers draw deadlines all around the project team[1], external partners build their APIs differently from what they had promised, and software products we depend on don't work as expected.

And then there is our own software architecture. It was so nice in the beginning. Everything was clear and beautiful. Then the deadlines pressed us into taking shortcuts. Now, the shortcuts are all that's left of the architecture and it takes longer and longer to deliver new features.

Our shortcut-driven architecture makes it hard to react to an API that had to be changed because an external partner screwed up. It seems easier to just send our project manager into battle with that partner to tell them to deliver the API we had agreed upon.

Now we have given up all control over the situation. In all likelihood, one of the following things will happen:

- The project manager is not strong enough to win the battle against the external partner.
- The external partner finds a loophole in the API specs, proving them right.
- The external partner needs another *<enter number here>* months to fix the API.

All of this leads to the same result: we have to change our code quickly because the deadline is looming.

We add another shortcut.

Instead of letting external factors govern the state of our software architecture, this book takes the stance of taking control ourselves. We gain this control by creating an architecture that *makes the software soft*, as in "flexible", "extensible" and "adaptable". Such an architecture will make it easy to react to external factors and take a lot of pressure off our backs.

What is the goal of this book?

I wrote this book because I was disappointed with the practicality of the resources available on domain-centric architecture styles such as Robert C. Martin's "Clean Architecture" and Alistair

[1]The word "deadline" presumably originates from the 19th century and described a line drawn around a prison or a camp of prisoners. A prisoner that crossed that line was shot. Think about this definition the next time someone "draws a deadline" around you … it will certainly open up new perspectives.

Cockburn's "Hexagonal Architecture". Many books and online resources explain valuable concepts but not how we can actually implement them.

That's probably because there is more than one way to implement any architecture style.

With this book, I am trying to fill this void by providing a hands-on-code discussion about creating a web application in the Hexagonal Architecture or "Ports and Adapters" style. In order to live up to that goal, the code examples and concepts discussed in this book provide *my interpretation* of how to implement a Hexagonal Architecture. There are certainly other interpretations out there, and I do not claim mine to be authoritative.

I certainly hope, however, that you will get some inspiration from the concepts in this book, so that you can create your own interpretation of Hexagonal / Clean Architecture.

Who should read this book?

This book is aimed at software developers of all experience levels involved in creating web applications, but many of the aspects discussed are also valuable for other types of applications.

As a junior developer, you'll learn about how to design software components and complete applications in a clean and maintainable manner. You will also learn some arguments for when to apply a certain technique. You should, however, have participated in building a web application in the past to get the most out of this book.

If you're an experienced developer, you'll enjoy comparing the concepts from the book with your own way of doing things and, hopefully, incorporating bits and pieces into your own software development style.

The code examples in this book are in Java and Kotlin, but all discussions are equally applicable to other object-oriented programming languages. If you're not a Java programmer but can read object-oriented code in other languages, you'll be fine. In the few places where we need some Java or framework specifics, I will explain them.

The example application

To have a recurrent theme throughout the book, most of the code examples show code from an example web application for transferring money online. We'll call it "BuckPal".[2]

The BuckPal application allows a user to register an account, transfer money between accounts, and view the activities (deposits and withdrawals) on the account.

I'm not a finance specialist by any means, so please don't judge the example code based on legal or functional correctness. Rather, judge it on structure and maintainability.

[2] A quick online search has revealed that a company named PayPal has stolen my idea and even copied part of the name. Joking aside: try to find a name similar to "PayPal" that is not the name of an existing company. It's hilarious!

The curse of example applications for software engineering books and online resources is that they're too simple to highlight the real-world problems we struggle with every day. On the other hand, an example application must stay simple enough to effectively convey the discussed concepts.

I hope to have found a balance between "too simple" and "too complex" as we discuss use cases of the BuckPal application throughout this book.

The code of the example application can be found on GitHub.[3]

Print version

If you enjoy reading a good old paper book, you might be interested in the print version of this book. It's available at Amazon and other major online book stores. Have a look at https://reflectoring.io/book[4] for links and other news about the book.

Feedback

If you have anything to say about this book, I'd love to hear it! Get in touch with me via email to tom@reflectoring.io[5] or on Twitter via @TomHombergs[6].

Also feel free to create a review on Amazon[7] or on Goodreads[8] to let other readers know what you think about the book.

Since this book is self-published, I have the freedom to create as many releases as I wish, so I invite you to take part by giving feedback.

Notes about the second edition

The first edition of this book was way more successful than I would have hoped. It has reached almost 8,000 readers at the time of writing and I'm still getting feedback, questions, and comments from engaged readers regularly. For a self-published book that I just wrote to understand the topic of "Clean" and "Hexagonal" architecture, this was totally unexpected. Thanks to all the readers who encouraged me to make this a better book.

With this much engagement in the reader community, and some questions not addressed by the first edition of this book, a second edition is the natural next step. Let me walk you through what has changed in the second edition.

[3] https://github.com/thombergs/buckpal
[4] https://reflectoring.io/book
[5] mailto:tom@reflectoring.io
[6] https://twitter.com/TomHombergs
[7] https://www.amazon.com/Hands-Dirty-Clean-Architecture-hands-ebook/dp/B07YFS3DNF/
[8] https://www.goodreads.com/book/show/47938519-get-your-hands-dirty-on-clean-architecture

The book is now introduced by the new Chapter 1, Maintainability, which advertises maintainability as a main driver for successful software architecture. This chapter hopefully gives a better motivation about why we should care about software architecture in the first place.

The fine-grained package structure I originally proposed in Chapter 4, Organizing Code, of the first edition felt a bit over-engineered and implied that we can manage multiple domains by putting each domain into its own hexagon, which is not the intent of Hexagonal Architecture, so I simplified it in this edition and added a reference to the new chapter about managing bounded contexts.

In Chapter 8, Testing Architecture Elements, in the section about system tests, I have now included a discussion of the inherent testability of Hexagonal Architecture by replacing input and output adapters with mock adapters. At the same time, I advise against doing this in most circumstances.

Many first edition readers asked me was how to manage multiple domains / bounded contexts with Hexagonal Architecture. The new Chapter 13, Managing Multiple Bounded Contexts, now addresses this question with a dissatisfying answer, I'm afraid …

With Chapter 14, A Component-Based Approach to Software Architecture, the book now offers a light-weight alternative to Hexagonal Architecture, which might later evolve into Hexagonal Architecture (or not).

I hope the second edition of this book will spark just as many thoughts and discussions as the first edition!

1. Maintainability

This book is about software architecture. One of the definitions of architecture is the structure of a system or process. In our case, it’s the structure of a software system.

Architecture is designing this structure with a purpose. We’re consciously designing our software system to fulfill certain requirements. There are functional requirements that the software has to fulfill to create value for its users. Without functionality, software is worthless, because it produces no value.

There are also quality requirements (also called non-functional requirements) that the software should fulfill to be considered high quality by its developers and stakeholders. One such quality requirement is maintainability.

What would you say if I told you that maintainability as a quality attribute, in a way, is more important than functionality and that we should design our software for maintainability over everything else? Once we have established maintainability as an important quality, we will use the rest of this book to explore how we can improve the maintainability of our software by applying the concepts of Clean and Hexagonal Architecture.

What does maintainability even mean?

Before you write me off as a lunatic and start looking for options to return this book, let me explain what I mean by maintainability.

Maintainability is only one of the many quality requirements that potentially make up a software architecture. I asked ChatGPT for a list of quality requirements, and this is the result:

- scalability
- flexibility
- maintainability
- security
- reliability
- modularity
- performance
- interoperability
- testability
- cost-effectiveness
- …

The list doesn't end here.[9]

As software architects, we design our software to fulfill the quality requirements that are most important for the software. For a high-throughput trading application, we might focus on scalability and reliability. For an application dealing with personally identifiable information in Germany, we might want to focus on security.

I think it's wrong to lump maintainability in with the rest of the quality requirements because maintainability is special. If software is maintainable, that means it's easy to change.[10] If it's easy to change, it's flexible and probably modular. It's probably cost-effective, too, because easy changes mean cheap changes. If it's maintainable, we can probably evolve it to be scalable, secure, reliable, and performant, should the need arise. We can change the software to be interoperable with other systems because it's easy to change. Last, but not least, maintainability implies testability, because maintainable software is most likely designed from smaller and simpler components that make testing easy.

You can see what I did here. I asked an AI for a list of quality requirements and then tied them all back to maintainability. I could probably tie many more quality requirements back to maintainability with similarly plausible arguments. It's a bit simplistic, of course, but the core of it is true: if software is maintainable, it's easier to evolve in any direction, functionally and non-functionally. And we all know that change is common during the life of a software system.

Maintainability enables functionality

Now back to my claim that maintainability is more important than functionality from the beginning of this chapter.

If you ask a product person what's most important in a software project, they'll tell you that the value the software provides to its users is the most important thing. Software that doesn't provide value to its users means that users don't pay for it. And without paying users, we don't have a working business model, which is the main measure of success in the business world.

So, our software needs to provide value. But it shouldn't provide value at the cost of maintainability. Think about how much more efficient and joyful it is to add functionality to a software system that is easily changeable as compared to a software system where you have to fight your way through one line of code at a time! I'm pretty sure that you've worked on one of those software projects where there's so much cruft and ritual that it takes days or weeks to build a feature that you think should take no more than a couple of hours to complete.

In this way, maintainability is a key supporter of functionality. Bad maintainability means that changes in functionality become more and more expensive over time as shown in Figure 1.1.

[9] For some inspiration about software quality (that has been created by humans, and not a language model) have a look at https://quality.arc42.org/.

[10] In the context of this book, I use the term "maintainability" synonymously with "changeability of a codebase". Also see https://quality.arc42.org/qualities/maintainability for some definitions of maintainability (all of which have to do with changing the software).

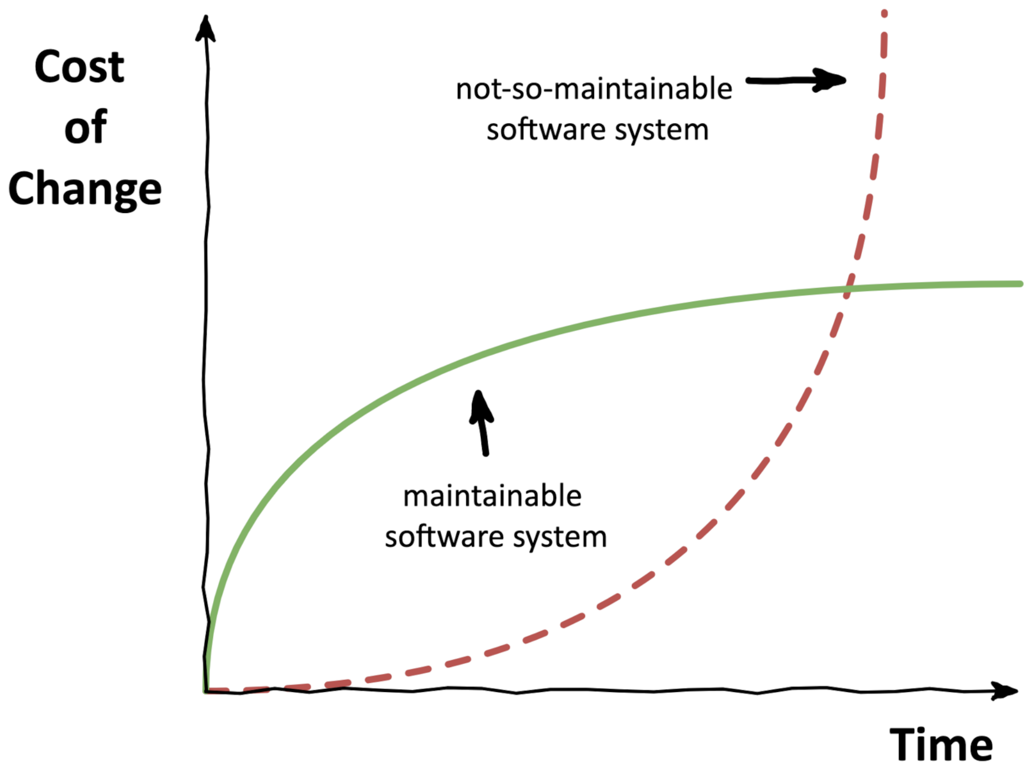

Figure 1.1 - A maintainable software system has a smaller lifetime cost than a not-so-maintainable software system.

In a not-so-maintainable software system, changes in functionality will soon become so expensive that change is a pain. Product people will complain to the engineers about the cost of changes. The engineers will defend themselves by saying that shipping new features has always had a higher priority than increasing maintainability. The probability of conflict increases with the cost of change.

Maintainability is a pacifier. It's inversely proportional to the cost of change and thus to the probability of conflict. Did you ever think about adding maintainability to a software system to avoid conflict? I think that's a good investment in itself.

But what about those big software systems that are successful in spite of bad maintainability? It's true that there are commercially successful software systems out there that are barely maintainable. I've worked on systems where adding a single field to a form is a project that takes weeks of developer time, and the client happily paid a premium for my time.

Those systems usually fall into one (or both) of two categories:

- they are at the end of their life where changes to the system are few and far between, or
- they are backed by a financially well-off company that is willing to throw money at the problem.

Even in the case where a company has a lot of money to spend, the company realizes that they can reduce the maintenance tax by investing in maintainability. So, usually, there are already initiatives underway to make the software more maintainable.

We should always care about the maintainability of the software we're creating, so it doesn't degrade into the dreaded "big ball of mud", but if our software doesn't fall into one of the two categories mentioned previously, we should care even more.

Does this mean that we have to spend a lot of time planning out a maintainable architecture before we even start programming? Do we have to do a big design up front (BDUF), which is often considered synonymous with the waterfall methodology? No, we don't. But we need to do *some* design up-front (SDUF?) to bake a seed of maintainability into the software that can make it easier to evolve the architecture to where it needs to be over time.

Part of that up-front design is choosing an architecture style that defines the guard rails of the software we're building. This book will help you decide whether a Clean - or Ports and Adapters / Hexagonal - architecture is a good fit for your context.

Maintainability generates developer joy

As a developer, would you rather work on software where changes are easy or on software where changes are hard? Don't answer, it's a rhetorical question.

Aside from the direct influence on the cost of change, maintainability has another benefit: it makes developers happy (or, depending on the current project they're working on, it at least makes them less sad).

The term I want to use to describe this happiness is developer joy. It's also known as developer experience or developer enablement. Whatever we call it, it means that we provide the context developers need to do their work well.

Developer joy is directly related to developer productivity. In general, if developers are happy, they do better work. And if they do good work, they are happier. There's a two-way correlation between developer joy and developer productivity:

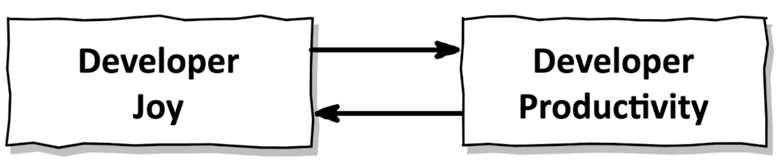

Figure 1.2 - Developer joy influences developer productivity and vice versa.

This correlation has been recognized in the SPACE framework for developer productivity.[11] While SPACE doesn't provide an easy answer on how to measure developer productivity, it provides five categories for such metrics so that we can consciously pick a set of metrics covering all those

[11] *The SPACE of Developer Productivity* by Nicole Forsgren et al., March 6, 2021. "SPACE" stands for satisfaction and well-being, performance, activity, communication and collaboration, and efficiency and flow.

categories to best measure developer productivity in the context of our company and projects. One of these categories (the S in SPACE) is satisfaction and well-being, which I've translated to developer joy for this chapter.

Developer joy not only leads to better productivity, but it naturally also leads to better retention. A developer who enjoys their work will stay with the company. Or rather, a developer who does not enjoy their work is more likely to leave for greener pastures.

So, where does maintainability come into the picture? Well, if our software system is maintainable, we need less time to implement a change, so we are more productive. Also, if our software system is maintainable, we find more joy in making changes, because it's more efficient and we can take more pride in it. Even if our software is not as maintainable as we would like it to be (which is a tautology, to be honest), but we get the opportunity to improve maintainability over time, we are happier and more productive. If we are happy, we're more likely to stay.

Expressed in a diagram, it looks like this:

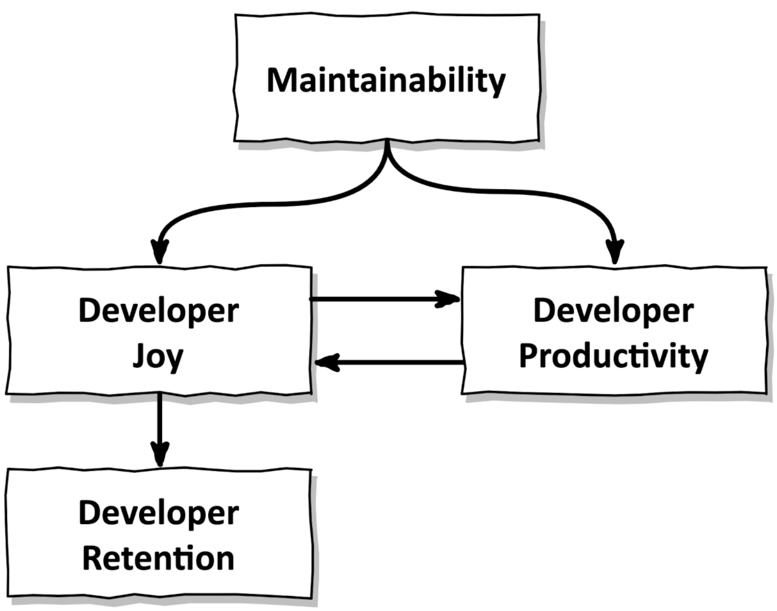

Figure 1.3 - Maintainability directly influences developer joy and productivity. Developer joy influences retention.

Maintainability supports decision-making

When building a software system, we solve problems every day. To most problems we're facing, there is more than one solution. We have to make decisions to choose between those solutions.

Do we copy this bit of code for the new feature we're building? Do we create our objects ourselves or do we use a dependency injection framework? Do we use an overloaded constructor to create this object, or do we create a builder?

Many of those decisions we don't even make consciously. We just apply a pattern or principle we've used before that our intuition says will work in the current situation, as follows:

- we apply DRY (don't repeat yourself) when we find code duplication,
- we use dependency injection to make the code more testable,
- we introduce a builder to make it simpler to create an object,
- … and so on.

If we take a look at these and many other well-known patterns, then what is their effect? In many cases, the main effect is that they make the code easier to change in the future (i.e., they make it more maintainable). Maintainability is built into many of the decisions we're making automatically every day!

We can take advantage of that even when facing tougher decisions that require more than just applying a pre-canned pattern. Whenever we have to decide between multiple options, we can choose the one that makes the code easier to change in the future.[12] No more agonizing between different options. We just take the one that increases maintainability the most.

Expressed as a diagram, it's pretty simple:

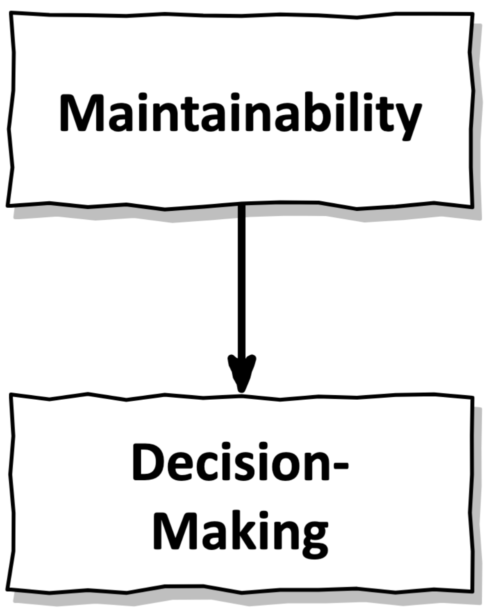

Figure 1.4 - Maintainability influences decision-making.

Like most principles, this is a generalization, of course. In a given context, the right decision might be to take the option that does not improve maintainability or even reduces maintainability. But as a default rule to fall back on, choosing maintainability is a guide that simplifies daily decision-making.

[12]In a talk from 2022 with the same name, (Pragmatic) Dave Thomas called the principle of making decisions based on changeability "One Rule to Rule Them All". I didn't find the talk online, but I hope he will add it to his website at some point.

Maintaining maintainability

Alright, I assume that you believe me that maintainability positively influences developer joy, productivity, and decision-making. How do we know that the changes we make to our codebase increase (or at least don't decrease) maintainability? How do we manage maintainability over time?

The answer to that question is to create and maintain an architecture that makes it easy to create maintainable code. A good architecture makes it easy to navigate the codebase. In an easily navigable codebase, it's a breeze to modify existing features or add new features. The dependencies between the components of our application are clear and not tangled. In summary, good architecture increases maintainability:

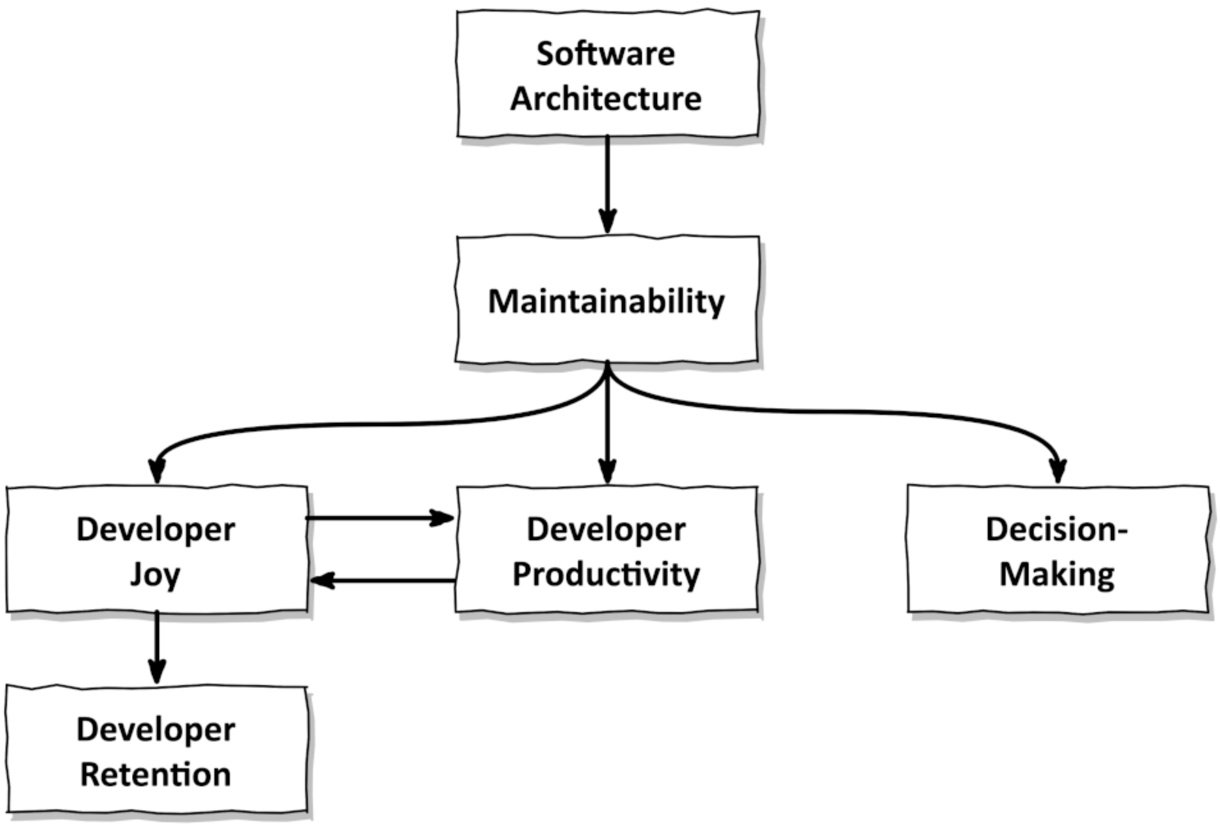

Figure 1.5 - Software architecture influences maintainability.

By extension, a good architecture increases developer joy, developer productivity, developer retention, and decision-making. We could go on and find even more things influenced directly or indirectly by software architecture.

This correlation means that we should invest a bit of thought into how we structure our code. How do we group our code files into components? How do we manage the dependencies between those components? Which dependencies are necessary, and which should be discouraged to keep the codebase supple to change?

Which brings us to the purpose of this book. This book shows one way of structuring a codebase to make it maintainable. The architecture style described in this book is one way of implementing a Clean/Hexagonal Architecture. This architecture style is not a silver bullet to solve all problems with building software, however. As we will learn in Chapter 15, Deciding on an Architecture Style, it's not suitable for all kinds of software applications.

I encourage you to take what you learn in this book, play around with the ideas, modify them to make them yours, and then add them to your toolbox to apply when they feel right in a given context. Each of the following chapters ends with a section titled "How does this help me build maintainable software?". This section will summarize the main ideas of each chapter and hopefully help you to make decisions regarding the architecture of your current or future software projects.

2. What's Wrong with Layers?

Chances are that you have developed a layered (web) application in the past. You might even be doing it in your current project right now.

Thinking in layers has been drilled into us in computer science classes, tutorials, and best practices. It has even been taught in books.[13]

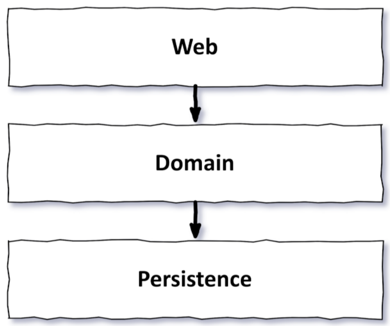

Figure 2.1 - A conventional web application architecture consists of a web layer, a domain layer, and a persistence layer.

Figure 2.1 shows a high-level view of the very common three-layer architecture. We have a web layer that receives requests and routes them to a service in the domain layer.[14] The service does some business logic and calls components from the persistence layer to query for or modify the current state of our domain entities in the database.

You know what? Layers are a solid architecture pattern! If we get them right, we're able to build domain logic that is independent of the web and persistence layers. We can switch out the web or persistence technologies without affecting our domain logic, if the need arises. We can also add new features without affecting existing features.

With a good layered architecture, we're keeping our options open and are able to quickly adapt to changing requirements and external factors (such as our database vendor doubling their prices overnight). A good layered architecture is *maintainable*.

So, what's wrong with layers?

In my experience a layered architecture is very vulnerable to changes that make it hard to maintain. It allows bad dependencies to creep in and make the software increasingly harder to change over time. Layers don't provide enough guard rails to keep the architecture on track. We need to rely too much on human discipline and diligence to keep it maintainable. In the following sections, I'll tell you why.

[13] *Software Architecture Patterns* by Mark Richards, O'Reilly, 2015

[14] In this book, I use the terms "domain" and "business" synonymously. The domain layer or business layer is the place in the code that solves the business problems, as opposed to code that solves technical problems, like persisting things in a database or processing web requests.

They promote database-driven design

By its very definition, the foundation of a conventional layered architecture is the database. The web layer depends on the domain layer which in turn depends on the persistence layer and thus the database. Everything builds on top of the persistence layer. This is problematic for several reasons.

Let's take a step back and think about what we're trying to achieve with almost any application we're building. We're typically trying to create a model of the rules or "policies" that govern the business in order to make it easier for the users to interact with them.

We're primarily trying to model behavior, not state. Yes, state is an important part of any application, but the behavior is what changes the state and thus drives the business!

So, why are we making the database the foundation of our architecture and not the domain logic?

Think back to the last use cases you implemented in any application. Did you start by implementing the domain logic or the persistence layer? Most likely, you thought about what the database structure would look like and only then moved on to implementing the domain logic on top of it.

This makes sense in a conventional layered architecture, since we're going with the natural flow of dependencies. But it makes absolutely no sense from a business point of view! We should build the domain logic before building anything else! We want to find out whether we have understood the business rules correctly. And only once we know we're building the right domain logic should we move on to build a persistence and web layer around it.

A driving force in such a database-centric architecture is the use of object-relational mapping (ORM) frameworks. Don't get me wrong, I love those frameworks and work with them regularly. But if we combine an ORM framework with a layered architecture, we're easily tempted to mix business rules with persistence aspects.

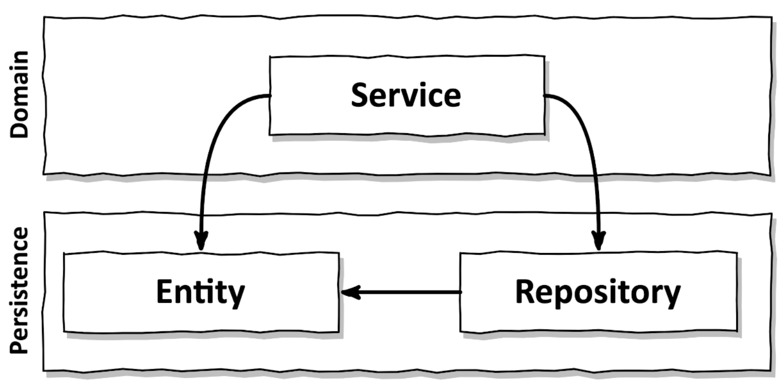

Figure 2.2 - Using the database entities in the domain layer leads to strong coupling with the persistence layer.

Usually, we have ORM-managed entities as part of the persistence layer, as shown in Figure 2.2. Since a layer may access the layers below them, the domain layer is allowed to access those entities. And if it's allowed to use them, it will use them at some point.

This creates a strong coupling between the domain layer and the persistence layer. Our business services use the persistence model as their business model and have to deal not only with the domain logic, but also with eager versus lazy loading, database transactions, flushing caches and similar housekeeping tasks.[15]

The persistence code is virtually fused into the domain code and thus it's hard to change one without the other. That's the opposite of being flexible and keeping options open, which should be the goal of our architecture.

They're prone to shortcuts

In a conventional layered architecture, the only global rule is that from a certain layer, we can only access components in the same layer or a layer below. There may be other rules that a development team has agreed upon and some of them might even be enforced by tooling, but the layered architecture style itself does not impose those rules on us.

So, if we need access to a certain component in a layer above ours, we can just push the component down a layer and we're allowed to access it. Problem solved. Doing this once may be OK. But doing it once opens the door for doing it a second time. And if someone else was allowed to do it, so am I, right?

I'm not saying that as developers, we take such shortcuts lightly. But if there is an option to do something, someone will do it, especially in combination with a looming deadline. And if something has been done before, the likelihood of someone doing it again will increase drastically. This is a psychological effect called the "Broken Windows Theory" - more on this in Chapter 11, Taking Shortcuts Consciously.

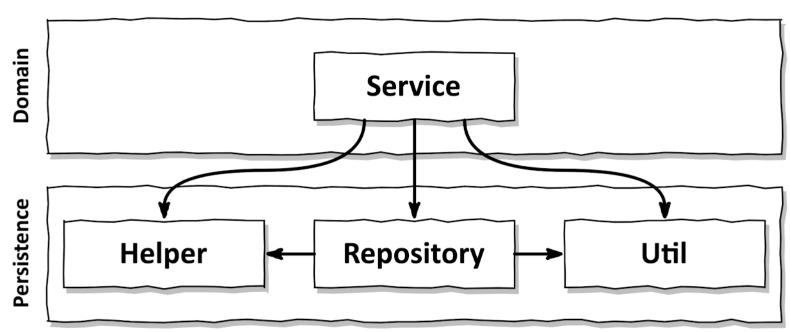

Figure 2.3 - Since any layer may access everything in the persistence layer, it tends to grow fat over time.

Over years of development and maintenance of a software project, the persistence layer may very well end up like in Figure 2.3.

[15] In his seminal book *Refactoring* (Pearson, 2018), Martin Fowler calls this symptom "divergent change": having to change seemingly unrelated parts of the code to implement a single feature. This is a code smell that should trigger a refactoring.

The persistence layer (or, in more generic terms, the bottom-most layer) will grow fat as we push components down through the layers. Perfect candidates for this are helper or utility components since they don't seem to belong to any specific layer.

So, if we want to disable the shortcut mode for our architecture, layers are not the best option, at least not without enforcing some kind of additional architecture rules. And by enforcing, I don't mean a senior developer doing code reviews, but automatically enforced rules that make the build fail when they're broken.

They grow hard to test

A common evolution within a layered architecture is that layers are skipped. We access the persistence layer directly from the web layer, since we're only manipulating a single field of an entity, and for that we need not bother the domain layer, right?

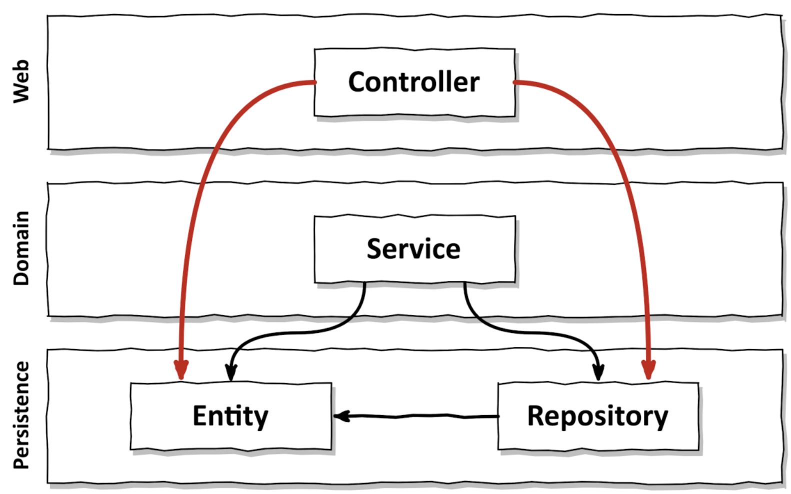

Figure 2.4 - Skipping the domain layer tends to scatter domain logic across the codebase.

Figure 2.4 shows how we're skipping the domain layer and accessing the persistence layer right from the web layer.

Again, this feels OK the first couple of times, but it has two drawbacks if it happens often (and it will, once someone has done the first step).

First, we're implementing domain logic in the web layer, even if it's only manipulating a single field. What if the use case expands in the future? We're most likely going to add more domain logic to the web layer, mixing responsibilities and spreading essential domain logic across all layers.

Second, in the unit tests of our web layer, we not only have to manage the dependencies on the domain layer, but also the dependencies on the persistence layer. If we're using mocks in our tests, that means we have to create mocks for both layers. This adds complexity to the tests. And a complex test setup is the first step towards no tests at all because we don't have time for them. As the web component grows over time, it may accumulate a lot of dependencies on different persistence components, adding to the test's complexity. At some point, it takes more time for us to understand the dependencies and create mocks for them than to actually write test code.

They hide the use cases

As developers, we like to create new code that implements shiny new use cases. But we usually spend much more time changing existing code than we do creating new code. This is not only true for those dreaded legacy projects in which we're working on a decades-old codebase but also for a hot new greenfield project after the initial use cases have been implemented.

Since we're so often searching for the right place to add or change functionality, our architecture should help us to quickly navigate the codebase. How does a layered architecture hold up in this regard?

As already discussed previously, in a layered architecture, it easily happens that domain logic is scattered throughout the layers. It may exist in the web layer if we're skipping the domain logic for an "easy" use case. And it may exist in the persistence layer if we have pushed a certain component down so it can be accessed from both the domain and persistence layers. This already makes finding the right spot to add new functionality hard.

But there's more. A layered architecture does not impose rules on the "width" of domain services. Over time, this often leads to very broad services that serve multiple use cases (see Figure 2.5).

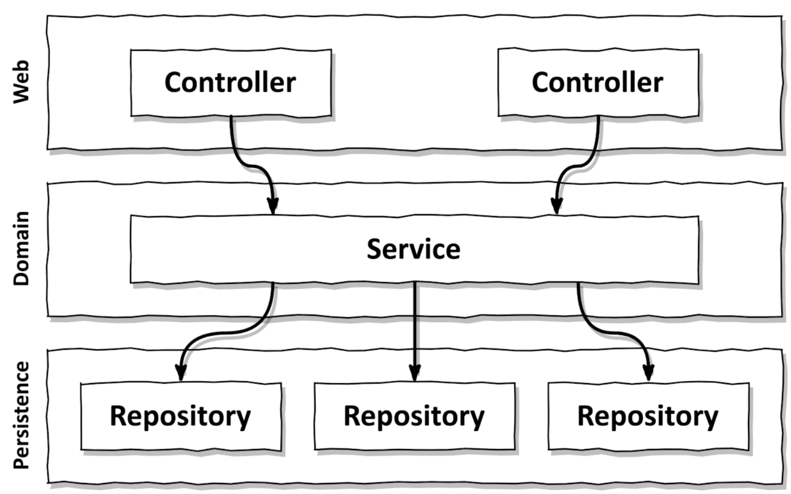

Figure 2.5 - "Broad" services make it hard to find a certain use case within the codebase.

A broad service has many dependencies on the persistence layer and many components in the web layer depend on it. This not only makes the service hard to test but also makes it hard for us to find the code responsible for the use case we want to work on.

How much easier would it be if we had highly specialized, narrow domain services that each serve a single use case? Instead of searching for the user registration use case in `UserService`, we would just open up `RegisterUserService` and start hacking away.

They make parallel work difficult

Management usually expects us to be done with building the software they sponsor on a certain date. Actually, they even expect us to be done within a certain budget as well, but let's not complicate things here.

Aside from the fact that I have never seen "done" software in my career as a software engineer, to be "done" by a certain date usually implies that multiple people have to work in parallel.

You probably know this famous conclusion from "The Mythical Man-Month", even if you haven't read the book:

> *Adding manpower to a late software project makes it later.*[16]

This also holds true, to a degree, in software projects that are not (yet) late. You cannot expect a large group of 50 developers to be 5 times faster than a smaller team of 10 developers. If they're working on a very large application where they can split up into sub-teams and work on separate parts of the software, it may work, but in most contexts, they would step on each other's feet.

But on a healthy scale, we can certainly expect to be faster with more people on the project. And management is right to expect that of us.

To meet this expectation, our architecture must support parallel work. This is not easy. And a layered architecture doesn't really help us here.

Imagine we're adding a new use case to our application. We have three developers available. One can add the needed features to the web layer, one to the domain layer, and the third to the persistence layer, right?

Well, it usually doesn't work that way in a layered architecture. Since everything builds on top of the persistence layer, the persistence layer must be developed first. Then comes the domain layer and finally the web layer. So only one developer can work on the feature at a time!

"Ah, but the developers can define interfaces first", you say, "and then each developer can work against these interfaces without having to wait for the actual implementation." Sure, this is possible, but only if we haven't mixed our domain and persistence logic as discussed previously, blocking us from working on each aspect separately.

If we have broad services in our codebase, it may even be hard to work on different features in parallel. Working on different use cases will cause the same service to be edited in parallel, which leads to merge conflicts and potentially regressions.

[16] *The Mythical Man-Month: Essays on Software Engineering* by Frederick P. Brooks, Jr., Addison-Wesley, 1995

How does this help me build maintainable software?

If you have built layered architectures in the past, you can probably relate to some of the issues discussed in this chapter, and you could maybe even add some more.

If done correctly, and if some additional rules are imposed on it, a layered architecture can be very maintainable and can make changing or adding to the codebase a breeze.

However, the discussion shows that a layered architecture allows many things to go wrong. Without good self-discipline it's prone to degradeing and becoming less maintainable over time. And our self-discipline usually takes a hit each time a team member rotates into or out of the team, or a manager draws a new deadline around the development team.

Keeping the traps of a layered architecture in mind will help us the next time we argue against taking a shortcut and for building a more maintainable solution instead - be it in a layered architecture or a different architecture style.

3. Inverting Dependencies

After the talk about layered architecture in the previous chapter, you're right to expect this chapter to discuss an alternative approach. We'll start by discussing two of the SOLID[17] principles and then apply them to create a Clean or Hexagonal Architecture that addresses the problems of a layered architecture.

The Single Responsibility Principle

Everyone in software development probably knows the Single Responsibility Principle (SRP) or at least assumes to know it. A common interpretation of this principle is this:

> *A component should do only one thing and do it right.*

That's good advice, but not the actual intent of the SRP.

"Doing only one thing" is actually the most obvious interpretation of "single responsibility", so it's no wonder that the SRP is frequently interpreted like this. Let's just observe that the name of the SRP is misleading.

Here's the actual definition of the SRP:

> *A component should have only one reason to change.*

As we see, "responsibility" should actually be translated to "reason to change" instead of "do only one thing". Perhaps we should rename the SRP to "Single Reason to Change Principle".

If a component has only one reason to change, it might end up doing only one thing, but the more important part is that it has only this one reason to change.

What does that mean for our architecture?

If a component has only one reason to change, we don't have to worry about this component at all if we change the software for any other reason, because we know that it will still work as expected.

Sadly, it's very easy for a reason to change to propagate through code via the dependencies of a component to other components (see Figure 3.1).

[17]**S**ingle Responsibility Principle, **O**pen-Closed Principle, **L**iskov Substitution Principle, **I**nterface Segregation Principle, **D**ependency Inversion Principle. You can read more about these Principles in *Clean Architecture* by Robert C. Martin or on Wikipedia.

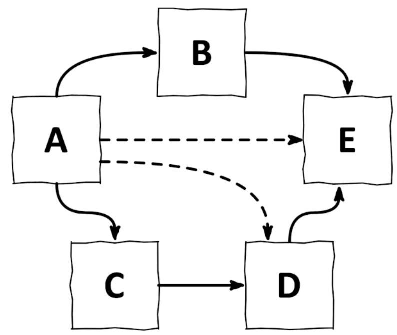

Figure 3.1 - Each dependency of a component is a possible reason to change this component, even if it is only a transitive dependency (dashed arrows).

In the figure above, component A depends on many other components (either directly or transitively) while component E has no dependencies at all.

The only reason to change component E is when the functionality of E must change due to some new requirement. Component A, however, might have to change when any of the other components change, because it depends on them.

Many codebases grow harder - and thus more expensive - to change over time because the SRP is violated. Over time, components collect more and more reasons to change. After having collected many reasons to change, changing one component might cause another component to fail.

A tale about side effects

I once was part of a project where my team inherited a ten-year-old codebase built by another software shop. The client had decided to replace the development team to reduce the ongoing maintenance costs and improve development speed for new features. So, we got the contract.

As was to be expected, it was not easy to gain an understanding of what the code actually did, and the changes we did in one area of the codebase often had side effects in other areas. But we managed by testing exhaustively, adding automated tests, and refactoring a lot.

After some time of successfully maintaining and extending the codebase, the client requested a new feature. And they wanted us to build it in a way that was very awkward for the users of the software. So I proposed to do it in a more user-friendly way that was even less expensive to implement since it needed fewer overall changes. It needed a small change in a certain very central component, however.

The client declined and ordered the more awkward and expensive solution. When I asked for the reason, they said that they were afraid of the side effects because changes in that one component by the previous development team have always broken something else in the past.

Sadly, this is an example of how you can indoctrinate your client to pay extra for modifying badly architected software. Luckily, most clients will not play along with this game, so let’s try to build well-architected software instead.

The Dependency Inversion Principle

In our layered architecture, the cross-layer dependencies always point downward to the next layer. When we apply the Single Responsibility Principle on a high level, we notice that the upper layers have more reasons to change than the lower layers.

Thus, due to the domain layer's dependency on the persistence layer, each change in the persistence layer potentially requires a change in the domain layer. But the domain code is the most important code in our application! We don't want to have to change it when something changes in the persistence code!

So, how can we get rid of this dependency?

The Dependency Inversion Principle provides the answer.

In contrast to the SRP, the Dependency Inversion Principle (DIP) means what the name suggests:

> *We can turn around (invert) the direction of any dependency within our codebase*[18]

How does that work? Let's try to invert the dependency between our domain and persistence code so that the persistence code depends on the domain code, reducing the number of reasons to change for the domain code.

We start with a structure like in Figure 2.2 from Chapter 2, What's Wrong with Layers? We have a service in the domain layer that works with entities and repositories from the persistence layer.

First of all, we want to pull up the entities into the domain layer because they represent our domain objects and our domain code pretty much revolves around changing state in those entities.

But now, we'd have a circular dependency between both layers since the repository from the persistence layer depends on the entity, which is now in the domain layer. This is where we apply the DIP. We create an interface for the repository in the domain layer and let the actual repository in the persistence layer implement it. The result is something like in Figure 3.2.

[18] Actually, we can only invert dependencies when we have control over the code on both ends of the dependency. If we have a dependency on a third-party library, we cannot invert it, since we don't control the code of that library.

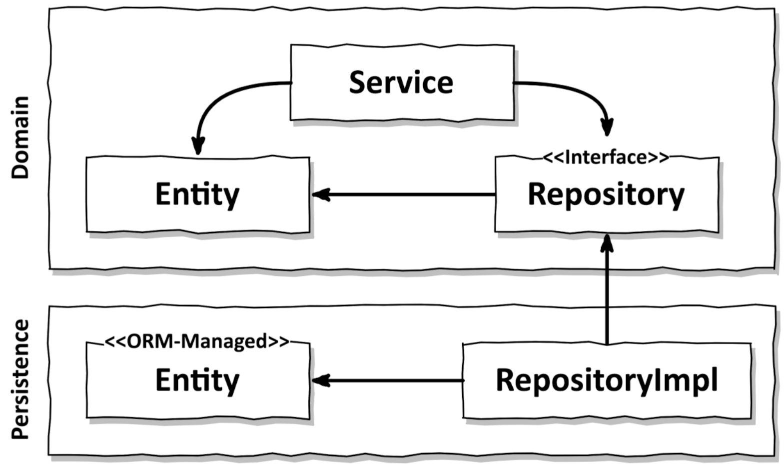

Figure 3.2 - By introducing an interface in the domain layer, we can invert the dependency so that the persistence layer depends on the domain layer.

With this trick, we have liberated our domain logic from the oppressive dependency on the persistence code. This is a core feature of the two architecture styles we're going to discuss in the upcoming sections.

Clean Architecture

Robert C. Martin coined the term "Clean Architecture" in his book with the same name.[19] In a Clean Architecture, in his opinion, the business rules are testable by design and independent of frameworks, databases, UI technologies, and other external applications or interfaces.

This means that the domain code must not have any outward facing dependencies. Instead, with the help of the Dependency Inversion Principle, all dependencies point toward the domain code.

Figure 3.3 shows what such an architecture might look like on an abstract level.

[19] *Clean Architecture* by Robert C. Martin, Prentice Hall, 2017, Chapter 22

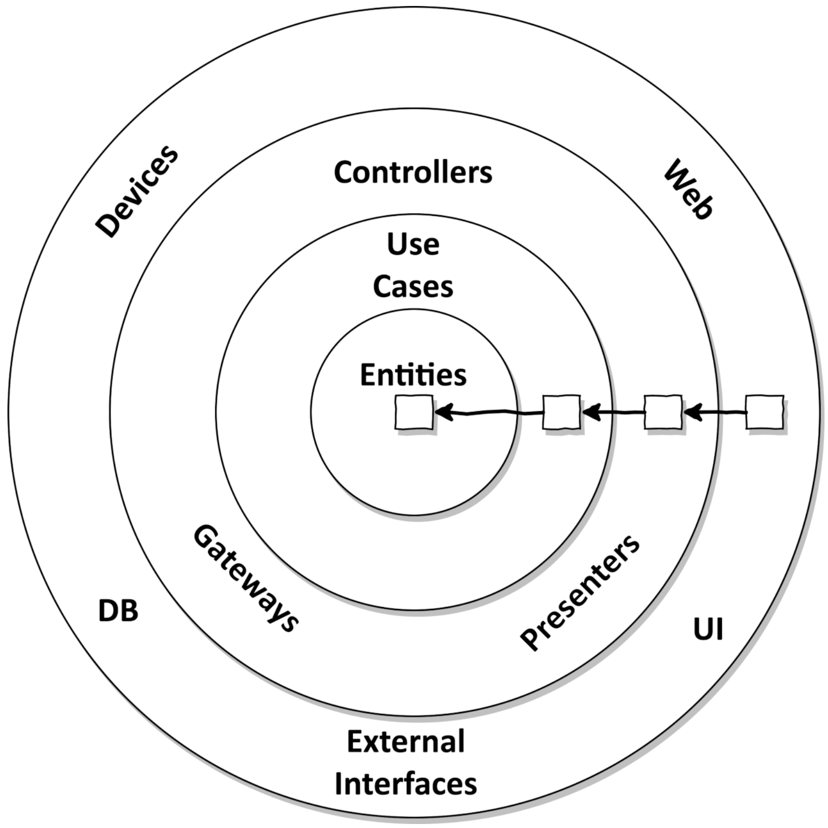

Figure 3.3 - In a Clean Architecture, all dependencies point inward toward the domain logic. Source: *Clean Architecture* by Robert C. Martin.

The layers in this architecture are wrapped around each other in concentric circles. The main rule in such an architecture is the "Dependency Rule", which states that all dependencies between those layers must point inward.

The core of the architecture contains the domain entities which are accessed by the surrounding use cases. The use cases are what we have called services earlier, but are more fine-grained to have a single responsibility (i.e. a single reason to change), thus avoiding the problem of broad services we have discussed earlier.

Around this core, we can find all the other components of our application that support the business rules. This support can mean providing persistence or providing a user interface, for example. Also, the outer layers may provide adapters to any other third-party component.

Since the domain code knows nothing about which persistence or UI framework is used, it cannot contain any code specific to those frameworks and will concentrate on the business rules. We have all the freedom we can wish for to model the domain code. We could for example apply Domain-Driven Design (DDD) in its purest form. Not having to think about persistence or UI-specific problems makes that so much easier.

As we might expect, a Clean Architecture comes at a cost. Since the domain layer is completely decoupled from the outer layers like persistence and UI, we have to maintain a model of our application's entities in each of the layers.

Let's assume, for instance, that we're using an object-relational mapping (ORM) framework in our persistence layer. An ORM framework usually expects specific entity classes that contain metadata describing the database structure and the mapping of object fields to database columns. Since the domain layer doesn't know the persistence layer, we cannot use the same entity classes in the domain layer and have to create them in both layers. This means that the persistence layer needs to map the domain entities into its own representation. A similar mapping applies between the domain layer and other outer layers.

But that's a good thing! This decoupling is exactly what we wanted to achieve to free the domain code from framework-specific problems. The Java Persistence API (the standard object-relational API in the Java world), for instance, requires the ORM-managed entities to have a default constructor without arguments that we might want to avoid in our domain model. In Chapter 9, Mapping between Boundaries, we'll talk about different mapping strategies, including a no-mapping strategy that just accepts the coupling between the domain and persistence layers.

Since the Clean Architecture by Robert C. Martin is somewhat abstract, let's go a level of detail deeper and look at "Hexagonal Architecture", which gives the Clean Architecture principles a more concrete shape.

Hexagonal Architecture

The term "Hexagonal Architecture" stems from Alistair Cockburn and has been around for quite some time.[20] It applies the same principles that Robert C. Martin later described in more general terms in his Clean Architecture.

[20]The primary source for the term "Hexagonal Architecture" seems to be an article on Alistair Cockburn's website at https://alistair.cockburn.us/hexagonal-architecture/.

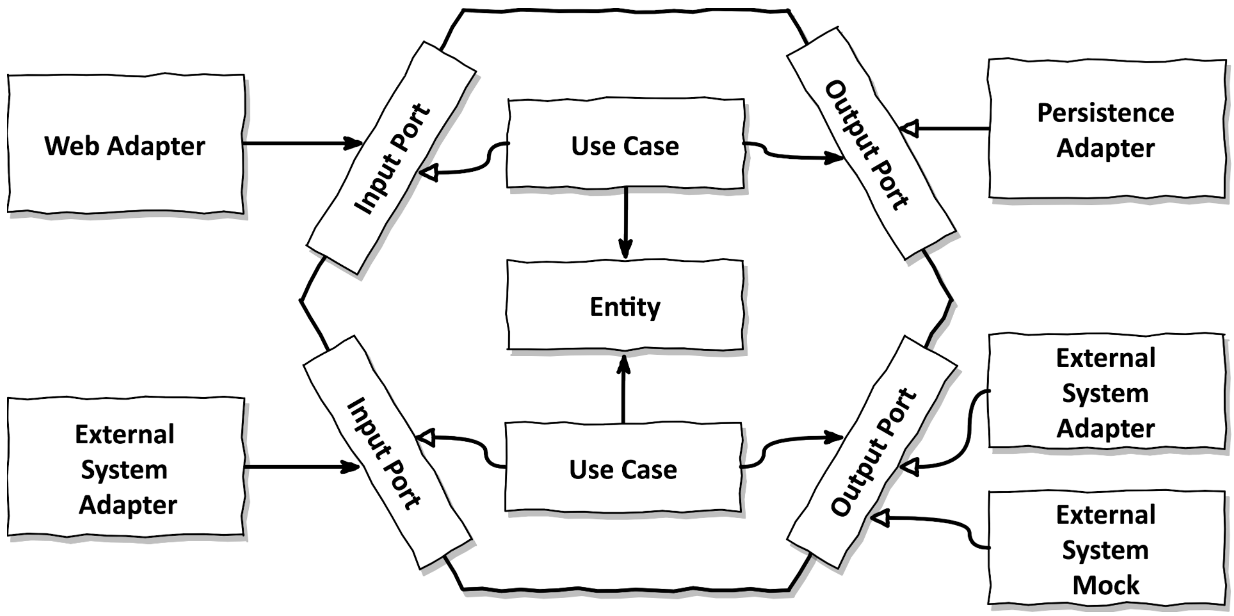

Figure 3.4 - A Hexagonal Architecture is also called a "Ports and Adapters" architecture, since the application core provides specific ports for each adapter to interact with.

Figure 3.4 shows what a Hexagonal Architecture might look like. The application core is represented as a hexagon, giving this architecture style its name. The hexagon shape has no meaning, however, so we might just as well draw an octagon and call it "Octagonal Architecture". According to legend, the hexagon was simply used instead of the common rectangle to show that an application can have more than four sides connecting it to other systems or adapters.

Within the hexagon, we find our domain entities and the use cases that work with those entities. Note that the hexagon has no outgoing dependencies, so that the Dependency Rule from Martin's Clean Architecture holds true. Instead, all dependencies point towards the center.

Outside the hexagon, we find various adapters that interact with the application. There might be a web adapter that interacts with a web browser, some adapters interacting with external systems and an adapter that interacts with a database to implement persistence.

The adapters on the left side are adapters that drive our application (because they call our application core) while the adapters on the right side are driven by our application (because they are called by our application core).

To allow communication between the application core and the adapters, the application core provides specific ports. For driving adapters, such a port might be an interface that is implemented by one of the use case classes in the core and called by the adapter. For a driven adapter, it might be an interface that is implemented by the adapter and called by the core. We might even have multiple adapters implementing the same port: one for communicating with a real external system, and one for communicating with a mock to be used in testing, for example.

To clearly call out a central attribute of Hexagonal Architecture: the application core (the hexagon) defines and owns the interface to the outside (the ports). The adapters then work with this interface. This is the Dependency Inversion Principle applied on the architecture level.

Due to its central concepts this architecture style is also known as a "Ports and Adapters" architecture.

Just like Clean Architecture, we can organize this Hexagonal Architecture into layers. The outermost layer consists of the adapters that translate between the application and other systems.

Next, we can combine the ports and use case implementations to form the application layer, because they define the interface of our application. The final layer contains the domain entities implementing the business rules.

The business logic is implemented in the use case classes and entities. The use case classes are narrow domain services, implementing just a single use case. We can choose to combine multiple use cases to a broader domain service, of course, but ideally we do this only when the use cases are often used together, to increase maintainability.

Potentially, we will want to introduce the concept of application services, too. An application service is a service that coordinates calls to use cases (domain services) as shown in Figure 3.5.

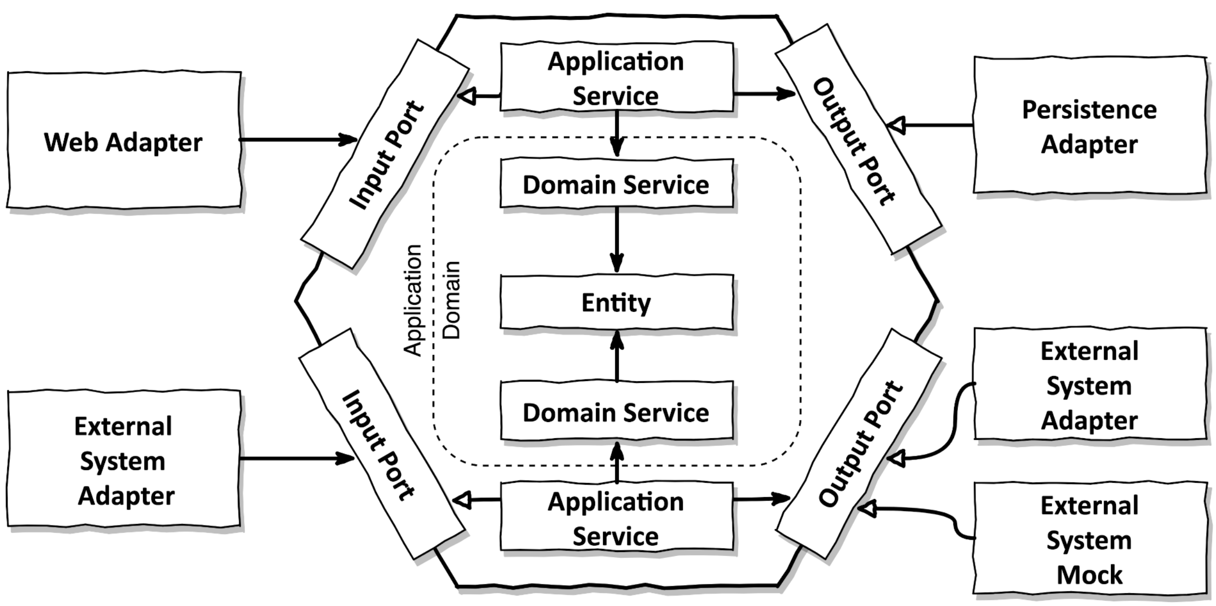

Figure 3.5 - A Hexagonal Architecture using the Domain-Driven Design concepts of application and domain services.

Here, the application services translate between the input and output ports and the domain services, shielding the domain services from the outside world, and potentially coordinating between the domain services. The "Domain Service" boxes are synonymous to the "Use Case" boxes from above, we're just now using terminology borrowed from Domain-Driven Design.

As this discussion implies, we're free to design our application code as we see fit inside the hexagon. We can go simple or sophisticated, matching the complexity and size of our application. We will learn more about managing code within our hexagon in Chapter 13, Managing Multiple Bounded Contexts.

In the next chapter, we'll discuss a way to organize such an architecture in code.

How does this help me build maintainable software?

Call it "Clean Architecture", "Hexagonal Architecture", or "Ports and Adapters Architecture" - by inverting our dependencies so that the domain code has no dependencies on the outside, we can decouple our domain logic from all those persistence and UI-specific problems and reduce the number of reasons to change throughout the codebase. And fewer reasons to change lead to better maintainability.

The domain code is free to be modelled as it best fits the business problems, while the persistence and UI code are free to be modelled as it best fits the persistence and UI problems.

In the rest of this book, we'll apply the Hexagonal Architecture style to a web application. We'll start by creating the package structure of our application and discussing the role of dependency injection.

4. Organizing Code

Wouldn't it be nice to recognize the architecture just by looking at the code?

In this chapter, we'll examine different ways of organizing code and introduce an expressive package structure that directly reflects a Hexagonal Architecture.

In greenfield software projects, the first thing we try to get right is the package structure. We set up a nice-looking structure that we intend to use for the rest of the project. Then, during the project, things become hectic, and we realize that in many places the package structure is just a nice-looking facade for an unstructured mess of code. Classes in one package import classes from other packages that should not be imported.

We'll discuss different options for structuring the code of the BuckPal example application that was introduced in the preface. More specifically, we'll look at the "Send Money" use case that allows a user to transfer money from their account to another.

Organizing by layer

The first approach to organizing our code is by layer. We might organize the code like this:

```
buckpal
├── domain
|    ├── Account
|    ├── Activity
|    ├── AccountRepository
|    └── AccountService
├── persistence
|    └── AccountRepositoryImpl
└── web
     └── AccountController
```

For each of our layers `web`, `domain` and `persistence` we have a dedicated package. As discussed in Chapter 2, What's Wrong with Layers?, simple layers may not be the best structure for our code for several reasons, so we have already applied the Dependency Inversion Principle here, only allowing dependencies toward the domain code in the `domain` package. We did this by introducing the `AccountRepository` interface in the `domain` package and implementing it in the `persistence` package.

We can find at least three reasons why this package structure is suboptimal, however.

- First, we have no package boundary between functional slices or features of our application. If we add a feature for managing users, we'll add a `UserController` to the `web` package, a `UserService`, `UserRepository`, and `User` to the `domain` package and a `UserRepositoryImpl` to the `persistence` package. Without further structure, this might quickly become a mess of classes leading to unwanted side effects between supposedly unrelated features of the application.
- Second, we can't see which use cases our application provides. Can you tell what use cases the `AccountService` or `AccountController` classes implement? If we're looking for a certain feature, we have to guess which service implements it and then search for the responsible method within that service.
- Finally, we can't see our target architecture within the package structure. We can guess that we have followed the Hexagonal Architecture style and then browse the classes in the `web` and `persistence` packages to find the web and persistence adapters. But we can't see at a glance which functionality is called by the web adapter and which functionality the persistence adapter provides to the domain layer. The incoming and outgoing ports are hidden in the code.

Let's try to address some issues of the "organize by layer" approach.

Organizing by feature

The next approach is to organize our code by feature:

```
buckpal
└── account
    ├── Account
    ├── SendMoneyController
    ├── AccountRepository
    ├── AccountRepositoryImpl
    └── SendMoneyService
```

In essence, we have put all the code related to accounts into the high-level package `account`. We have also removed the layer packages.

Each new group of features will get a new high-level package next to `account` and we can enforce package boundaries between the features by using package-private visibility for the classes that should not be accessed from the outside.

The package boundaries, combined with package-private visibility, enable us to avoid unwanted dependencies between features.

We have also renamed `AccountService` to `SendMoneyService` to narrow its responsibility (we actually could have done that in the package-by-layer approach, too). We can now see that the code implements the "Send Money" use case just by looking at the class name. Making the application's functionality visible in the code is what Robert Martin calls a “Screaming Architecture”, because it screams its intention at us.[21]

However, the package-by-feature approach makes our architecture *even less evident* in the code than the package-by-layer approach. We have no package names to identify our adapters, and we still don’t see the incoming and outgoing ports. What’s more, even though we have inverted the dependencies between the domain code and persistence code so that `SendMoneyService` only knows about the `AccountRepository` interface and not its implementation, we cannot use package-private visibility to protect the domain code from accidental dependencies on the persistence code.

So, how can we make our target architecture visible at a glance? It would be nice if we could point a finger at a box in an architecture diagram like Figure 3.4 and instantly know which part of the code is responsible for that box.

Let’s take one more step to create a package structure that is expressive enough to support this.

An architecturally expressive package structure

In a Hexagonal Architecture, we have entities, use cases, input and output ports, and input and output (or “driving” and “driven”) adapters as our main architectural elements. Let’s fit them into a package structure that expresses this architecture:

```
buckpal
├── adapter
|   ├── in
|   |   └── web
|   |       └── SendMoneyController
|   └── out
|       └── persistence
|           ├── AccountPersistenceAdapter
|           └── SpringDataAccountRepository
├── application
|   ├── domain
|   |   ├── model
|   |   |   └── Account
|   |   └── service
|   |       └── SendMoneyService
|   └── port
|       ├── in
|       |   └── SendMoneyUseCase
|       └── out
|           └── UpdateAccountStatePort
└── common
```

[21] *Clean Architecture* by Robert C. Martin, Prentice Hall, 2017, Chapter 21

We can map each element of the architecture directly to one of the packages. At the highest level, we have the `adapter` and `application` packages.

The `adapter` package contains the incoming adapters that call the application's incoming ports and the outgoing adapters that provide implementations for the application's outgoing ports. In our case, we're building a simple web application with the `web` and `persistence` adapters, each having its own sub-package.

Moving the adapters' code to their own packages has the benefit that we can very easily replace one adapter with another implementation, should the need arise. Imagine we have started implementing a persistence adapter against a simple key-value database, because we *thought* we knew the required access patterns, but those patterns have changed, and we would be better off with an SQL database now. We simply implement all relevant outgoing ports in a new adapter package and then remove the old package.

The `application` package contains the "hexagon", as in, our application code. This code consists of our domain model, which lives in the `domain` package and the port interfaces, which live in the `port` package.

Why are the ports inside the `application` package and not next to it? The ports are our way to apply the Dependency Inversion Principle. The application defines these ports to communicate with the outside world. Putting the `port` package inside the `application` package expresses that the application owns the ports.

The `domain` package contains our domain entities and domain services that implement the input ports and coordinate between the domain entities.

Finally, there is a `common` package, which contains some code that is shared across the rest of the codebase.

Phew, that's a lot of technical-sounding packages. Isn't that confusing?

Imagine we have a high-level view of our Hexagonal Architecture hanging on the office wall and we're talking to a colleague about modifying a client to a third-party API we're consuming. While discussing this, we can point at the corresponding outgoing adapter on the poster to better understand each other. Then, when we're finished talking, we sit down in front of our IDE and can start working on the client right away because the code of the API client we have talked about can be found in the *adapter/out/<name-of-adapter>* package.

Rather helpful instead of confusing, don't you think?

This package structure is a powerful element in the fight against the so-called "architecture/code gap" or "model/code gap".[22] These terms describe the fact that in most software development projects the architecture is only an abstract concept that cannot be directly mapped to the code. With time, if the package structure (among other things) does not reflect the architecture, the code will usually deviate more and more from the target architecture.

[22] *Just Enough Architecture* by George Fairbanks, Marshall & Brainerd, 2010, page 167

Also, this expressive package structure promotes active thinking about the architecture. We have to actively decide which package our code to put into.

But don't so many packages mean that everything has to be public in order to allow access across packages?

For the adapter packages, at least, this is not true. All the classes they contain may be package private since they are not called by the outside world except over port interfaces, which live within the `application` package. So, there are no accidental dependencies from the application layer to the adapter classes.

Within the `application` package, however, some classes indeed have to be public. The ports must be public because they must be accessible to the adapters by design. The domain model must be public to be accessible to the services and, potentially, to the adapters. The services don't need to be public because they can be hidden behind the incoming port interfaces.

So, yes, a fine-grained package structure such as this requires us to make some classes public that might be package-private in a coarser-grained package structure. We'll look at ways to catch unwanted access to those public classes in Chapter 12, Enforcing Architecture Boundaries.

You might notice that this package structure contains only one domain, namely the domain handling account transactions. Many applications will contain code from more than one domain, however.

As we will learn in Chapter 13, Managing Multiple Bounded Contexts, Hexagonal Architecture doesn't really tell us how to manage multiple domains. We can, of course, put the code for each domain into its own sub-package under the `domain` package and have the domains separated this way. If you're thinking about separating the ports and adapters per domain, however, be careful, because this quickly turns into a mapping nightmare. More about this in chapter 13.

As with every structure, it takes discipline to maintain this package structure over the lifetime of a software project. Also, there will be cases when the package structure just does not fit and we see no other way than to widen the architecture/code gap and create a package that does not reflect the architecture.

There is no perfection. But with an expressive package structure, we can at least reduce the gap between code and architecture.

The role of dependency injection

The package structure described previously goes a long way towards achieving a Clean Architecture, but an essential requirement of such an architecture is that the application layer does not have dependencies on the incoming and outgoing adapters, as we have learned in Chapter 3, Inverting Dependencies.

For incoming adapters, such as our web adapter, this is easy since the control flow points in the same direction as the dependency between the adapter and domain code. The adapter simply calls the service within the application layer. In order to clearly bring out the entry points to our application, we'll want to hide the actual services behind port interfaces.

For outgoing adapters, such as our persistence adapter, we have to make use of the Dependency Inversion Principle to turn the dependency against the direction of the control flow.

We have already seen how that works. We create an interface within the application layer, which is implemented by a class within the adapter. Within our Hexagonal Architecture, this interface is a port. The application layer then calls this port interface to call the functionality of the adapter as shown in Figure 4.1.

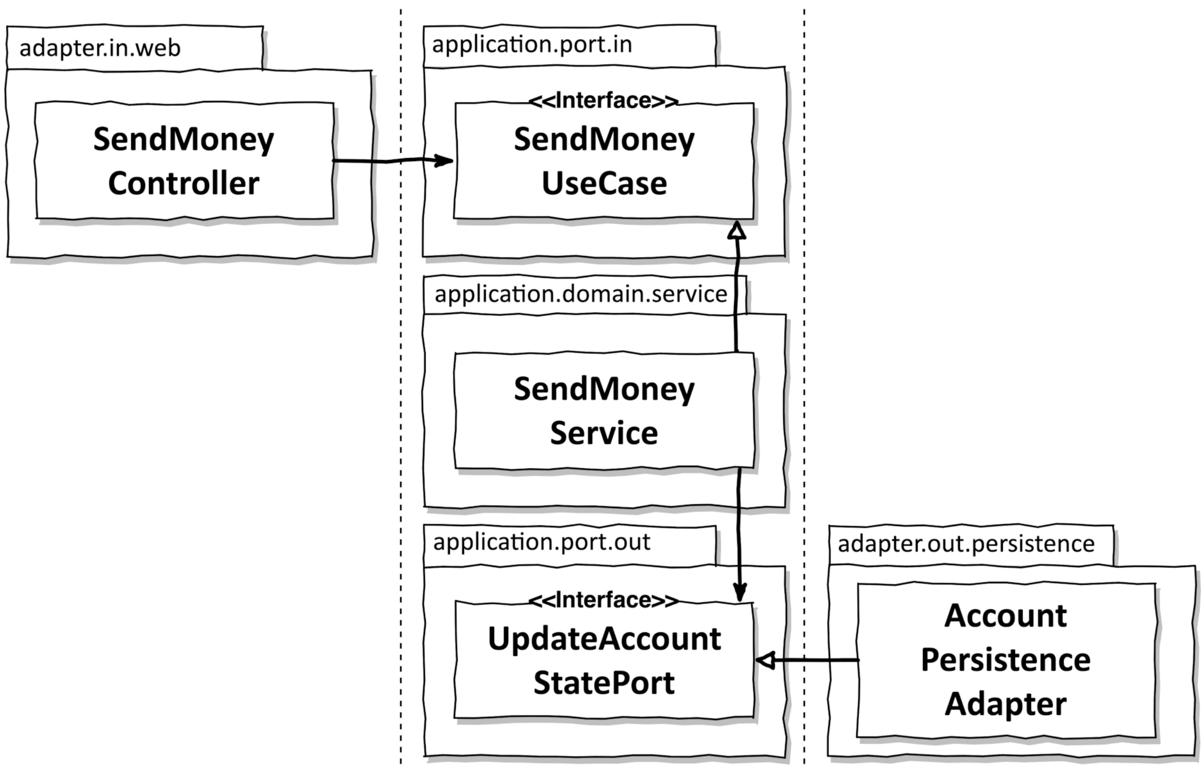

Figure 4.1 - The web controller calls an incoming port, which is implemented by a service. The service calls an outgoing port, which is implemented by an adapter.

But who provides the application with the actual objects that implement the port interfaces? We don't want to instantiate the ports manually within the application layer because we don't want to introduce a dependency on an adapter.

This is where dependency injection comes into play. We introduce a neutral component that has a dependency on all layers. This component is responsible for instantiating most of the classes that make up our architecture.

In the preceding example figure, the neutral dependency injection component would create instances of the `SendMoneyController`, `SendMoneyService`, and `AccountPersistenceAdapter` classes. Since `SendMoneyController` requires a `SendMoneyUseCase`, the dependency injection mechanism will give it an instance of the `SendMoneyService` class during construction. The controller doesn't know that it actually got a `SendMoneyService` instance since it only needs to know the interface.

Similarly, when constructing the `SendMoneyService` instance, the dependency injection mechanism will inject an instance of the `AccountPersistenceAdapter` class, in the guise of the `UpdateAccountStatePort` interface. The service never knows the actual class behind the interface.

We'll talk more about initializing an application using the Spring framework as an example in Chapter 10, Assembling the Application.

How does this help me build maintainable software?

We looked at a package structure for a Hexagonal Architecture that takes the actual code structure as close to the target architecture as possible. Finding an element of the architecture in the code is now a matter of navigating down the package structure along the names of certain boxes in an architecture diagram, helping with communication, development and maintenance.

In the following chapters, we'll see this package structure and dependency injection in action as we implement a use case in the application layer, a web adapter, and a persistence adapter.

5. Implementing a Use Case

Let's finally look at how we can manifest the architecture we have discussed in actual code.

Since the application, web, and persistence layers are so loosely coupled in our architecture, we're totally free to model our domain code as we see fit. We can do DDD, implement a rich or anemic domain model, or invent our own way of doing things.

This chapter describes an opinionated way of implementing use cases within the Hexagonal Architecture style we introduced in the previous chapters.

As is fitting for a domain-centric architecture, we'll start with a domain entity and then build a use case around it.

Implementing the domain model

We want to implement the use case of sending money from one account to another. One way to model this in an object-oriented fashion is to create an `Account` entity that allows us to withdraw money from a source account and deposit it into a target account:

```java
package buckpal.application.domain.model;

public class Account {

  private AccountId id;
  private Money baselineBalance;
  private ActivityWindow activityWindow;

  // constructors and getters omitted

  public Money calculateBalance() {
    return Money.add(
            this.baselineBalance,
            this.activityWindow.calculateBalance(this.id));
  }

  public boolean withdraw(Money money, AccountId targetAccountId) {
    if (!mayWithdraw(money)) {
      return false;
    }
```

```java

    Activity withdrawal = new Activity(
        this.id,
        this.id,
        targetAccountId,
        LocalDateTime.now(),
        money);
    this.activityWindow.addActivity(withdrawal);
    return true;
  }

  private boolean mayWithdraw(Money money) {
    return Money.add(
        this.calculateBalance(),
        money.negate())
        .isPositive();
  }

  public boolean deposit(Money money, AccountId sourceAccountId) {
    Activity deposit = new Activity(
        this.id,
        sourceAccountId,
        this.id,
        LocalDateTime.now(),
        money);
    this.activityWindow.addActivity(deposit);
    return true;
  }
}
```

The `Account` entity provides the current snapshot of an actual account. Every withdrawal from and deposit to an account is captured in an `Activity` entity. Since it would not be wise to always load *all* activities of an account into memory, the `Account` entity only holds a window of the last few days or weeks of activities, captured in the `ActivityWindow` value object.

To still be able to calculate the current account balance, the `Account` entity additionally has the `baselineBalance` attribute, representing the balance the account had just before the first activity of the activity window. The total balance then is the baseline balance plus the balance of all activities in the window.

With this model, withdrawing and depositing money into an account is a matter of adding a new activity to the activity window, as is done in the `withdraw()` and `deposit()` methods. Before we can withdraw, we check the business rule that says that we cannot overdraw an account.

Now that we have an `Account` that allows us to withdraw and deposit money, we can move outward to build a use case around it.

A use case in a nutshell

First, let's discuss what a use case actually does. Usually, it follows these steps:

1. Take the input.
2. Validate the business rules.
3. Manipulate the model state.
4. Return the output.

A use case takes input from an incoming adapter. You might wonder why I didn't call the step "Validate input". The answer is that I believe use case code should only be concerned with domain logic and we shouldn't pollute it with input validation. So, we'll do input validation somewhere else, as we'll see shortly.

The use case is, however, responsible for validating *business rules*. It shares this responsibility with the domain entities. We'll discuss the distinction between input validation and business rule validation later in this chapter.

If the business rules were satisfied, the use case then manipulates the state of the model in one way or another, based on the input. Usually, it will change the state of a domain object and pass this new state to a port implemented by the persistence adapter to be persisted. If the use case drives other side effects than persistence, it invokes an appropriate adapter for each side effect.

The last step is to translate the return value from the outgoing adapter into an output object, which will be returned to the calling adapter.

With these steps in mind, let's see how we can implement our "Send Money" use case.

To avoid the problem of broad services discussed in Chapter 2, What's Wrong with Layers?, we'll create a separate service class for each use case instead of putting all use cases into a single service class.

Here's a teaser:

```java
package buckpal.application.domain.service;

@RequiredArgsConstructor
@Transactional
public class SendMoneyService implements SendMoneyUseCase {

  private final LoadAccountPort loadAccountPort;
  private final UpdateAccountStatePort updateAccountStatePort;

  @Override
  public boolean sendMoney(SendMoneyCommand command) {
    // TODO: validate business rules
    // TODO: manipulate model state
    // TODO: return output
  }
}
```

The service implements the `SendMoneyUseCase` incoming port interface and calls the `LoadAccountPort` outgoing port interface to load an account and the `UpdateAccountStatePort` port to persist an updated account state in the database.

The service also sets the boundary for a database transaction, as implied by the `@Transactional` annotation. More about this in Chapter 7, Implementing a Persistence Adapter.

Figure 5.1 provides a visual overview of the relevant components.

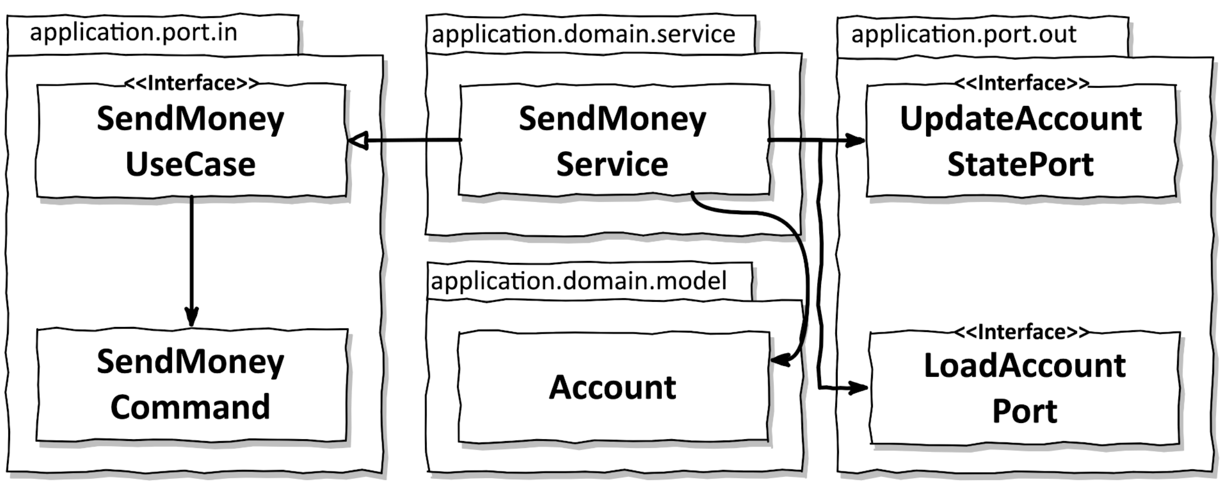

Figure 5.1 - A service implements a use case, modifies the domain model and calls an outgoing port to persist the modified state.

Note that `UpdateAccountStatePort` and `LoadAccountPort` in this example are port interfaces implemented by a persistence adapter. If they are often used together, we could also combine them into a broader interface. We could even call that interface `AccountRepository` to stick with the DDD language. In this example, and in the rest of the book, I chose to use the name "Repository" only in the persistence adapter, but you may choose different names!

Let's take care of those `TODO` comments we left in the preceding code.

Validating input

Now we're talking about validating input, even though I just claimed that it's not a responsibility of a use case class. I still think, however, that it belongs in the application layer, so this is the place to discuss it.

Why not let the calling adapter validate the input before sending it to the use case? Well, do we want to trust the caller to have validated everything as needed for the use case? Also, the use case might be called by more than one adapter, so the validation would have to be implemented by each adapter, and one might get it wrong or forget it altogether.

The application layer should care about input validation because, well, otherwise it might get invalid input from outside the application core. This might cause damage to the state of our model.

But where do we put the input validation, if not in the use case class?

We'll let the *input model* take care of it. For the "Send Money" use case, the input model is the `SendMoneyCommand` class we have already seen in the previous code example. More precisely, we'll do the validation it within the constructor:

```
package buckpal.application.port.in;

public record SendMoneyCommand(
        AccountId sourceAccountId,
        AccountId targetAccountId,
        Money money) {

    public SendMoneyCommand(
            AccountId sourceAccountId,
            AccountId targetAccountId,
            Money money) {
        requireNonNull(sourceAccountId);
        requireNonNull(targetAccountId);
        requireNonNull(money);
        requireGreaterThan(money, 0);
        this.sourceAccountId = sourceAccountId;
        this.targetAccountId = targetAccountId;
        this.money = money;

```

```
    }
}
```

To send money, we need the IDs of the source and target account and the amount of money that is to be transferred. None of the parameters may be null and the amount must be greater than zero. If any of these conditions is violated, we simply refuse object creation by throwing an exception during construction.

By using a Java record to implement `SendMoneyCommand`, we make it immutable. So, once constructed successfully, we can be sure that the state is valid and cannot be changed to something invalid.

Since `SendMoneyCommand` is part of the use cases' API, it's located in the incoming port package. Thus, the validation remains in the core of the application (at the edge of the hexagon of our architecture) but does not pollute the sacred use case code.

But do we really want to implement each validation check by hand when there are libraries that can do the dirty work for us? I have often heard statements such as "You shouldn't use libraries in your model classes." There's wisdom in reducing dependencies to a minimum, of course, but if we can get away with a small-footprint dependency that saves us time, then why not use it? Let's explore what this might look like with Java's Bean Validation API.[23]

Bean Validation allows us to express the validation rules we need as annotations on the fields of a class:

```java
package buckpal.application.port.in;

public record SendMoneyCommand(
        @NotNull AccountId sourceAccountId,
        @NotNull AccountId targetAccountId,
        @NotNull @PositiveMoney Money money) {

    public SendMoneyCommand(
            AccountId sourceAccountId,
            AccountId targetAccountId,
            Money money) {
        this.sourceAccountId = sourceAccountId;
        this.targetAccountId = targetAccountId;
        this.money = money;
        Validator.validate(this);
    }
}
```

The `Validator` class provides the `validate()` method which we simply call as the last statement in the constructor. This will evaluate the Bean Validation annotations on the fields (`@NotNull`, in this

[23] https://beanvalidation.org/

case) and throw an exception in case of a violation. If the default Bean Validation annotations are not expressive enough for a certain validation, we can implement our own annotations and validators as we did with the `@PositiveMoney` annotation.[24]

The implementation of the `Validator` class might look like this:

```
public class Validator {

    private final static jakarta.validation.Validator validator =
            Validation.buildDefaultValidatorFactory()
                    .getValidator();

    /**
     * Evaluates all Bean Validation annotations on the subject.
     */
    public static <T> void validate(T subject) {
        Set<ConstraintViolation<T>> violations = validator.validate(subject);
        if (!violations.isEmpty()) {
            throw new ConstraintViolationException(violations);
        }
    }
}
```

With validation located in the input model, we have created an anti-corruption layer around our use case implementations. This is not a layer in the sense of a layered architecture, calling the next layer below it, but instead a thin, protective screen around our use cases that bounces bad input back to the caller.

Note that the term “command” as used in the `SendMoneyCommand` class does not match with the common interpretation of the “command pattern”.[25] In the command pattern, a command is executable, that is, it has a method called `execute()` that actually invokes the use case. In our case, the command is just a data transfer object that transfers the required parameters to the use case service that executes the command. We could call it `SendMoneyDTO` instead, but I like the term “command” to make it very clear that we’re changing the model state with this use case.

The power of constructors

Our input model, `SendMoneyCommand`, puts a lot of responsibility on its constructor. Since the class is immutable, the constructor’s argument list contains a parameter for each attribute of the class. And since the constructor also validates the parameters, it’s not possible to create an object with an invalid state.

[24] You can find the full code implementing the `@PositiveMoney` annotation and validator in the GitHub repository.
[25] https://en.wikipedia.org/wiki/Command_pattern

In our case, the constructor has only three parameters. What if we had more parameters? Couldn't we use the Builder pattern to make it more convenient to use? We could make the constructor with the long parameter list private and hide the call to it in the `build()` method of our builder. Then, instead of having to call a constructor with 20 parameters, we could build an object like this:

```
new SendMoneyCommandBuilder()
    .sourceAccountId(new AccountId(41L))
    .targetAccountId(new AccountId(42L))
    // ... initialize many other fields
    .build();
```

We could still let our constructor do the validation so that the builder cannot construct an object with an invalid state.

Sound good? Think about what happens if we have to add another field to `SendMoneyCommandBuilder` (which will happen quite a few times in the lifetime of a software project). We add the new field to the constructor and to the builder. Then, a colleague (or a phone call, an email, a butterfly…) interrupts our train of thought. After the break, we go back to coding and *forget to add the new field to the code that calls the builder to create an object.*

We don't get a word of warning from the compiler about trying to create an immutable object in an invalid state! Sure, at runtime - hopefully in a unit test - our validation logic will still kick in and throw an error because we missed a parameter.

But if we use the constructor directly instead of hiding it behind a builder, each time a new field is added or an existing field is removed, we can just follow the trail of compile errors to reflect that change in the rest of the codebase.

Long parameter lists can even be formatted nicely, and good IDEs help with parameter name hints:

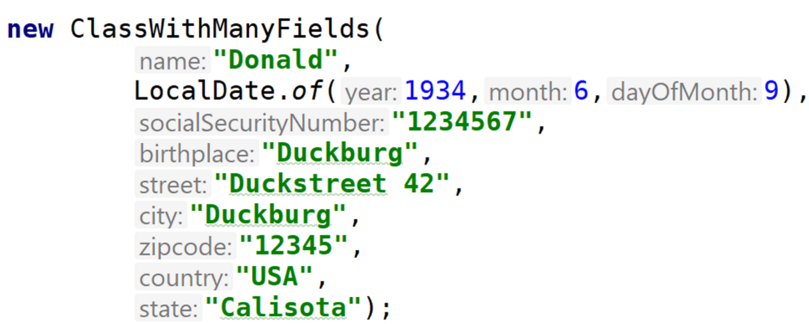

Figure 5.2 - The IDE shows parameter name hints in parameter lists to help us not to get lost.

To make the preceding code even more readable and safer to work with, we can introduce immutable value objects to replace some of the primitives we used as constructor parameters. A value object is an object whose value is its identity. Two value objects with the same value are considered the same. Instead of passing the street, city, zipcode, country, and state separately, we could combine them to an `Address` value object, for example, because they belong together. We could even go a step further and create `City` and `ZipCode` value objects, for example. This would reduce the chance of confusing one `String` parameter with another, because the compiler would complain if we tried to pass a `City` into a `ZipCode` parameter and vice versa.

There are cases where a builder may be the better solution, though. If some parameters in `ClassWithManyFields` from the preceding example were optional, for example, we would have to pass `null` values into the constructor, which is ugly at best. A builder would allow us to define only the required parameters. But if using builders, we should make very sure that the `build()` method fails loudly when we forget to define a required parameter, because the compiler doesn't check that for us!

Different input models for different use cases

We might be tempted to use the same input model for different use cases. Let's consider the "Register Account" and "Update Account Details" use cases. Both will initially need almost the same input, namely some account details such as a username and email address.

The "Update" use case will need the ID of the account that needs to be updated, however, while the "Register" use case does not. If both use cases use the same input model, we will always have to pass a `null` account ID into the "Register" use case. This is annoying at best, and detrimental at worst, because both use cases are coupled to evolve together, now.

Allowing `null` as a valid state of a field in our immutable command object is a code smell by itself. But more importantly, how are we handling input validation now? Validation has to be different for the register and update use cases, since one needs an ID and the other doesn't. We'd have to build custom validation logic into the use cases themselves, polluting our sacred business code with input validation concerns.

Also, what do we do if the account ID field accidentally has a non-null value in the "Register Account" use case? Do we throw an error? Do we simply ignore it? These are the questions the maintenance engineers - including future us - will ask when seeing the code.

A dedicated input model for each use case makes the use case much clearer and also decouples it from other use cases, preventing unwanted side effects. It comes with a cost, however, because we have to map incoming data into different input models for different use cases. We'll discuss this mapping strategy along with other mapping strategies in Chapter 9, Mapping between Boundaries.

Validating business rules

While validating input is not part of the use case logic, validating business rules definitely is. Business rules are the core of the application and should be handled with appropriate care. But when are we dealing with input validation and when with a business rule?

A very pragmatic distinction between the two is that validating a business rule requires access to the current state of the domain model while validating input does not. Input validation can be implemented declaratively, as we did with the `@NotNull` annotations previously, while a business rule needs more context.

We might also say that input validation is a *syntactic* validation, while a business rule is a *semantic* validation in the context of a use case.

As per the previous definition, this is a business rule since it needs access to the current state of the model to check the balance of the source account.

In contrast, the rule "the transfer amount must be greater than zero" can be validated without access to the model and thus can be implemented as part of the input validation.

I'm aware that this distinction may be subject to debate. You might argue that the transfer amount is so important that validating it should be considered a business rule in any case.

The distinction helps us, however, to place certain validations within the codebase and easily find them again later on. It's as simple as answering the question if the validation needs access to the current model state or not. This not only helps us to implement the rule in the first place, but it also helps the future maintenance engineer to find it again. It's also a great example of my claim from Chapter 1, Maintainability, that maintainability supports decision-making.

So, how do we implement a business rule?

The best way is to put the business rules into a domain entity as we did for the rule "the source account must not be overdrawn":

```
package buckpal.application.domain.model;

public class Account {

  // ...

  public boolean withdraw(
          Money money,
          AccountId targetAccountId) {
    if (!mayWithdraw(money)) {
      return false;
    }
    // ...
  }
}
```

This way, the business rule is easy to locate and reason about, because it's right next to the business logic that requires this rule to be honored.

If it's not feasible to validate a business rule in a domain entity, we can do it in the use case code before it starts working on the domain entities:

```
package buckpal.application.domain.service;

@RequiredArgsConstructor
@Transactional
public class SendMoneyService implements SendMoneyUseCase {

  // ...

  @Override
  public boolean sendMoney(SendMoneyCommand command) {
    requireAccountExists(command.sourceAccountId());
    requireAccountExists(command.targetAccountId());
    ...
  }
}
```

We call a method that does the actual validation and throws a dedicated exception in the case that this validation fails. The adapter interfacing with the user can then display this exception to the user as an error message or handle it in any other way it deems fit.

In the preceding case, the validation simply checks whether the source and target accounts actually exist in the database. More complex business rules might require us to load the domain model from the database first and then do some checks on its state. If we have to load the domain model anyway, we should implement the business rule in the domain entities themselves, as we did with the rule "the source account must not be overdrawn".

Rich versus anemic domain model

Our architecture style leaves open how to implement our domain model. This is a blessing, because we can do what seems right in our context, and a curse, because we don't have any guidelines to help us.

A frequent discussion is whether to implement a rich domain model following the DDD philosophy or an "anemic" domain model. Let's discuss how each of those fits into our architecture.

In a rich domain model, as much of the domain logic as possible is implemented within the entities at the core of the application. The entities provide methods to change state and only allow changes that are valid according to the business rules. This is the way we pursued with the `Account` entity previously. Where is our use case implementation in this scenario?

In this case, our use case serves as an entry point to the domain model. A use case then only represents the intent of the user and translates it into orchestrated method calls to the domain entities which do the actual work. Many of the business rules are located in the entities instead of the use case implementation.

The "Send Money" use case service would load the source and target account entities, call their `withdraw()` and `deposit()` methods, and send them back to the database.[26]

In an "anemic" domain model, the entities themselves are very thin. They usually only provide fields to hold the state and getter and setter methods to read and change the state. They don't contain any domain logic.

This means that the domain logic is implemented in the use case classes. They are responsible for validating business rules, changing the state of the entities and passing them into the outgoing ports responsible for storing them in the database. The "richness" is contained within the use cases instead of the entities.

Either style, and any number of other styles, can be implemented using the architecture approach discussed in this book. Feel free to choose the one that fits your needs.

Different output models for different use cases

Once the use case has done its work, what should it return to the caller?

Similar to the input, it has benefits if the output is as specific to the use case as possible. The output should only include the data that is really needed for the caller to work.

In the example code of the "Send Money" use case, we return a `boolean`. This is the minimal and most specific value we could possibly return in this context.

We might be tempted to return a complete `Account` with the updated entity to the caller. Perhaps the caller is interested in the new balance of the account?

But do we really want to make the "Send Money" use case return this data? Does the caller really need it? If so, shouldn't we create a dedicated use case for accessing that data that can be used by different callers?

There is no single right answer to these questions. But we should ask them to try to keep our use cases as specific as possible. When in doubt, return as little as possible.

Sharing the same output model between use cases also tends to tightly couple those use cases. If one of the use cases needs a new field in the output model, the other use cases have to handle this field as well, even if it's irrelevant to them. Shared models tend to grow tumorously for multiple reasons in the long run. Applying the Single Responsibility Principle and keeping models separated helps in decoupling use cases.

For the same reason we might want to resist the temptation to use our domain entities as the output model. We don't want our domain entities to change for more reasons than necessary. However, we'll talk more about using entities as input or output models in Chapter 11, Taking Shortcuts Consciously.

[26]Actually, the use case would also have to make sure that no other money transfer to and from the source and target account is happening at the same time to avoid overdrawing an account.

What about read-only use cases?

As of now, we have discussed how we might implement a use case that modifies the state of our model. How do we go about implementing read-only cases?

Let's assume the UI needs to display the balance of an account. Do we create a specific use case implementation for this?

It's awkward to talk of use cases for read-only operations like this one. Sure, the UI needs the data for a use case we might call "View Account Balance", but in some cases calling this a "use case" is a bit artificial. If this is considered a use case in the context of the project, by all means we should implement it just like the other ones.

From the viewpoint of the application core, however, this is a simple query for data. So, if it's not considered a use case in the context of the project, we can implement it as a query to set it apart from the real use cases.

One way of doing this within our architecture style is to create a dedicated incoming port for the query and implement it in a "query service":

```
package buckpal.application.domain.service;

@RequiredArgsConstructor
class GetAccountBalanceService implements GetAccountBalanceUseCase {

    private final LoadAccountPort loadAccountPort;

    public Money getAccountBalance(GetAccountBalanceQuery query) {
        return loadAccountPort.loadAccount(query.accountId(), LocalDateTime.now())
                .calculateBalance();
    }
}
```

The query service acts just as our "command" use case services do. It implements an incoming port we named `GetAccountBalanceUseCase` and calls the outgoing port, `LoadAccountPort`, to actually load the data from the database. It's using the `GetAccountBalanceQuery` type as its input model.

This way, read-only queries are clearly distinguishable from modifying use cases (or "commands") in our codebase. We just have to look at the names of the input types to know which we're dealing with. This plays nicely with concepts like Command-Query Separation (CQS) and Command-Query Responsibility Segregation (CQRS).

In the preceding code, the service doesn't really do any work other than passing the query on to the outgoing port. If we use the same model across layers, we can take a shortcut and let the client call the outgoing port directly. We'll talk about this shortcut in Chapter 11, Taking Shortcuts Consciously.

How does this help me build maintainable software?

Our architecture lets us implement the domain logic as we see fit, but if we model the input and output of our use cases independently, we avoid unwanted side effects.

Yes, it's more work than just sharing models between use cases. We have to introduce a separate model for each use case and map between this model and our entities.

But use case-specific models allow for a crisp understanding of a use case, making it easier to maintain in the long run. Also, they allow multiple developers to work on different use cases in parallel without stepping on each other's toes.

Together with tight input validation, use case-specific input and output models go a long way toward a maintainable codebase.

In the next chapter, we're going a step "outward" from the center of our application and will explore building a web adapter that provides a channel for users to talk to our use case.

6. Implementing a Web Adapter

Most applications today have some kind of web interface - either a UI that we can interact with via web browser or an HTTP API that other systems can call to interact with our application.

In our target architecture, all communication with the outside world goes through adapters. So, let's discuss how we can implement an adapter that provides such a web interface.

Dependency Inversion

Figure 6.1 gives a zoomed-in view of the architecture elements that are relevant to our discussion of a web adapter - the adapter itself and the ports through which it interacts with our application core:

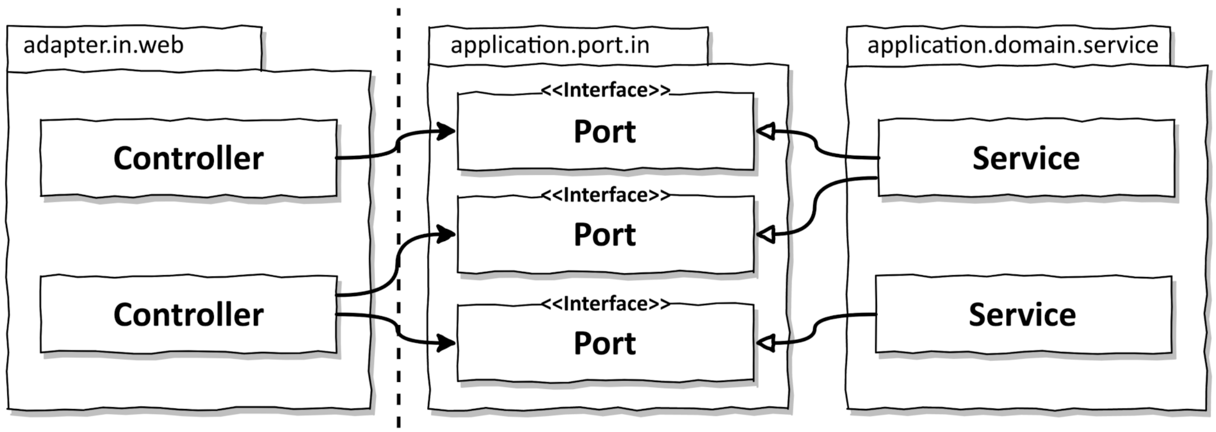

Figure 6.1 - An incoming adapter talks to the application layer through dedicated incoming ports which are interfaces implemented by the domain services.

The web adapter is a "driving" or "incoming" adapter. It takes requests from the outside and translates them into calls to our application core, telling it what to do. The control flow goes from the controllers in the web adapter to the services in the application layer.

The application layer provides specific ports through which the web adapter may communicate. Each port is what I have called a "use case" in the previous chapter, and it is implemented by a domain service in the application layer.

If we look closer, we notice that this is the Dependency Inversion Principle in action. Since the control flow goes from left to right, we could just as well let the web adapter call the use cases directly, as shown in Figure 6.2.

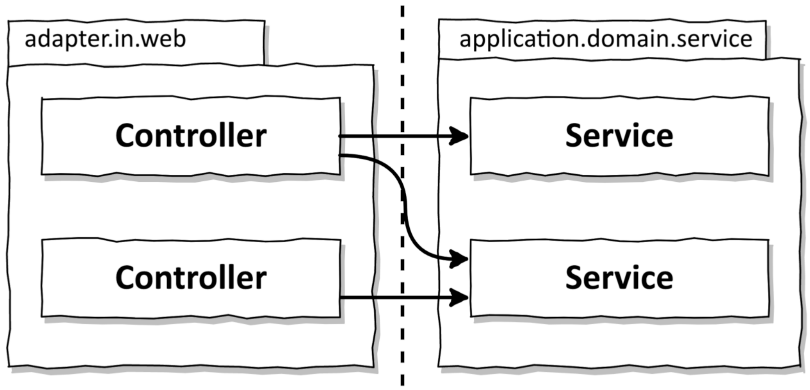

Figure 6.2 - We can remove the port interfaces and call the services directly.

So why do we add another layer of indirection between the adapter and the use cases? The reason is that the ports are a specification of the places where the outside world can interact with our application core. By having ports in place, we know exactly which communication with the outside world takes place, which is valuable information for any maintenance engineer working on your legacy codebase.

Knowing the ports that drive the application also lets us build a test driver for the application. This test driver is an adapter that calls the input ports to simulate and test certain usage scenarios - more about testing in Chapter 8, Testing Architecture Elements.

Having talked about the importance of input ports, one of the shortcuts we'll talk about in Chapter 11, Taking Shortcuts Consciously, is just leaving the incoming ports out and calling the application services directly.

One question remains, though, which is relevant for highly interactive applications. Imagine a server application that sends real-time data to the user's browser via WebSocket. How does the application core send this real-time data to the web adapter, which in turns sends it to the user's browser?

For this scenario, we definitely need a port because, without a port, the application would have to depend on an adapter implementation, breaking our efforts to keep the application free from dependencies on the outside. This port must be implemented by the web adapter and called by the application core, as depicted in Figure 6.3.

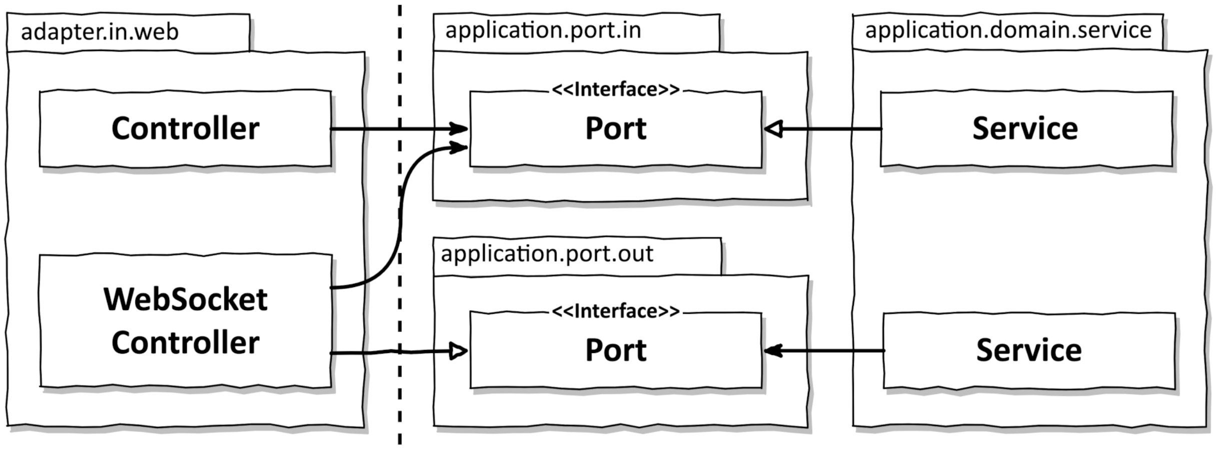

Figure 6.3 - If an application must actively notify a web adapter, we need to go through an outgoing port to keep the dependencies in the right direction.

The `WebSocketController` on the left implements the port interface in the `out` package, and services in the application core can call this port to send real-time data to the user's browser.

Technically speaking, this would be an outgoing port and make the web adapter an incoming *and* outgoing adapter. But there is no reason that the same adapter can not be both at the same time.

For the rest of this chapter we'll assume that the web adapter is an incoming adapter only, since this is the most common case.

Responsibilities of a web adapter

What does a web adapter actually do? Let's say we want to provide a REST API for our BuckPal application. Where do the responsibilities of the web adapter start and where do they end?

A web adapter usually does these things:

1. Map the incoming HTTP request to objects.
2. Perform authorization checks.
3. Validate the input.
4. Map the request objects to the input model of the use case.
5. Call the use case.
6. Map the output of the use case back to HTTP.
7. Return the HTTP response.

First of all, a web adapter must listen to HTTP requests that match certain criteria such as a URL path, HTTP method, and content type. The parameters and the content of a matching HTTP request must then be deserialized into objects we can work with.

Commonly, a web adapter then does an authentication and authorization check and returns an error if it fails.

The state of the incoming objects can then be validated. But haven't we already discussed input validation as a responsibility of the input model to the use cases? Yes, the input model to the use cases should only allow input that is valid in the context of the use cases. But here, we're talking about the input model to the *web adapter*. It might have a completely different structure and semantics from the input model to the *use cases*, so we might have to perform different validations.

I don't advocate implementing the same validations in the web adapter as we have already done in the input model of the use cases. Instead, we should validate that *we can transform the input model of the web adapter into the input model of the use cases*. Anything that prevents us from doing this transformation is a validation error.

This brings us to the next responsibility of a web adapter: to call a certain use case with the transformed input model. The adapter then takes the output of the use case and serializes it into an HTTP response, which is sent back to the caller.

If anything goes wrong on the way and an exception is thrown, the web adapter must translate the error into a message that is sent back to the caller.

That's a lot of responsibilities weighing on the shoulders of our web adapter. But it's also a lot of responsibilities that the application layer should not be concerned with. Anything that has to do with HTTP must not leak into the application layer. If the application core knows that we're dealing with HTTP on the outside, we have lost the option to perform the same domain logic from other incoming adapters that do not use HTTP. In a maintainable architecture, we want to keep options open.

Note that this boundary between the web adapter and application layer comes naturally if we start development with the domain and application layers instead of with the web layer. If we implement the use cases first, without thinking about any specific incoming adapter, we are not tempted to blur the boundary.

Slicing controllers

In most web frameworks - such as Spring MVC in the Java world - we create controller classes that perform the responsibilities we have discussed previously. So, do we build a single controller that answers all requests directed at our application? We don't have to. A web adapter may certainly consist of more than one class.

We should take care, however, to put these classes into the same package hierarchy to mark them as belonging together, as discussed in Chapter 4, Organizing Code.

So, how many controllers do we build? I say we should rather build too many than too few. We should make sure that each controller implements a slice of the web adapter that is as narrow as possible and that shares as little as possible with other controllers.

Let's take the operations on an account entity within our BuckPal application. A popular approach is to create a single `AccountController` that accepts requests for all operations that relate to accounts. A Spring controller providing a REST API might look like the following code snippet.

```
package buckpal.adapter.in.web;

@RestController
@RequiredArgsConstructor
class AccountController {

  private final GetAccountBalanceUseCase getAccountBalanceUseCase;
  private final ListAccountsQuery listAccountsQuery;
  private final LoadAccountQuery loadAccountQuery;

  private final SendMoneyUseCase sendMoneyUseCase;
  private final CreateAccountUseCase createAccountUseCase;

  @GetMapping("/accounts")
  List<AccountResource> listAccounts(){
    ...
  }

  @GetMapping("/accounts/{id}")
  AccountResource getAccount(@PathVariable("id") Long accountId){
    ...
  }

  @GetMapping("/accounts/{id}/balance")
  long getAccountBalance(@PathVariable("id") Long accountId){
    ...
  }

  @PostMapping("/accounts")
  AccountResource createAccount(@RequestBody AccountResource account){
    ...
  }

  @PostMapping("/accounts/send/{sourceAccountId}/{targetAccountId}/{amount}")
  void sendMoney(
      @PathVariable("sourceAccountId") Long sourceAccountId,
      @PathVariable("targetAccountId") Long targetAccountId,
      @PathVariable("amount") Long amount) {
    ...
```

```
  }
}
```

Everything concerning the account resource is in a single class, which feels good. But let's discuss the downsides of this approach.

First, less code per class is a good thing. I have worked on a legacy project where the largest class had 30,000 lines of code.[27] That's no fun. Even if the controller only accumulates 200 lines of code over the years, it's still harder to grasp than 50 lines, even when it's cleanly separated into methods.

The same argument is valid for test code. If the controller itself has a lot of code, there will be a lot of test code. And often, test code is even harder to grasp than production code, because it tends to be more abstract. We also want to make the tests for a certain piece of production code to be easy to find, which is easier in small classes.

Equally important, however, is that putting all operations into a single controller class encourages the reuse of data structures. In the preceding code example, many operations share the `AccountResource` model class. It serves as a bucket for everything that is needed in any of the operations. `AccountResource` probably has an `id` field. This is not needed in the `create` operation and will probably confuse here more than it will help. Imagine that an `Account` has a one-to-many relationship with `User` objects. Do we include those `User` objects when creating or updating an account? Will the users be returned by the list operation? This is an easy example, but in any above-playsize project, we'll ask these questions at some point.

So, I advocate the approach to create a separate controller, potentially in a separate package, for each operation. Also, we should name the methods and classes as close to our use cases as possible:

```java
package buckpal.adapter.in.web;

@RestController
@RequiredArgsConstructor
public class SendMoneyController {

  private final SendMoneyUseCase sendMoneyUseCase;

  @PostMapping(
    "/accounts/sendMoney/{sourceAccountId}/{targetAccountId}/{amount}"
  )
  void sendMoney(
      @PathVariable("sourceAccountId") Long sourceAccountId,
      @PathVariable("targetAccountId") Long targetAccountId,
      @PathVariable("amount") Long amount) {
```

[27] It was actually a conscious architecture decision (by our predecessors, mind you) that lead to those 30,000 lines being in a single class: to change the system at runtime, without re-deployment, it allowed them to upload compiled Java bytecode in a .class file. And it only allowed them to upload *a single file*, so this file had to contain all the code… .

```

    SendMoneyCommand command = new SendMoneyCommand(
        new AccountId(sourceAccountId),
        new AccountId(targetAccountId),
        Money.of(amount));

    sendMoneyUseCase.sendMoney(command);
  }
}
```

We can take primitives as input, as did with `sourceAccountId`, `targetAccountId`, and `amount` in the example. But each controller can also have its own input model. Instead of a generic model such as `AccountResource`, we might then have a model specific to the use case such as `CreateAccountResource` or `UpdateAccountResource`. Those specialized model classes may even be private to the controller's package to prevent accidental re-use. Controllers may still share models, but using shared classes from another package makes us think about it more and perhaps we will find out that we don't need half of the fields and will create our own, after all.

Also, we should think hard about the names of the controllers and services. Instead of `CreateAccount`, for instance, wouldn't `RegisterAccount` be a better name? In our BuckPal application the only way to create an account is for a user to register it. So, we use the word "register" in class names to better convey their meaning. There are certainly cases where the usual suspects (`Create...`, `Update...`, and `Delete...`) sufficiently describe a use case, but we might want to think twice before actually using them.

Another benefit of this slicing style is that it makes parallel work on different operations a breeze. We won't have merge conflicts if two developers work on different operations.

How does this help me build maintainable software?

When building a web adapter to an application we should keep in mind that we're building an adapter that translates the HTTP protocol to method calls on the use cases of our application and translates the results back to HTTP, and does not do any domain logic.

The application layer, on the other hand, should not do HTTP, so we should make sure not to leak HTTP details. This makes the web adapter replaceable with another adapter should the need arise.

When slicing web controllers, we should not be afraid to build many small classes that don't share a model. They're easier to grasp and test, and they support parallel work. It's more work initially to set up such fine-grained controllers, but it will pay off during maintenance.

Having looked at the incoming side of our application, we'll now take a look at the outgoing side and how to implement a persistence adapter.

7. Implementing a Persistence Adapter

In Chapter 2, I complained about a traditional layered architecture and claimed that it promotes "database-driven design", because, ultimately, everything depends on the persistence layer. In this chapter, we'll have a look at how to make the persistence layer a plugin to the application layer to invert this dependency.

Dependency inversion

Instead of a persistence layer, we'll talk about a persistence adapter that provides persistence functionality to the domain services. Figure 7.1 shows how we can apply the Dependency Inversion Principle to do just that.

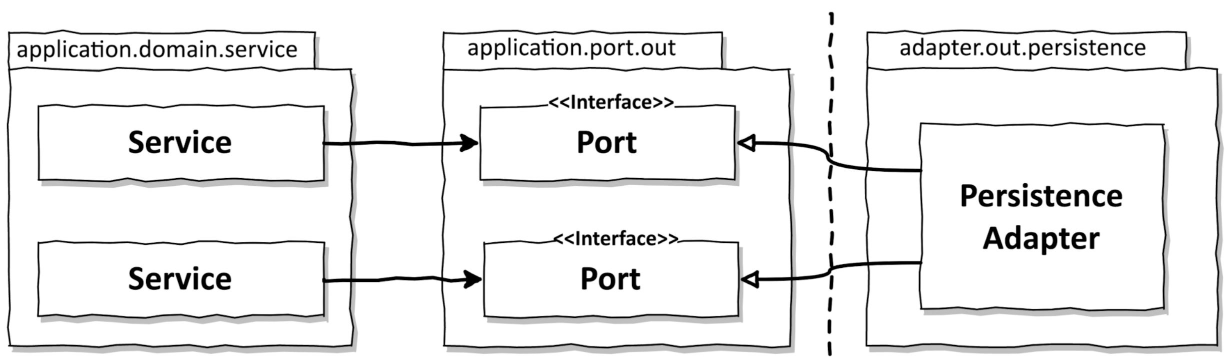

Figure 7.1 - The services from the core use ports to access the persistence adapter.

Our domain services call port interfaces to access persistence functionality. These ports are implemented by a persistence adapter class that does the actual persistence work and is responsible for talking to the database.

In Hexagonal Architecture lingo, the persistence adapter is a "driven" or "outgoing" adapter because it's called by our application and not the other way around.

The ports are effectively a layer of indirection between the domain services and the persistence code. Let's remind ourselves that we're adding this layer of indirection in order to be able to evolve the domain code without having to think about persistence problems, meaning without code dependencies on the persistence layer. Refactoring in persistence code will not lead to a code change in the core.

Naturally, at runtime, we still have a dependency from our application core to the persistence adapter. If we modify code in the persistence layer and introduce a bug, for example, we may still break functionality in the application core. However, as long as the contracts of the ports are fulfilled, we're free to do what we want in the persistence adapter without affecting the core.

Responsibilities of a persistence adapter

Let's have a look at what a persistence adapter usually does:

1. Take the input.
2. Map the input into database format.
3. Send the input to the database.
4. Map the database output into application format.
5. Return the output.

The persistence adapter takes input through a port interface. The input model may be a domain entity or an object dedicated to a specific database operation, as specified by the interface.

It then maps the input model to a format it can work with to modify or query the database. In Java projects, we commonly use the Java Persistence API (JPA) to talk to a database, so we might map the input into JPA entity objects that reflect the structure of the database tables. Depending on the context, mapping the input model into JPA entities may be a lot of work for little gain, so we'll talk about strategies without mapping in Chapter 9, Mapping between Boundaries.

Instead of using JPA or another object-relational mapping framework, we might use any other technique to talk to the database. We might map the input model into plain SQL statements and send these statements to the database, or we might serialize incoming data into files and read them back from there.

The important part is that the input model to the persistence adapter lies within the application core, and not within the persistence adapter itself, so that changes in the persistence adapter don't affect the core.

Next, the persistence adapter queries the database and receives the query results.

Finally, it maps the database answer into the output model expected by the port and returns it. Again, it's important that the output model lies within the application core and not within the persistence adapter to have the dependencies point in the right direction.

Aside from the fact that the input and output models lie in the application core instead of the persistence adapter itself, the responsibilities are not really different from those of a traditional persistence layer.

However, implementing a persistence adapter as described here will inevitably raise some questions that we probably wouldn't ask when implementing a traditional persistence layer, as we're so used to the traditional way that we don't think about them.

Slicing port interfaces

One question that comes to mind when implementing services is how to slice the port interfaces that define the database operations available to the application core.

It's common practice to create a single repository interface that provides all database operations for a certain entity as outlined in Figure 7.2.

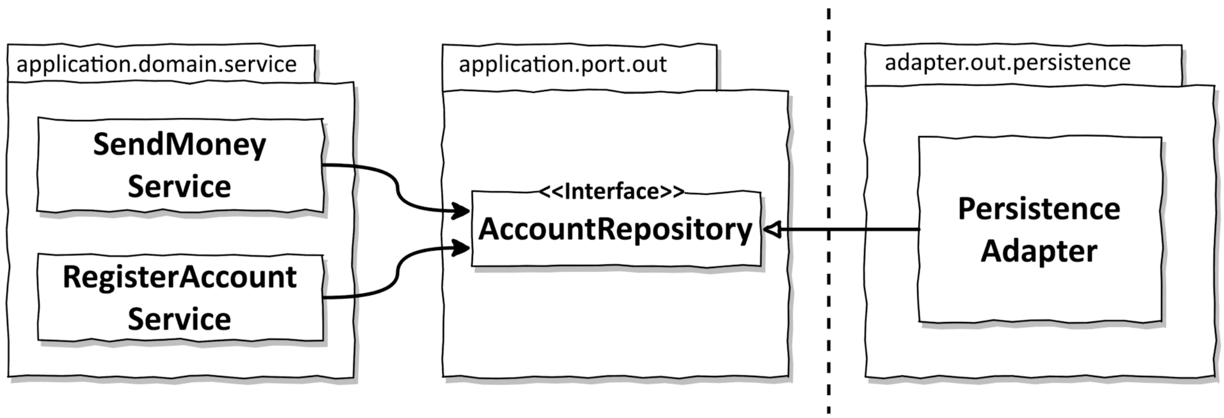

Figure 7.2 - Centralizing all database operations into a single outgoing port interface makes all services depend on methods they don't need.

Each service that relies on database operations will then have a dependency on this single "broad" port interface, even if it uses only a single method from the interface. This means we have unnecessary dependencies in our codebase.

Dependencies on methods that we don't need in our context make the code harder to understand and test. Imagine we're writing a unit test for `RegisterAccountService` from the preceding figure. Which of the methods of the `AccountRepository` interface do we have to create a mock for? We have to first find out which of the `AccountRepository` methods the service actually calls. Having mocked only part of the interface may lead to other problems as the next person working on that test might expect the interface to be completely mocked and run into errors. So, they again have to do some research.

To put it in the words of Robert C. Martin:

> *Depending on something that carries baggage that you don't need can cause you troubles that you didn't expect.*[28]

The Interface Segregation Principle provides an answer to this problem. It states that broad interfaces should be split into specific ones so that clients only know the methods they need.

If we apply this to our outgoing ports, we might get a result as shown in Figure 7.3.

[28] *Clean Architecture* by Robert C. Martin, page 86.

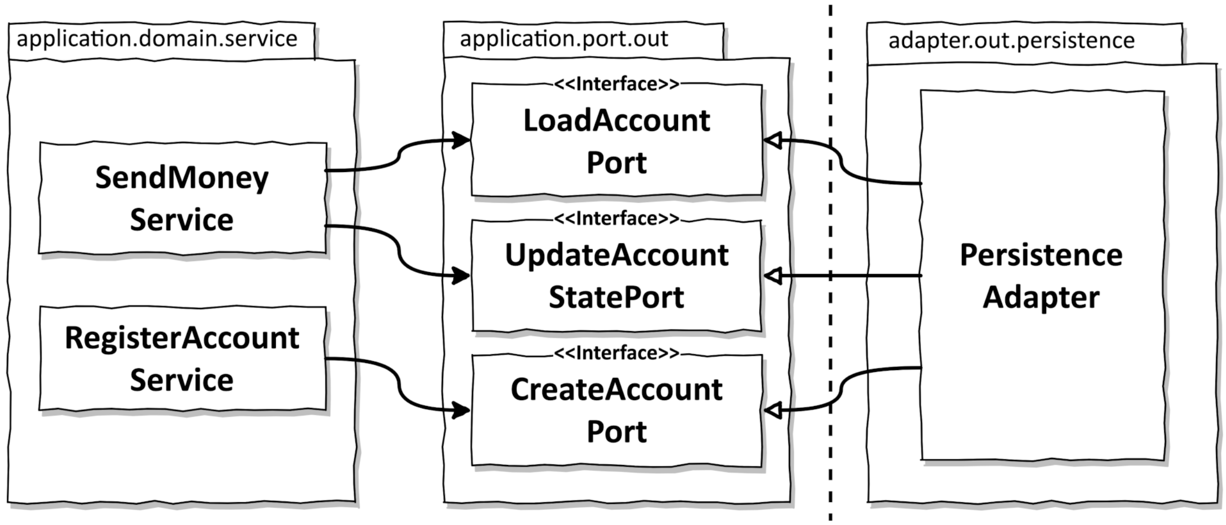

Figure 7.3 - Applying the Interface Segregation Principle removes unnecessary dependencies and makes the existing dependencies more visible.

Each service now only depends on the methods it actually needs. What's more, the names of the ports clearly state what they're about. In a test, we no longer have to think about which methods to mock, since most of the time, there is only one method per port.

Having very narrow ports such as these makes coding a plug and play experience. When working on a service, we just "plug in" the ports we need. There is no baggage to carry around.

Of course, the "one method per port" approach may not be applicable in all circumstances. There may be groups of database operations that are so cohesive and often used together that we may want to bundle them together in a single interface.

Slicing persistence adapters

In the preceding figures, we saw a single persistence adapter class that implements all persistence ports. There is no rule, however, that forbids us to create more than one persistence adapter, as long as all persistence ports are implemented.

We might choose, for instance, to implement one persistence adapter per group of domain entities for which we need persistence operations (or "aggregate" in Domain-Driven Design lingo), as shown in Figure 7.4.

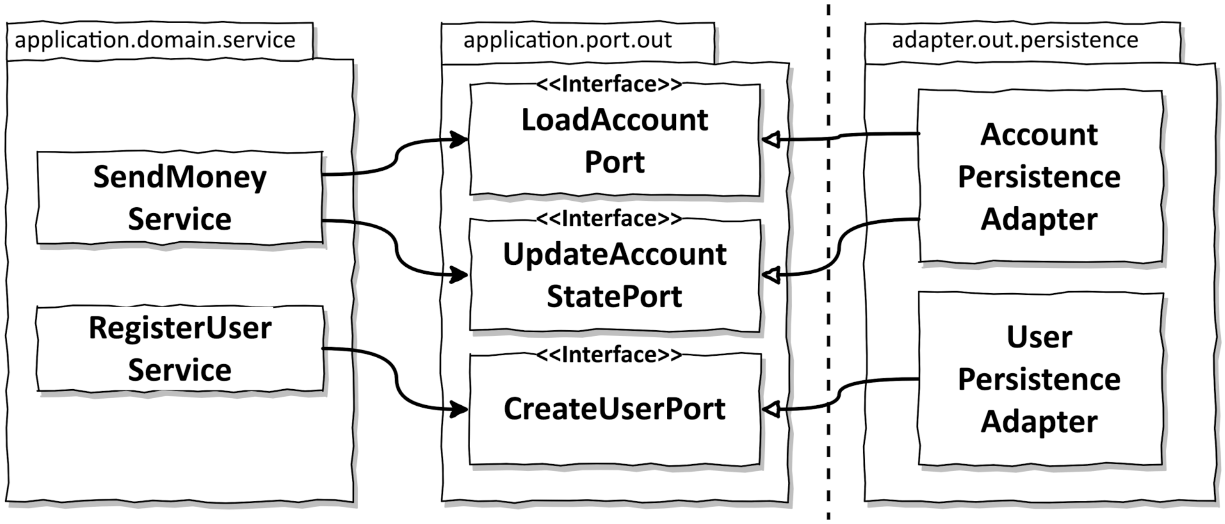

Figure 7.4 - We can create multiple persistence adapters, one for each aggregate.

This way, our persistence adapters are automatically sliced along the seams of the domain that we support with persistence functionality.

We might split our persistence adapters into even more classes, for instance, when we want to implement a couple of persistence ports using JPA or another object-relational mapper and some other ports using plain SQL for better performance. We might then create one JPA adapter and one plain SQL adapter, each implementing a subset of the persistence ports.

Remember that our domain code doesn't care about which class ultimately fulfills the contracts defined by the persistence ports. We're free to do as we see fit in the persistence layer, as long as all ports are implemented.

The "one persistence adapter per aggregate" approach is also a good foundation to separate the persistence needs for multiple bounded contexts in the future. Say, after a time we identify a bounded context responsible for use cases around billing. Figure 7.5 adds that new domain to the application.

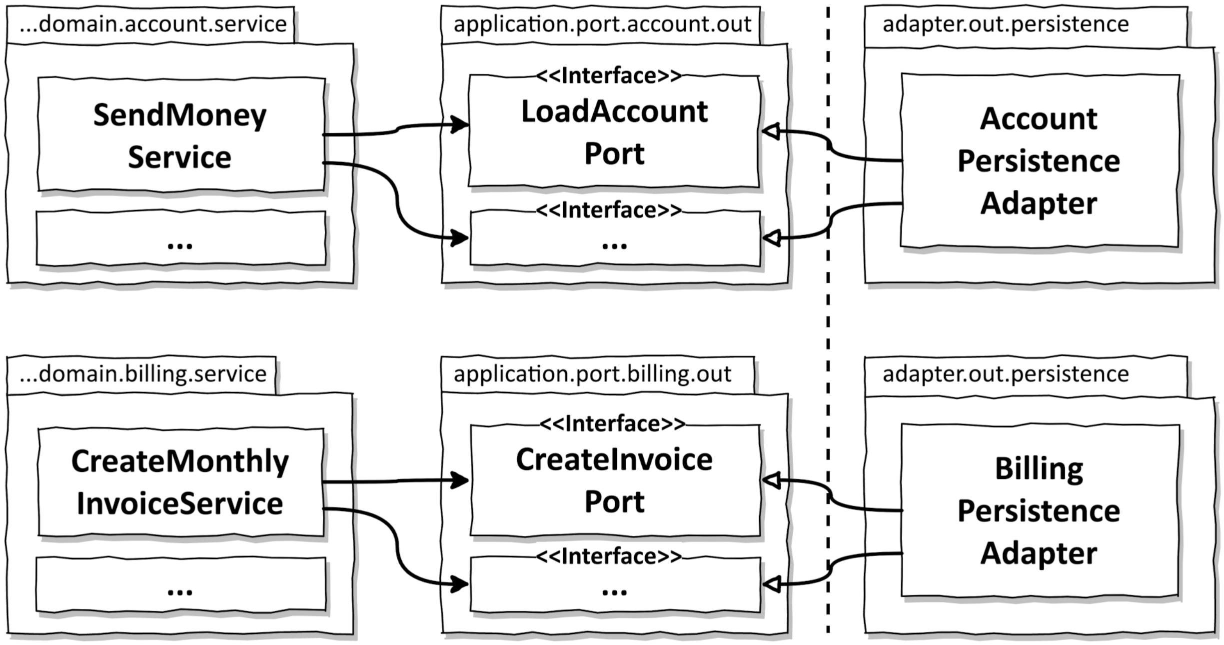

Figure 7.5 - If we want to create hard boundaries between bounded contexts, each bounded context should have its own persistence adapter(s).

Each bounded context has its own persistence adapter (or potentially more than one, as described previously). The term "bounded context" implies boundaries, which means that services of the `account` context may not access persistence adapters of the `billing` context and vice versa. If one context needs something of the other, they can call each other's domain services, or we can introduce an application service as a coordinator between the bounded contexts. We will talk more about this topic in Chapter 13, Managing Multiple Bounded Contexts.

An example with Spring Data JPA

Let's have a look at a code example that implements `AccountPersistenceAdapter` from the preceding figures. This adapter will have to save and load accounts to and from the database. We already saw the `Account` entity in Chapter 5, Implementing a Use Case, but here is its skeleton again for reference:

```
package buckpal.application.domain.model;

public class Account {

  private AccountId id;
  private Money baselineBalance;
  private ActivityWindow activityWindow;

  // constructors and getters omitted
```

```java

  public static Account withoutId(
          Money baselineBalance,
          ActivityWindow activityWindow) {
    return new Account(null, baselineBalance, activityWindow);
  }

  public static Account withId(
          AccountId accountId,
          Money baselineBalance,
          ActivityWindow activityWindow) {
    return new Account(accountId, baselineBalance, activityWindow);
  }

  public Money calculateBalance() {
    // ...
  }

  public boolean withdraw(Money money, AccountId targetAccountId) {
    // ...
  }

  public boolean deposit(Money money, AccountId sourceAccountId) {
    // ...
  }
}
```

Note that the `Account` class is not a simple data class with getters and setters but instead tries to be as immutable as possible. It only provides factory methods that create an `Account` in a valid state, and all mutating methods do some validation, such as checking the account balance before withdrawing money, so that we cannot create an invalid domain model.

We'll use Spring Data JPA to talk to the database, so we also need `@Entity`-annotated classes to represent the database state of an account:

```java
package buckpal.adapter.out.persistence;

@Entity
@Table(name = "account")
@Data
@AllArgsConstructor
@NoArgsConstructor
class AccountJpaEntity {

  @Id
  @GeneratedValue
  private Long id;

}
```

```java
package buckpal.adapter.out.persistence;

@Entity
@Table(name = "activity")
@Data
@AllArgsConstructor
@NoArgsConstructor
class ActivityJpaEntity {

  @Id
  @GeneratedValue
  private Long id;

  @Column private LocalDateTime timestamp;
  @Column private long ownerAccountId;
  @Column private long sourceAccountId;
  @Column private long targetAccountId;
  @Column private long amount;

}
```

The state of an account consists merely of an ID at this stage. Later, additional fields such as a user ID may be added. More interesting is `ActivityJpaEntity`, which contains all the activities of a specific account. We could have connected `ActivitiyJpaEntity` with the `AccountJpaEntity` via JPA's `@ManyToOne` or `@OneToMany` annotations to mark the relation between them, but we have opted to leave this out for now, as it adds side effects to the database queries. In fact, at this stage, it would probably be easier to use a simpler object-relational mapper than JPA to implement the persistence adapter, but we will use it anyway because we think we might need it in the future.[29]

Next, we will use Spring Data to create repository interfaces that provide basic CRUD (create, read, updated, delete) functionality out of the box as well as custom queries to load certain activities from the database:

```
interface AccountRepository extends JpaRepository<AccountJpaEntity, Long> {
}
```

```
interface ActivityRepository extends
        JpaRepository<ActivityJpaEntity, Long> {

    @Query("""
            select a from ActivityJpaEntity a
            where a.ownerAccountId = :ownerAccountId
            and a.timestamp >= :since
            """)
    List<ActivityJpaEntity> findByOwnerSince(
            @Param("ownerAccountId") long ownerAccountId,
            @Param("since") LocalDateTime since);

    @Query("""
            select sum(a.amount) from ActivityJpaEntity a
            where a.targetAccountId = :accountId
            and a.ownerAccountId = :accountId
            and a.timestamp < :until
            """)
    Optional<Long> getDepositBalanceUntil(
            @Param("accountId") long accountId,
            @Param("until") LocalDateTime until);

    @Query("""
            select sum(a.amount) from ActivityJpaEntity a
            where a.sourceAccountId = :accountId
            and a.ownerAccountId = :accountId
            and a.timestamp < :until
            """)
    Optional<Long> getWithdrawalBalanceUntil(
            @Param("accountId") long accountId,
```

[29]Does that sound familiar to you? You choose JPA as an OR mapper because it's the thing people use for this problem. A couple of months into development, you curse eager and lazy loading and the caching features, wishing for something simpler. JPA is a great tool, but for many problems, simpler solutions may be, well, simpler. Take a look at Spring Data JDBC or jOOQ as an alternative.

```java
            @Param("until") LocalDateTime until);

}
```

Spring Boot will automatically find these repositories, and Spring Data will do its magic to provide an implementation behind the repository interface that will actually talk to the database.

Having JPA entities and repositories in place, we can implement the persistence adapter that provides the persistence functionality to our application:

```java
@RequiredArgsConstructor
@Component
class AccountPersistenceAdapter implements
    LoadAccountPort,
    UpdateAccountStatePort {

  private final AccountRepository accountRepository;
  private final ActivityRepository activityRepository;
  private final AccountMapper accountMapper;

  @Override
  public Account loadAccount(
          AccountId accountId,
          LocalDateTime baselineDate) {

    AccountJpaEntity account =
        accountRepository.findById(accountId.getValue())
            .orElseThrow(EntityNotFoundException::new);

    List<ActivityJpaEntity> activities =
        activityRepository.findByOwnerSince(
            accountId.getValue(),
            baselineDate);

      Long withdrawalBalance = activityRepository
          .getWithdrawalBalanceUntil(
                  accountId.getValue(),
                  baselineDate)
          .orElse(0L);

      Long depositBalance = activityRepository
          .getDepositBalanceUntil(
                  accountId.getValue(),
```

```java
                    baselineDate)
            .orElse(0L);

      return accountMapper.mapToDomainEntity(
          account,
          activities,
          withdrawalBalance,
          depositBalance);

    }

    @Override
    public void updateActivities(Account account) {
      for (Activity activity : account.getActivityWindow().getActivities()) {
        if (activity.getId() == null) {
          activityRepository.save(accountMapper.mapToJpaEntity(activity));
        }
      }
    }

    private Long orZero(Long value){
      return value == null ? 0L : value;
    }
}
```

The persistence adapter implements two ports that are needed by the application, `LoadAccountPort` and `UpdateAccountStatePort`.

To load an account from the database, we load it from `AccountRepository` and then load the activities of this account for a certain time window through `ActivityRepository`.

To create a valid `Account` domain entity, we also need the balance the account had before the start of this activity window, so we get the sum of all withdrawals and deposits of this account from the database.

Finally, we map all this data to an `Account` domain entity and return it to the caller.

To update the state of an account, we iterate over all activities of the `Account` entity and check whether they have IDs. If they don't, they are new activities, which we then persist through the `ActivityRepository`.

In the scenario described previously, we have a two-way mapping between the `Account` and `Activity` domain models and the `AccountJpaEntity` and `ActivityJpaEntity` database models. Why do we make the effort to map back and forth? Couldn't we just move the JPA annotations to the `Account` and `Activity` classes and directly store them as entities in the database?

Such a “no-mapping” strategy may be a valid choice, as we’ll see in Chapter 9, Mapping between Boundaries, when we talk about mapping strategies. However, JPA then forces us to make compromises in the domain model. For instance, JPA requires entities to have a no-args constructor. Alternatively, it might be that in the persistence layer, a “many-to-one” relationship makes sense from a performance point of view, but in the domain model, we want this relationship to be the other way around.

So, if we want to create a rich domain model without making compromises to the persistence layer, we’ll have to map between the domain model and the persistence model.

What about database transactions?

We have not touched on the topic of database transactions, yet. Where do we put our transaction boundaries?

A transaction should span all write operations to the database that are performed within a certain use case, ensuring that all those operations can be rolled back together if one of them fails.

Since the persistence adapter doesn’t know which other database operations are part of the same use case, it cannot decide when to open and close a transaction. We have to delegate this responsibility to the services that orchestrate the calls to the persistence adapter.

The easiest way to do this with Java and Spring is to add the `@Transactional` annotation to the domain service classes so that Spring will wrap all public methods with a transaction:

```java
package buckpal.application.domain.service;

@Transactional
public class SendMoneyService implements SendMoneyUseCase {
  ...
}
```

But doesn’t the `@Transactional` annotation introduce a dependency on a framework that we don’t want to have in our precious domain code? Well, yes, we have a dependency on the annotation, but we get transaction handling for that dependency! We wouldn’t want to build our own transaction mechanism just for the code to stay “pure”.

How does this help me build maintainable software?

Building a persistence adapter that acts as a plugin to the domain code frees the domain code from persistence details so that we can build a rich domain model.

Using narrow port interfaces, we're flexible to implement one port in one way and another port in another way, perhaps even with a different persistence technology, without the application noticing. We can even switch out the complete persistence layer, as long as the port contracts are obeyed.[30]

Now that we've built a domain model and some adapters, let's take a look at how we can test that they're really doing what we expect them to do.

[30]While I have seen it happen a few times (and for good reasons), the probability of having to switch out the whole persistence layer is usually rather low. Even then, having dedicated persistence ports is still worthwhile, because it increases testability. We can easily implement an in-memory persistence adapter to be used in tests, for example.

8. Testing Architecture Elements

In many projects I've witnessed, especially the projects that have been around for a while and have rotated in and out many developers over time, automated testing is a mystery. Everyone writes tests as they see fit, because it's required by some dusty rule documented in a wiki, but no one can answer targeted questions about a team's testing strategy.

This chapter provides a testing strategy for a Hexagonal Architecture. For each element of our architecture, we'll discuss the type of test to cover it.

The test pyramid

Let's start the discussion about testing along the lines of the test pyramid[31] in Figure 8.1, which is a metaphor that helps us to decide on how many tests of which type we should aim for.

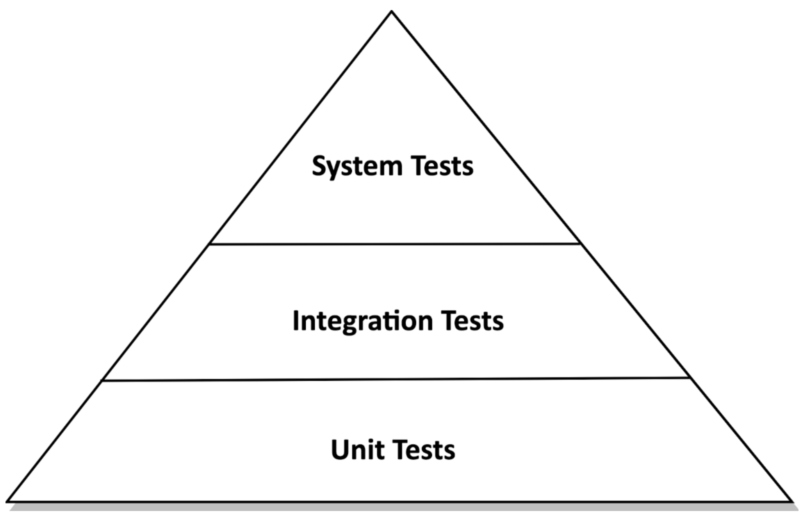

Figure 8.1 - According to the test pyramid, we should create many cheap tests and fewer expensive ones.

The basic statement of the pyramid is that we should have high coverage of fine-grained tests that are cheap to build, easy to maintain, fast-running, and stable. These are unit tests that verify that a single "unit" (usually a class) works as expected.

Once tests combine multiple units and go across unit boundaries, architectural boundaries, or even system boundaries, they tend to become more expensive to build, slower to run and more brittle (failing due to some configuration error instead of a functional error). The pyramid tells us that the more expensive those tests become, the less we should aim for high coverage of these tests, because otherwise, we'll spend too much time building tests instead of new functionality.

[31]The test pyramid can be traced back to Mike Cohn's book *Succeeding with Agile* from 2009.

Depending on the context, the test pyramid is often shown with different layers. Let's take a look at the layers I chose to discuss testing our Hexagonal Architecture. Note that the definition of "unit test", "integration test", and "system test" varies with context. In one project they may mean a different thing than in another. The following are interpretations of these terms as we'll use them in this chapter.

- Unit tests are the base of the pyramid. A unit test usually instantiates a single class and tests its functionality through its interface. If the class under test has non-trivial dependencies on other classes, we can replace those dependencies with mock objects that simulate the behavior of the real objects as required by the test.
- Integration tests form the next layer of the pyramid. These tests instantiate a network of multiple units and verify whether this network works as expected, by sending some data into it through the interface of an entry class. In our interpretation, integration tests will cross the boundary between two layers, so the network of objects is not complete or must work against mocks at some point.
- System tests, finally, spin up the whole network of objects that make up our application and verify if a certain use case works as expected through all the layers of the application.

Above the system tests, there might be a layer of end-to-end tests that include the UI of the application. We'll not consider end-to-end tests here, since we're only discussing a backend architecture in this book.

Note that the test pyramid, like any other guidance, is not a silver bullet for your test strategy. It's a good default, but if, in your context, you can create and maintain integration or system tests cheaply, you can and should create more of those tests, as they are less vulnerable to changes in implementation details than unit tests. This would make the sides of the pyramid steeper, or maybe even invert them.

Now that we have defined some test types, let's see which type of test fits best to each of the layers of our Hexagonal Architecture.

Testing a domain entity with unit tests

We will start by looking at a domain entity at the center of our architecture. Let's recall the `Account` entity from Chapter 5, Implementing a Use Case. The state of `Account` consists of a balance an account had at a certain point in the past (the baseline balance) and a list of deposits and withdrawals (activities) since then.

We now want to verify that the `withdraw()` method works as expected:

```java
class AccountTest {

  @Test
  void withdrawalSucceeds() {
    AccountId accountId = new AccountId(1L);
    Account account = defaultAccount()
        .withAccountId(accountId)
        .withBaselineBalance(Money.of(555L))
        .withActivityWindow(new ActivityWindow(
            defaultActivity()
                .withTargetAccount(accountId)
                .withMoney(Money.of(999L)).build(),
            defaultActivity()
                .withTargetAccount(accountId)
                .withMoney(Money.of(1L)).build()))
        .build();

    AccountId randomTargetAccount = new AccountId(99L);
    boolean success = account.withdraw(Money.of(555L), randomTargetAccount);

    assertThat(success).isTrue();
    assertThat(account
            .getActivityWindow()
            .getActivities())
            .hasSize(3);
    assertThat(account.calculateBalance())
            .isEqualTo(Money.of(1000L));
  }
}
```

The preceding test is a plain unit test that instantiates an `Account` in a specific state, calls its `withdraw()` method, and verifies that the withdrawal was successful and had the expected side effects on the state of the `Account` object under test.

The test is rather easy to set up, is easy to understand, and runs very fast. Tests don't come much simpler than this. Unit tests such as these are our best bet to verify the business rules encoded within our domain entities. We don't need any other type of test, since domain entity behavior has little to no dependencies on other classes.

Testing a use case with unit tests

Going a layer outward, the next architecture element to test is the use cases implemented as domain services. Let's look at a test of `SendMoneyService`, discussed in Chapter 5, Implementing a Use Case.

The “Send Money” use case withdraws money from the source account and deposits it into the target account. We want to verify that everything works as expected when the transaction succeeds:

```
class SendMoneyServiceTest {

  // declaration of fields omitted

  @Test
  void transactionSucceeds() {

    // given
    Account sourceAccount = givenSourceAccount();
    Account targetAccount = givenTargetAccount();

    givenWithdrawalWillSucceed(sourceAccount);
    givenDepositWillSucceed(targetAccount);

    Money money = Money.of(500L);

    SendMoneyCommand command = new SendMoneyCommand(
        sourceAccount.getId(),
        targetAccount.getId(),
        money);

    // when
    boolean success = sendMoneyService.sendMoney(command);

    // then
    assertThat(success).isTrue();

    AccountId sourceAccountId = sourceAccount.getId();
    AccountId targetAccountId = targetAccount.getId();

    then(sourceAccount).should().withdraw(eq(money), eq(targetAccountId));
    then(targetAccount).should().deposit(eq(money), eq(sourceAccountId));
    thenAccountsHaveBeenUpdated(sourceAccountId, targetAccountId);
  }

  // helper methods omitted
}
```

To make the test a little more readable, it’s structured into given/when/then sections that are commonly used in Behavior-Driven Development.

In the "given" section, we create the source and target `Account` objects and put them into the correct state with some methods whose names start with `given...()`. We also create a `SendMoneyCommand` object to act as input to the use case. In the "when" section, we simply call the `sendMoney()` method to invoke the use case. The "then" section asserts that the transaction was successful and verifies that certain methods have been called on the source and target `Account` objects.

Under the hood, the test makes use of the Mockito library to create mock objects in the `given...()` methods.[32] Mockito also provides the `then()` method to verify whether a certain method has been called on a mock object.

Note that, if used too much, mocking can give a false sense of security. Mocks may behave differently from the real thing, causing issues in production even though our tests are green. If you can use real objects instead of mocks without too much extra effort, you should probably do it. In the preceding example, we might choose to work with real `Account` objects instead of mocks, for example. This shouldn't prove much more effort, because the `Account` class is a domain model class that doesn't have any complicated dependencies on other classes.

Since the use case service under test is stateless, we cannot verify a certain state in the "then" section. Instead, the test verifies that the service has interacted with certain methods on its (mocked) dependencies. This means that the test is vulnerable to changes in the *structure* of the code under test and not only its *behavior*. This, in turn, means that there is a higher chance that the test has to be modified if the code under test is refactored.

With this in mind, we should think hard about which interactions we actually want to verify in the test. It might be a good idea not to verify *all* interactions as we did in the preceding test, and instead focus on the most important ones. Otherwise, we have to change the test with every single change to the class under test, undermining the value of the test.

While this test is still a unit test, it borders on being an integration test because we test the interaction on dependencies. However, it's easier to create and maintain than a full-blown integration test, however, because we're working with mocks and don't have to manage the real dependencies.

Testing a web adapter with integration tests

Moving outward another layer, we arrive at our adapters. Let's discuss testing a web adapter.

Recall that a web adapter takes input, for example in the form of JSON strings, via HTTP, might do some validation on it, maps the input to the format a use case expects and then passes it to that use case. It then maps the result of the use case back to JSON and returns it to the client via an HTTP response.

In the test for a web adapter, we want to make certain that all those steps work as expected:

[32] https://site.mockito.org/

```java
@WebMvcTest(controllers = SendMoneyController.class)
class SendMoneyControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private SendMoneyUseCase sendMoneyUseCase;

  private static final String ENDPOINT
    = "/accounts/sendMoney/{sourceAccountId}/{targetAccountId}/{amount}";

  @Test
  void testSendMoney() throws Exception {

    mockMvc.perform(
        post(ENDPOINT, 41L, 42L, 500)
          .header("Content-Type", "application/json"))
          .andExpect(status().isOk());

    then(sendMoneyUseCase).should()
        .sendMoney(eq(new SendMoneyCommand(
            new AccountId(41L),
            new AccountId(42L),
            Money.of(500L))));
  }
}
```

The preceding test is a standard integration test for a web controller named `SendMoneyController`, built with the Spring Boot framework. In the `testSendMoney()` method, we send a mock HTTP request to the web controller to trigger a transaction from one account to another.

With the `isOk()` method, we then verify that the status of the HTTP response is 200, and we verify that the mocked use case class has been called.

Most responsibilities of a web adapter are covered by this test.

We're not actually testing via the HTTP protocol, since we're mocking that away with the `MockMvc` object. We trust that the framework translates everything to and from HTTP properly. There is no need to test the framework.

However, the whole path from mapping the input from JSON into a `SendMoneyCommand` object is covered. If we build the `SendMoneyCommand` object as a self-validating command, as explained in Chapter 5, Implementing a Use Case, we even make sure that this mapping produces syntactically valid input to the use case. Also, we have verified that the use case is actually called, and that the

HTTP response has the expected status.

So, why is this an integration test and not a unit test? Even though it seems that we only test a single web controller class in this test, there's a lot more going on under the hood. With the `@WebMvcTest` annotation we tell Spring to instantiate a whole network of objects that is responsible for responding to certain request paths, mapping between Java and JSON, validating HTTP input, and so on. And in this test, we verify that our web controller works as a part of this network.

Since the web controller is heavily coupled to the Spring framework, it makes sense to test it when integrated into this framework instead of testing it in isolation. If we tested the web controller with a plain unit test, we'd lose coverage of all the mapping, validation, and HTTP stuff, and we could never be sure whether it actually worked in production, where it's just a cog in the mechanics of the framework.

Testing a persistence adapter with integration tests

For a similar reason it makes sense to cover persistence adapters with integration tests instead of unit tests, since we not only want to verify the logic within the adapter, but also the mapping into the database.

We want to test the persistence adapter we built in Chapter 7, Implementing a Persistence Adapter. The adapter has two methods, one to load an `Account` entity from the database and another to save new account activities to the database:

```
@DataJpaTest
@Import({AccountPersistenceAdapter.class, AccountMapper.class})
class AccountPersistenceAdapterTest {

  @Autowired
  private AccountPersistenceAdapter adapter;

  @Autowired
  private ActivityRepository activityRepository;

  @Test
  @Sql("AccountPersistenceAdapterTest.sql")
  void loadsAccount() {
    Account account = adapter.loadAccount(
        new AccountId(1L),
        LocalDateTime.of(2018, 8, 10, 0, 0));

    assertThat(account.getActivityWindow()
            .getActivities())
```

```
            .hasSize(2);
    assertThat(account.calculateBalance())
            .isEqualTo(Money.of(500));
  }

  @Test
  void updatesActivities() {
    Account account = defaultAccount()
        .withBaselineBalance(Money.of(555L))
        .withActivityWindow(new ActivityWindow(
            defaultActivity()
                .withId(null)
                .withMoney(Money.of(1L)).build()))
        .build();

    adapter.updateActivities(account);

    assertThat(activityRepository.count()).isEqualTo(1);

    ActivityJpaEntity savedActivity = activityRepository
            .findAll()
            .get(0);
    assertThat(savedActivity.getAmount()).isEqualTo(1L);
  }
}
```

With `@DataJpaTest`, we tell Spring to instantiate the network of objects that are needed for database access, including our Spring Data repositories that connect to the database. We add some additional `@Import`s to make sure that certain objects are added to that network. These objects are needed by the adapter under test to map incoming domain objects into database objects, for instance.

In the test for the `loadAccount()` method, we put the database into a certain state using an SQL script with the name `AccountPersistenceAdapterTest.sql`. Then, we simply load the account through the adapter API and verify that it has the state that we would expect it to have given the database state in the SQL script.

The test for `updateActivities()` goes the other way around. We create an `Account` object with a new account activity and pass it to the adapter to persist. Then, we check whether the activity has been saved to the database through the API of `ActivityRepository`.

An important aspect of these tests is that we're not mocking away the database. The tests actually hit the database. Had we mocked the database away, the tests would still cover the same lines of code, producing the same high coverage of lines of code. However, despite this high coverage the tests would still have a rather high chance of failing in a setup with a real database due to errors in SQL statements or unexpected mapping errors between database tables and Java objects.

Note that, by default, Spring will spin up an in-memory database to use during tests. This is very practical, as we don't have to configure anything, and the tests will work out of the box. However, since this in-memory database is most probably not the database we use in production, there is still a significant chance of something going wrong with the real database even when the tests work perfectly against the in-memory database. Database vendors love to implement their own flavor of SQL, for instance.

For this reason, persistence adapter tests should run against the real database. Libraries such as Testcontainers are a great help in this regard, spinning up a Docker container with a database on demand.[33]

Running against the real database has the added benefit that we don't have to take care of two different database systems. If we use the in-memory database during tests, we might have to configure it in a certain way, or we might have to create separate versions of database migration scripts for each database, which is a big hit on the maintainability of our tests.

Testing main paths with system tests

At the top of the pyramid are what I call "system tests". A system test starts up the whole application and runs requests against its API, verifying that all our layers work in concert.

Hexagonal Architecture is all about creating a well-defined boundary between our application and the outside world. By doing so, it makes our application boundaries very testable by design. To test our application locally, we just need to swap out the adapters with mock adapters, as outlined in Figure 8.2.

[33] https://www.testcontainers.org/

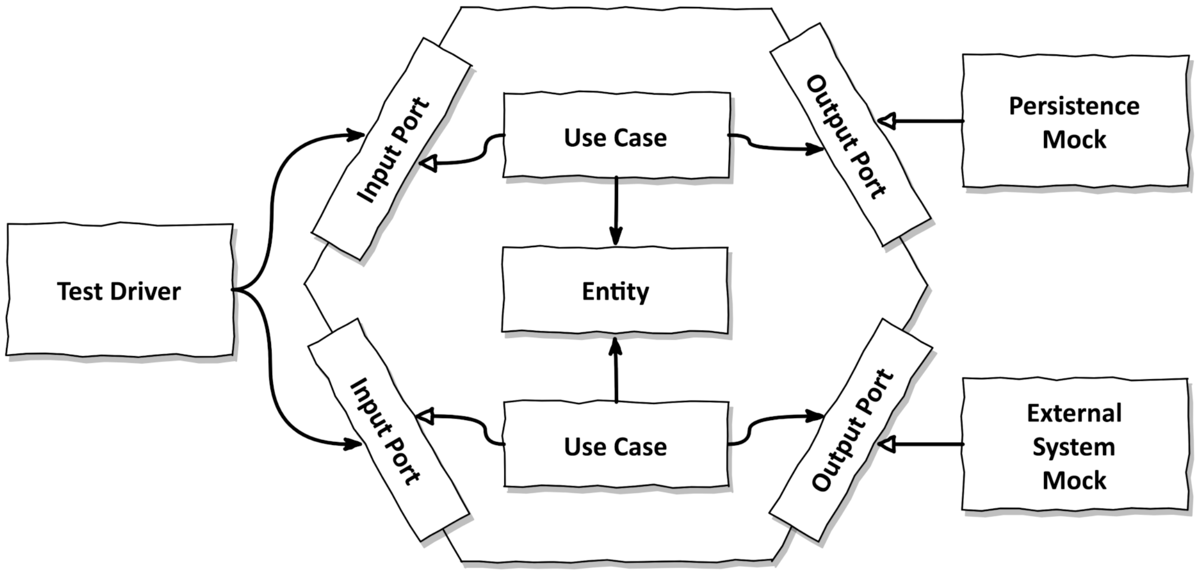

Figure 8.2 - By replacing the adapters with mocks, we can run and test our application without dependencies on the outside world.

On the left, we can replace the input adapters with a test driver that calls the application's input ports to interact with it. The test driver can implement certain test scenarios that simulate user behavior during an automated test.

On the right, we can replace the output adapters with mock adapters that simulate the behavior of a real adapter and return previously specified values.[34]

This way, we could create "application tests" that cover the "hexagon" of our application from input ports, through our domain services and entities, to the output ports.

I would argue, however, that, instead of writing "application tests" that mock away the input and output adapters, we should aim to write "system tests" that cover the whole path from a real input adapter to a real output adapter. These tests uncover many subtle bugs that we wouldn't catch if we mocked away the input and output adapters. These bugs include mapping errors between the layers or simply wrong expectations between the application and the outside systems it's talking to.

A "system test" such as this requires that we can spin up the real external systems our application talks to in a test setup.

On the input side, we need to make sure that we can make real HTTP calls to our application, for example, so that the requests go through our real web adapter. That should be rather easy, however, since we just need to start our application locally and let it listen to HTTP calls like it would in a production environment.

On the output side, we need to spin up a real database, for example, so that our tests go through the

[34]Depending on who you ask and what you're doing in your test, instead of calling it a "mock", you should call it a "fake" or "stub". Each term seems to have a slightly different semantic, but in the end they all replace a "real" thing with a "mock" thing to be used in tests. I'm usually a fan of naming things just right, but in this case, I don't see value in discussing the nuances between where a mock ends and a stub starts. Or is it the other way around?

real persistence adapter. Most databases make that easy today by providing a Docker image that we can spin up locally. If our application talks to a third-party system that is not a database, we should still try to find (or create) a Docker image that contains that system, so we can test our application against it by spinning up a local Docker container.

If no Docker image is available for a given external system, we can write a custom mock output adapter that simulates the real thing. Hexagonal Architecture makes it easy for us to replace the real output adapter with this mock for the purpose of our tests. And if a Docker image becomes available, we can switch to the real output adapter without too much effort.

There are valid reasons to test against mock adapters instead of real adapters, of course. If our application runs in multiple profiles, for example, and each profile uses a different (real) input or output adapter implemented against the same input and output ports, we might want to have tests that isolate errors in the application from errors in the adapters. "Application tests" that cover only our "hexagon" are exactly the tool we want, then. However, for the "standard" web application with a database, where the input and output adapters are rather static, we probably want to focus on system tests instead.

What would a system test look like? In a system test for the "Send Money" use case, we send an HTTP request to the application and validate the response as well as the new balance of the account. In the Java and Spring world, this is what it might look like:

```java
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class SendMoneySystemTest {

  @Autowired
  private TestRestTemplate restTemplate;

  private static final String ENDPOINT
    = "/accounts/sendMoney/{sourceAccountId}/{targetAccountId}/{amount}";

  @Test
  @Sql("SendMoneySystemTest.sql")
  void sendMoney() {

    Money initialSourceBalance = sourceAccount().calculateBalance();
    Money initialTargetBalance = targetAccount().calculateBalance();

    ResponseEntity response = whenSendMoney(
        sourceAccountId(),
        targetAccountId(),
        transferredAmount());

    then(response.getStatusCode())
        .isEqualTo(HttpStatus.OK);
```

```

    then(sourceAccount().calculateBalance())
        .isEqualTo(initialSourceBalance.minus(transferredAmount()));

    then(targetAccount().calculateBalance())
        .isEqualTo(initialTargetBalance.plus(transferredAmount()));
  }

  private ResponseEntity whenSendMoney(
      AccountId sourceAccountId,
      AccountId targetAccountId,
      Money amount) {
    HttpHeaders headers = new HttpHeaders();
    headers.add("Content-Type", "application/json");
    HttpEntity<Void> request = new HttpEntity<>(null, headers);

    return restTemplate.exchange(
        ENDPOINT,
        HttpMethod.POST,
        request,
        Object.class,
        sourceAccountId.getValue(),
        targetAccountId.getValue(),
        amount.getAmount());
  }

    // some helper methods omitted
}
```

With `@SpringBootTest`, we tell Spring to start up the whole network of objects that make up the application. We also configure the application to expose itself on a random port.

In the test method, we simply create a request, send it to the application, and then check the response status and the new balance of the accounts.

We use a `TestRestTemplate` to send the request, and not `MockMvc`, as we did earlier in the web adapter test. This means that the tests makes real HTTP calls, bringing the test a little closer to a production environment.

Just like we go over real HTTP, we go through the real output adapters. In our case, this is only a persistence adapter that connects the application to a database. In an application that talks to other systems, we would have additional output adapters in place. It's not always feasible to have all those third-party systems up-and-running, even for a system test, so we might mock them away, after all. Our Hexagonal Architecture makes this as easy as it can be for us, since we only have to stub out a couple of output port interfaces.

Note that I went out of my way to make the test as readable as possible. I hid every bit of ugly logic within helper methods. These methods now form a domain-specific language that we can use to verify the state of things.

While a domain-specific language such as this is a good idea in any type of test, it's even more important in system tests. System tests simulate the real users of the application much better than unit or integration tests can, so we can use them to verify the application from the viewpoint of the user. This is much easier with a suitable vocabulary at hand. This vocabulary also enables domain experts, who are best suited to embody a user of the application and probably aren't programmers, to reason about the tests and give feedback. There are whole libraries for behavior-driven development like JGiven that provide a framework to create a vocabulary for your tests.[35]

If we created unit and integration tests as described in the previous sections, the system tests will cover a lot of the same code. Do they even provide any additional benefits? Yes, they do. Usually, they flush out other types of bugs than the unit and integration tests do. Some mapping between the layers could be off, for instance, which we would not notice with the unit and integration tests alone.

System tests play out their strength best if they combine multiple use cases to create scenarios. Each scenario represents a certain path a user might typically take through the application. If the most important scenarios are covered by passing system tests, we can assume that we haven't broken them with our latest modifications and are ready to ship.

How much testing is enough?

A question many project teams I've been part of couldn't answer is how much testing we should do. Is it enough if our tests cover 80% of our lines of code? Should it be higher than that?

Line coverage is a bad metric to measure test success. Any goal other than 100% is completely meaningless because important parts of the codebase might not be covered at all[36]. And even at 100%, we still can't be sure that every bug has been squashed.

I suggest measuring test success by how comfortable we feel shipping the software. If we trust the tests enough to ship after having executed them, we're good. The more often we ship, the more trust we have in our tests. If we only ship twice a year, no one will trust the tests because they only prove themselves twice a year.

This requires a leap of faith the first couple of times we ship, but if we make it a priority to fix *and learn from* bugs in production, we're on the right track. For each production bug, we should ask the question "Why didn't our tests catch this bug?", document the answer, and then add a test that covers it. Over time this will make us comfortable with shipping, and the documentation will even provide a metric to gauge our improvement over time.

[35]https://jgiven.org/

[36]If you want to read more about 100% test coverage, have a look at my article with the tongue-in-cheek title "Why you should enforce 100% code coverage" at https://reflectoring.io/100-percent-test-coverage/.

It helps, however, to start with a strategy that defines the tests we should create. One such strategy for our Hexagonal Architecture is this one:

- while implementing a domain entity, cover it with a unit test
- while implementing a use case service, cover it with a unit test
- while implementing an adapter, cover it with an integration test
- cover the most important paths a user can take through the application with a system test.

Note the words "while implementing": when tests are done *during* the development of a feature and not *after*, they become a development tool and no longer feel like a chore.

However, if we have to spend an hour fixing tests each time we add a new field, however, we're doing something wrong. Probably, our tests are too vulnerable to structural changes in the code, and we should look at how to improve that. Tests lose their value if we have to modify them for each refactoring.

How does this help me build maintainable software?

The Hexagonal Architecture style cleanly separates domain logic and outward-facing adapters. This helps us to define a clear testing strategy that covers the central domain logic with unit tests and the adapters with integration tests.

The input and output ports provide very visible mocking points in tests. For each port, we can decide to mock it or use the real implementation. If the ports are each very small and focused, mocking them is a breeze instead of a chore. The fewer methods a port interface provides, the less confusion there is about which of the methods we have to mock in a test.

If it becomes too much of a burden to mock things away, or if we don't know which kind of test we should use to cover a certain part of the codebase, it's a warning sign. In this regard, our tests have the additional responsibility of a canary - to warn us about flaws in the architecture and steer us back on the path to creating a maintainable codebase.

So far, we have talked about our use cases and our adapters mostly in isolation, so far. How do they communicate with each other? In the next chapter we'll take a look at some strategies on how to design data models that make up the common language between them.

9. Mapping between Boundaries

In the previous chapters, we've discussed the web, application, domain, and persistence layers and what each of those layers contributes to implementing a use case.

We have, however, barely touched on the dreaded and omnipresent topic of mapping between the models of each layer. I bet you've had a discussion at some point about whether to use the same model in two layers in order to avoid implementing a mapper.

The argument might have gone something like this:

Pro-mapping developer:
> *If we don't map between layers, we have to use the same model in both layers which means that the layers will be tightly coupled!*

Contra-mapping developer:
> *But if we **do** map between layers, we produce a lot of boilerplate code which is overkill for many use cases, since they're only doing CRUD and have the same model across layers anyways!*

As is often the case in discussions like this, there's truth to both sides of the argument. Let's discuss some mapping strategies with their pros and cons and see whether we can help those developers make a decision.

The "No Mapping" strategy

The first strategy is actually not mapping at all.

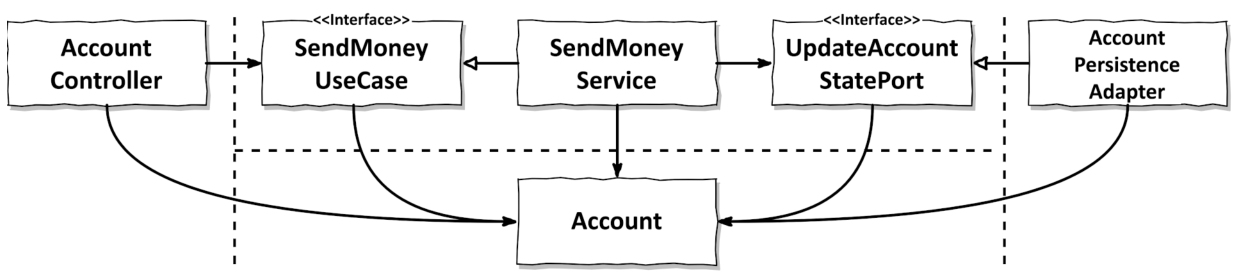

Figure 9.1 - If the port interfaces use the domain model as input and output model we can choose not to map between layers.

Figure 9.1 shows the components that are relevant for the "Send Money" use case from our BuckPal example application.

In the web layer, the web controller calls the `SendMoneyUseCase` interface to execute the use case. This interface takes an `Account` object as an argument. This means that both the web and application layers need access to the `Account` class - both are using the same model.

On the other side of the application, we have the same relationship between the persistence and application layer.

Since all layers use the same model, we don't need to implement a mapping between them.

But what are the consequences of this design?

The web and persistence layers may have special requirements for their models. If our web layer exposes its model via REST, for instance, the model classes might need some annotations that define how to serialize certain fields into JSON. The same is true for the persistence layer if we're using an object-relational mapping (ORM) framework, which might require some annotations that define the database mapping. The framework might also require the class to follow a certain contract.

In the example, all of those special requirements have to be dealt with in the `Account` domain model class, even though the domain and application layers are not interested in them. This violates the Single Responsibility Principle since the `Account` class has to be changed for reasons related to the web, application, *and* persistence layers.

Aside from the technical requirements, each layer might require certain custom fields on the `Account` class. This might lead to a fragmented domain model with certain fields only relevant in one layer.

Does this mean, though, that we should never, ever implement a "no mapping" strategy? Certainly not.

Even though it might feel dirty, a "no mapping" strategy can be perfectly valid.

Consider a simple CRUD use case. Do we really need to map the same fields from the web model into the domain model and from the domain model into the persistence model? I'd say we don't.

And what about those JSON or ORM annotations on the domain model? Do they really bother us? Even if we have to change an annotation or two in the domain model if something changes in the persistence layer, so what?

As long as all layers need exactly the same information in exactly the same structure, a "no mapping" strategy is a perfectly valid option.

As soon as we're dealing with web or persistence issues in the application or domain layer (aside from annotations, perhaps), however, we should move to another mapping strategy.

There is a lesson for the two developers from the introduction here: even though we have decided on a certain mapping strategy in the past, we can change it later.

In my experience, many use cases start their life as simple CRUD use cases. Later, they might grow into a full-fledged business use case with rich behavior and validations that justify a more expensive mapping strategy. Or they might forever keep their CRUD status, in which case we're glad that we haven't invested in a different mapping strategy.

The “Two-Way” mapping strategy

A mapping strategy where each layer has its own model is what I call the “Two-Way” mapping strategy, as outlined in Figure 9.2.

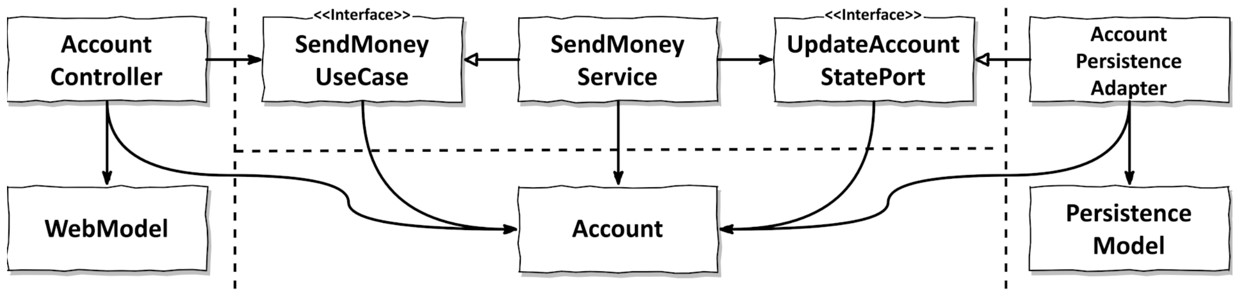

Figure 9.2 - With each adapter having its own model, the adapters are responsible for mapping their model into the domain model and back.

Each layer has its own model, which may have a structure that is completely different from the domain model.

The web layer maps the web model into the input model that is expected by the incoming ports. It also maps domain objects returned by the incoming ports back into the web model.

The persistence layer is responsible for a similar mapping between the domain model, which is used by the outgoing ports, and the persistence model.

Both layers map in two directions, hence the name “Two-Way” mapping.

With each layer having its own model, it can modify its own model without affecting the other layers (as long as the contents are unchanged). The web model can have a structure that allows for optimal presentation of the data. The domain model can have a structure that best allows for implementing the use cases. And the persistence model can have the structure needed by an OR-mapper for persisting objects to a database.

This mapping strategy also leads to a clean domain model that is not dirtied by web or persistence concerns. It does not contain JSON or ORM mapping annotations. The Single Responsibility Principle is satisfied.

Another bonus of “Two-Way” mapping is that, after the “No Mapping” strategy, it’s conceptually the simplest mapping strategy. The mapping responsibilities are clear: the outer layers / adapters map into the model of the inner layers and back. The inner layers only know their own model and can concentrate on the domain logic instead of mapping.

As with every mapping strategy, the “Two-Way” mapping also has its drawbacks.

First of all, it usually ends up in a lot of boilerplate code. Even if we use one of the many mapping frameworks out there to reduce the amount of code, implementing the mapping between models usually takes up a good portion of our time. This is partly due to the fact that debugging mapping logic is a pain - especially when using a mapping framework that hides its inner workings behind a layer of generic code and reflection.

Another potential drawback is that the incoming and outgoing ports use domain objects as input parameters and return values. The adapters map these into their own model, but this still creates more coupling between the layers than if we introduce a dedicated "transport model" as in the "full" mapping strategy we're going to discuss next.

Just like the "No Mapping" strategy, the "Two-Way" mapping strategy is not a silver bullet. In many projects, however, this kind of mapping is considered a holy law that we have to comply with throughout the whole codebase, even for the simplest CRUD use cases. This unnecessarily slows down development.

No single mapping strategy should be considered an iron law. Instead, we should decide for each use case.

The "Full" mapping strategy

Another mapping strategy is what I call the "Full" mapping strategy, as outlined in Figure 9.3.

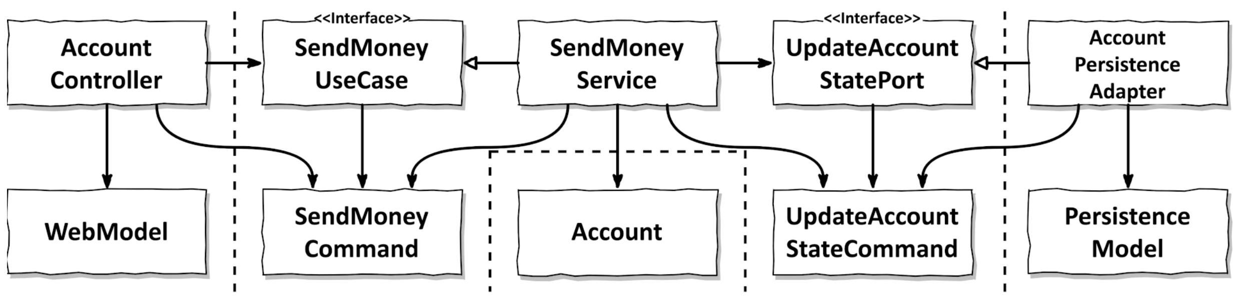

Figure 9.3 - With each operation requiring its own model, the web adapter and application layer each map their model into the model expected by the operation they want to execute.

This mapping strategy introduces a separate input and output model *per operation*. Instead of using the domain model to communicate across layer boundaries, we use a model specific to each operation, such as the `SendMoneyCommand`, which acts as an input model to the `SendMoneyUseCase` port in the figure. We can call those models "commands", "requests", or similar.

The web layer is responsible for mapping its input into the command object of the application layer. Such a command makes the interface to the application layer very explicit, with little room for interpretation. Each use case has its own command with its own fields and validations. There's no guessing involved as to which fields should be filled and which fields would be better left empty since they would otherwise trigger a validation we don't want for our current use case.

The application layer is then responsible for mapping the command object into whatever it needs to modify the domain model according to the use case.

Naturally, mapping from one layer into many different commands requires even more mapping code than mapping between a single web model and a domain model. This mapping, however, is significantly easier to implement and maintain than a mapping that has to handle the needs of many use cases instead of only one.

I don't advocate this mapping strategy as a global pattern. It plays out its advantages best between the web layer (or any other incoming adapter) and the application layer to clearly demarcate the state-modifying use cases of the application. I would not use it between the application and persistence layer due to the mapping overhead.

Usually, I would restrict this kind of mapping to the input model of operations and simply use a domain object as the output model. `SendMoneyUseCase` might then return an `Account` object with the updated balance, for instance.

This shows that the mapping strategies can and should be mixed. No single mapping strategy needs to be a global rule across all layers.

The "One-Way" mapping strategy

There is yet another mapping strategy with another set of pros and cons: the "One-Way" strategy visualized in Figure 9.4.

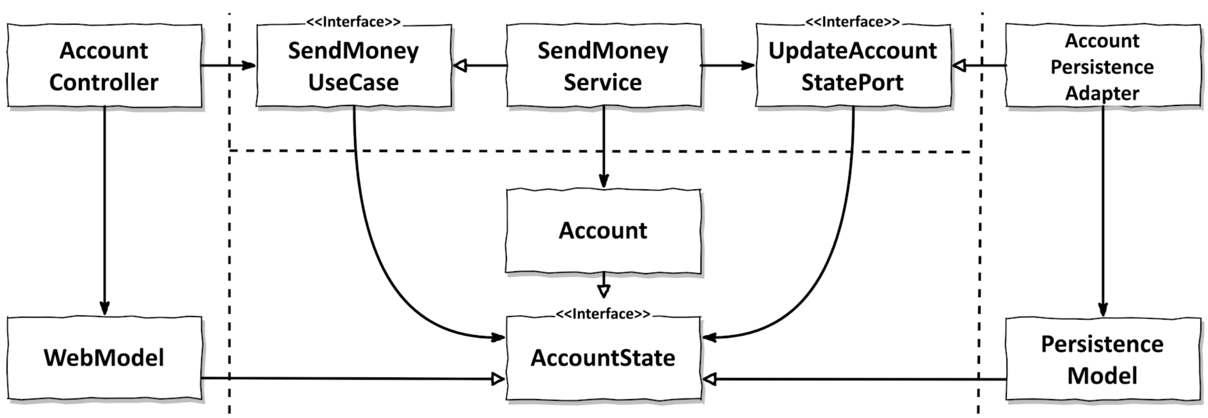

Figure 9.4 - With the domain model and the adapter models implementing the same "state" interface, each layer only needs to map objects it receives from other layers - one way.

In this strategy, the models in all layers implement the same interface that encapsulates the state of the domain model by providing getter methods on the relevant attributes.

The domain model itself can implement a rich behavior, which we can access from our services within the application layer. If we want to pass a domain object to the outer layers, we can do so without mapping, since the domain object implements the state interface expected by the incoming and outgoing ports.

The outer layers can then decide whether they can work with the interface or whether they need to map it into their own model. They cannot inadvertently modify the state of the domain object since the modifying behavior is not exposed by the state interface.

Objects we pass from an outer layer into the application layer also implement this state interface. The application layer then has to map it into the real domain model in order to get access to its behavior. This mapping plays well with the Domain-Driven Design (DDD) concept of a factory. A factory in terms of DDD is responsible for reconstituting a domain object from a certain state, which is exactly what we're doing.[37]

The mapping responsibility is clear: if a layer receives an object from another layer, we map it into something the layer can work with. Thus, each layer only maps one way, making this the "One-Way" mapping strategy.

With the mapping distributed across layers, however, this strategy is conceptually more difficult than the other strategies.

This strategy plays out its strength best if the models across the layers are similar. For read-only operations, for instance, the web layer then might not need to map into its own model at all, since the state interface provides all the information it needs.

When to use which mapping strategy?

This is the million-dollar question, isn't it?

The answer is the usual, dissatisfying, "it depends".

Since each mapping strategy has different advantages and disadvantages, we should resist the urge to define a single strategy as a hard-and-fast global rule for the whole codebase. This goes against our instincts, as it feels untidy to mix patterns within the same codebase. But knowingly choosing a pattern that is not the best pattern for a certain job, just to serve our sense of tidiness, is irresponsible, plain and simple.

Also, as software evolves over time, the strategy that was the best for the job yesterday might not still be the best for the job today. Instead of starting with a fixed mapping strategy and keeping it over time - no matter what - we might start with a simple strategy that allows us to quickly evolve the code and later move to a more complex one that helps us to better decouple the layers.

In order to decide which strategy to use when, we need to agree upon a set of guidelines within the team. These guidelines should answer the question of which mapping strategy should be the first choice in which situation. They should also answer *why* they are the first choice so that we're able to evaluate whether those reasons still apply after some time.

We might for example define different mapping guidelines to modifying use cases than we do to queries. Also, we might want to use different mapping strategies between the web and application layers and between the application and persistence layers.

Guidelines for these situations might look like this:

[37] *Domain Driven Design* by Eric Evans, Addison-Wesley, 2004, p. 158

- If we're working on a *modifying use case*, the "Full" mapping strategy is the first choice *between the web and application layer*, in order to decouple the use cases from one another. This gives us clear per-use-case validation rules, and we don't have to deal with fields we don't need in a certain use case.
- If we're working on a *modifying use case*, the "No Mapping" strategy is the first choice *between the application and persistence layer* in order to be able to quickly evolve the code without mapping overhead. As soon as we have to deal with persistence issues in the application layer, however, we move to a "Two-Way" mapping strategy to keep persistence issues in the persistence layer.
- If we're working on a *query*, the "No Mapping" strategy is the first choice *between the web and application layer* and *between the application and persistence layer* in order to be able to quickly evolve the code without mapping overhead. As soon as we have to deal with web or persistence issues in the application layer, however, we move to a "two-way" mapping strategy between the web and application layer or the application layer and persistence layer, respectively.

In order to successfully apply guidelines like these, they must be present in the minds of the developers. So, the guidelines should be discussed and revised continuously as a team effort.

How does this help me build maintainable software?

Incoming and outgoing ports act as gatekeepers between the layers of our application. They define how the layers communicate with each other, and how we map models across layers.

With narrow ports in place for each use case, we can choose different mapping strategies for different use cases, and even evolve them over time without affecting other use cases, thus selecting the best strategy for a certain situation at a certain time.

Selecting a different mapping strategy for each use case is harder and requires more communication than simply using the same mapping strategy for all situations, but it will reward the team with a codebase that does just what it needs to do and is easier to maintain, as long as the mapping guidelines are known.

Now that we know which components make up our application and how they communicate, we can explore how to assemble a working application out of the different components.

10. Assembling the Application

Now that we have implemented some use cases, web adapters, and persistence adapters, we need to assemble them into a working application. As discussed in Chapter 4, Organizing Code, we rely on a dependency injection mechanism to instantiate our classes and wire them together at startup time. In this chapter, we'll discuss some approaches to doing this with plain Java and the Spring and Spring Boot frameworks.

Why even care about assembly?

Why aren't we just instantiating the use cases and adapters when and where we need them? Because we want to keep the code dependencies pointed in the right direction. Remember: all dependencies should point inward, towards the domain code of our application, so that the domain code doesn't have to change when something in the outer layers changes.

If a use case needs to call a persistence adapter and just instantiates it itself, we have created a code dependency in the wrong direction.

This is why we created outgoing port interfaces. The use case only knows the interface and is provided an implementation of this interface at runtime.

A nice side effect of this programming style is that the code we're creating is much easier to test. If we can pass all objects a class needs into its constructor, we can choose to pass in mocks instead of the real objects, which makes it easy to create an isolated unit test for the class.

So, who's responsible for creating our object instances? And how do we do it without violating the Dependency Rule?

The answer is that there must be a configuration component that is neutral to our architecture and that has a dependency on *all* classes in order to instantiate them as shown in Figure 10.1.

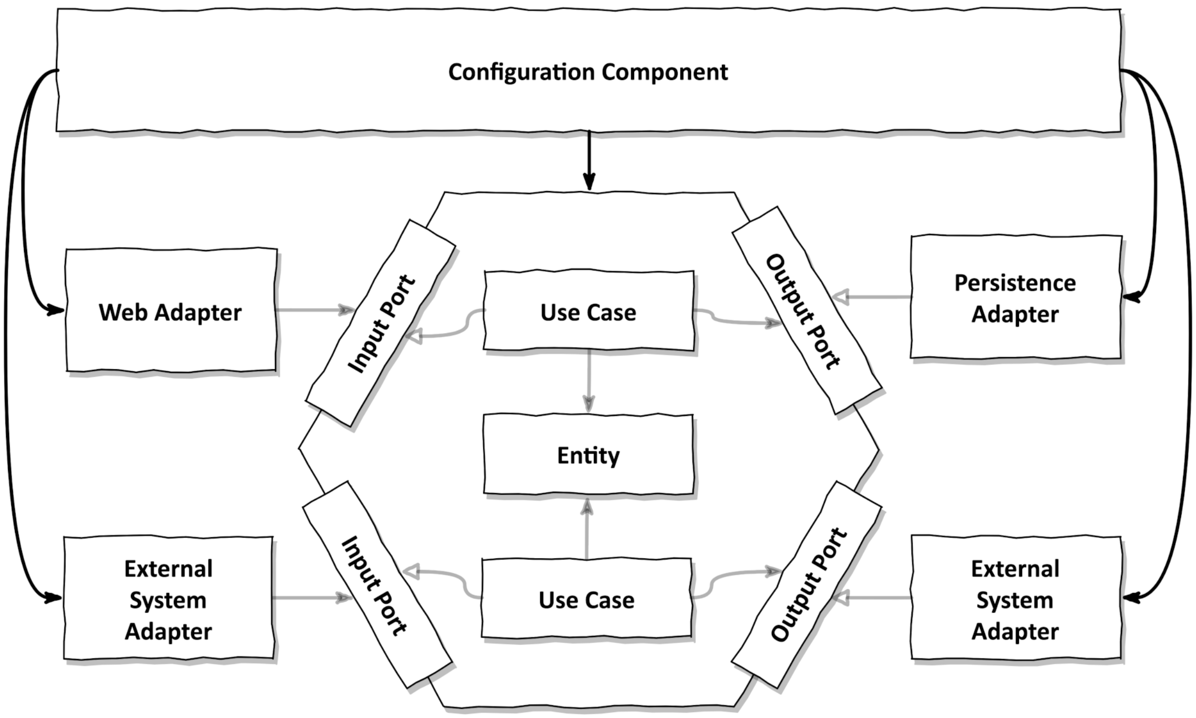

Figure 10.1 - A neutral configuration component may access all classes in order to instantiate them.

In the "Clean Architecture" introduced in Chapter 3, Inverting Dependencies, this configuration component would be in the outermost circle, which may access all inner layers, as defined by the Dependency Rule.

The configuration component is responsible for assembling a working application from the parts we provided. It must do the following:

- Create web adapter instances.
- Ensure that HTTP requests are actually routed to the web adapters.
- Create use case instances.
- Provide web adapters with use case instances.
- Create persistence adapter instances.
- Provide use cases with persistence adapter instances.
- Ensure that the persistence adapters can actually access the database.

Besides that, the configuration component should be able to access certain sources of configuration parameters, such as configuration files or command-line parameters. During application assembly, the configuration component then passes these parameters on to the application components to control behavior such as which database to access or which server to use to send emails.

These are a lot of responsibilities (read: "reasons to change" - remember the Single Responsibility Principle from Chapter 3, "Inverting Dependencies")! Aren't we violating the Single Responsibility

Principle here? Yes, we are, but if we want to keep the rest of the application clean, we need an outside component that takes care of the wiring. And this component has to know all the moving parts to assemble them into a working application.

Assembling via plain code

There are several ways to implement a configuration component responsible for assembling the application. If we're building an application without the support of a dependency injection framework, we can create such a component with plain code:

```java
package buckpal.configuration;

class Application {

  public static void main(String[] args) {
    AccountRepository accountRepository = new AccountRepository();
    ActivityRepository activityRepository = new ActivityRepository();
    AccountPersistenceAdapter accountPersistenceAdapter =
        new AccountPersistenceAdapter(accountRepository, activityRepository);

    SendMoneyUseCase sendMoneyUseCase =
        new SendMoneyUseService(
            accountPersistenceAdapter,  // LoadAccountPort
            accountPersistenceAdapter); // UpdateAccountStatePort

    SendMoneyController sendMoneyController =
        new SendMoneyController(sendMoneyUseCase);

    startProcessingWebRequests(sendMoneyController);
  }
}
```

This code snippet is a simplified example of how such a configuration component might look. In Java, an application is started from the `main` method. Within this method, we instantiate all the classes we need, from the web controller to the persistence adapter, and wire them together.

Finally, we call the mystic method `startProcessingWebRequests()` which exposes the web controller via HTTP.[38] The application is then ready to process requests.

This plain code approach is the most basic way of assembling an application. It has some drawbacks, however.

[38]This method is just a placeholder for any bootstrapping logic that is necessary to expose our web adapters via HTTP. We don't really want to implement this ourselves. In a real-world application, a framework does that for us.

- First of all, the preceding code is for an application that has only a single web controller, use case, and persistence adapter. Imagine how much code like this we would have to produce to bootstrap a full-blown enterprise application!
- Second, since we're instantiating all classes ourselves from outside of their packages, those classes all need to be public. This means, for example, that the Java compiler doesn't prevent a use case from directly accessing a persistence adapter, since it's public. It would be nice if we could avoid unwanted dependencies like this by using package-private visibility.

Luckily, there are dependency injection frameworks that can do the dirty work for us while still maintaining package-private dependencies. The Spring framework is currently the most popular one in the Java world. Spring also provides web and database support, among a lot of other things, so we don't have to implement the mystic `startProcessingWebRequests()` method after all.

Assembling via Spring's classpath scanning

If we use the Spring framework to assemble our application, the result is called the "application context". The application context contains all objects that together make up the application ("beans" in Java lingo).

Spring offers several approaches to assemble an application context, each having its own advantages and drawbacks. Let's start by discussing the most popular (and most convenient) approach: classpath scanning.

With classpath scanning, Spring goes through all classes that are available in a certain slice of the classpath and searches for classes that are annotated with the `@Component` annotation. The framework then creates an object from each of these classes. The classes should have a constructor that takes all required fields as an argument, like our `AccountPersistenceAdapter` from Chapter 7, Implementing a Persistence Adapter:

```
@Component
@RequiredArgsConstructor
class AccountPersistenceAdapter implements
    LoadAccountPort,
    UpdateAccountStatePort {

  private final AccountRepository accountRepository;
  private final ActivityRepository activityRepository;
  private final AccountMapper accountMapper;

  @Override
  public Account loadAccount(AccountId accountId, LocalDateTime baselineDate) {
    ...
  }
```

```

  @Override
  public void updateActivities(Account account) {
    ...
  }
}
```

In this case, we didn't even write the constructor ourselves, but instead let the Lombok library do it for us using the `@RequiredArgsConstructor` annotation, which creates a constructor that takes all `final` fields as arguments.

Spring will find this constructor and search for `@Component`-annotated classes of the required argument types and instantiate them in a similar manner to add them to the application context. Once all required objects are available, it will finally call the constructor of `AccountPersistenceAdapter` and add the resulting object to the application context as well.

Classpath scanning is a very convenient way of assembling an application. We only have to sprinkle some `@Component` annotations across the codebase and provide the right constructors.

We can also create our own stereotype annotation for Spring to pick up. We could, for example, create a `@PersistenceAdapter` annotation:

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface PersistenceAdapter {

  @AliasFor(annotation = Component.class)
  String value() default "";

}
```

This annotation is meta-annotated with `@Component` to let Spring know that it should be picked up during classpath scanning. We could now use `@PersistenceAdapter` instead of `@Component` to mark our persistence adapter classes as parts of our application. With this annotation we have made our architecture more evident to people reading the code.

The classpath scanning approach has its drawbacks, however. First, it's invasive in that it requires us to add a framework-specific annotation to our classes. If you're a Clean Architecture hardliner, you'd say that this is forbidden as it binds our code to a specific framework.

I'd say that in usual application development, a single annotation on a class is not such a big deal and can easily be refactored, if at all necessary.

In other contexts, however, like when building a library or a framework for other developers to use, this might be a no-go, since we don't want to encumber our users with a dependency on the Spring framework.

Another potential drawback of the classpath scanning approach is that magic things might happen. And by "magic" I mean the bad kind of magic causing inexplicable effects that might take days to figure out if you're not a Spring expert.

Magic happens because classpath scanning is a very blunt weapon to use for application assembly. We simply point Spring at the parent package of our application and tell it to go looking for `@Component`-annotated classes within this package.

Do you know every single class that exists within your application by heart? Probably not. There are bound to be some classes that we don't actually want to have in the application context. Perhaps this class even manipulates the application context in evil ways, causing errors that are hard to track.

Let's look at an alternative approach that gives us a little more control.

Assembling via Spring's Java Config

While classpath scanning is the cudgel of application assembly, Spring's Java Config is the scalpel.[39] This approach is similar to the plain code approach introduced earlier in this chapter, but it's less messy and provides us with a framework so that we don't have to code everything by hand.

In this approach, we create configuration classes, each responsible for constructing a set of beans that are to be added to the application context.

For example, we could create a configuration class that is responsible for instantiating all our persistence adapters:

```
@Configuration
@EnableJpaRepositories
class PersistenceAdapterConfiguration {

  @Bean
  AccountPersistenceAdapter accountPersistenceAdapter(
                AccountRepository accountRepository,
        ActivityRepository activityRepository,
        AccountMapper accountMapper){
    return new AccountPersistenceAdapter(
      accountRepository,
      activityRepository,
```

[39]If you don't spend far too many hours of your life killing monsters in role-playing video games like me and don't know what a cudgel is: a cudgel is a stick with a weighted end that can be used as a weapon. It's a very blunt weapon that can do a lot of damage without having to aim particularly well.

```
      accountMapper);
  }

  @Bean
  AccountMapper accountMapper(){
    return new AccountMapper();
  }
}
```

The `@Configuration` annotation marks this class as a configuration class to be picked up by Spring's classpath scanning. So, in this case, we're still using classpath scanning, but we only pick up our configuration classes instead of every single bean, which reduces the chance of evil magic happening.

The beans themselves are created within the `@Bean`-annotated factory methods of our configuration classes. In the preceding case, we add a persistence adapter to the application context. It needs two repositories and a mapper as input to its constructor. Spring automatically provides these objects as input to the factory methods.

But where does Spring get the repository objects from? If they are created manually in a factory method of another configuration class, then Spring would automatically provide them as parameters to the factory methods of the preceding code example. In this case, however, they are created by Spring itself, triggered by the `@EnableJpaRepositories` annotation. If Spring Boot finds this annotation, it will automatically provide implementations for all Spring Data repository interfaces we have defined.

If you're familiar with Spring Boot, you might know that we could have added the `@EnableJpaRepositories` annotation to the main application class instead of our custom configuration class. Yes, this is possible, but it would activate JPA repositories every time the application is started up, even if we start the application within a test that doesn't actually need persistence. So, by moving such "feature annotations" to a separate configuration "module" we've just become much more flexible and can start up parts of our application instead of always having to start the whole thing.

With the `PersistenceAdapterConfiguration` class, we have created a tightly-scoped persistence module that instantiates all objects we need in our persistence layer. It will be automatically picked up by Spring's classpath scanning while we still have full control over which beans are actually added to the application context.

Similarly, we could create configuration classes for web adapters, or for certain modules within our application layer. We can now create an application context that contains certain modules, but mocks the beans of other modules, which gives us great flexibility in tests. We could even push the code of each of those modules into its own codebase, package, or JAR file without much refactoring.

Also, this approach does not force us to sprinkle `@Component` annotations all over our codebase, like the classpath scanning approach does. So, we can keep our application layer clean without any dependency on the Spring framework (or any other framework, for that matter).

There is a catch with this solution, however. If the configuration class is not within the same package as the classes of the beans it creates (the persistence adapter classes in this case), those classes must be public. To restrict visibility, we can use packages as module boundaries and create a dedicated configuration class within each package. This way, we cannot use sub-packages, though, as will be discussed in Chapter 12, Enforcing Architecture Boundaries.

How does this help me build maintainable software?

Spring and Spring Boot (and similar frameworks) provide a lot of features that make our lives easier. One of the main features is assembling the application out of the parts (classes) that we, as application developers, provide.

Classpath scanning is a very convenient feature. We only have to point Spring to a package, and it assembles an application from the classes it finds. This allows for rapid development, with us not having to think about the application as a whole.

Once the codebase grows, however, this quickly leads to a lack of transparency. We don't know which beans exactly are loaded into the application context. Also, we cannot easily start up isolated parts of the application context to use in tests.

By creating a dedicated configuration component responsible for assembling our application, we can liberate our application code from this responsibility (read: "reason for change" - remember the "S" in "SOLID"?). We're rewarded with highly cohesive modules that we can start up in isolation from each other and that we can easily move around within our codebase. As usual, this comes at the price of spending some extra time to maintain this configuration component.

We've talked a lot about different options of how to do things the "right way" in this and the previous chapters. However, sometimes the "right way" is not feasible. In the next chapter, we'll talk about shortcuts, the price we pay for them, and when they're worth taking.

11. Taking Shortcuts Consciously

In the preface of this book, I cursed the fact that we feel forced to take shortcuts all the time, building up a great heap of technical debt we never have the chance to pay back.

To prevent shortcuts, we must be able to identify them. So, the goal of this chapter is to raise awareness of some potential shortcuts and discuss their effects.

With this information, we can identify and fix accidental shortcuts. Or, if justified, we can even consciously opt in to the effects of a shortcut.[40]

Why shortcuts are like broken windows

In 1969, psychologist Philip Zimbardo conducted an experiment to test a theory that later became known as the Broken Windows Theory.[41]

His team parked one car without license plates in a Bronx neighborhood and another in an allegedly "better" neighborhood in Palo Alto. Then, they waited.

The car in the Bronx was picked clean of valuable parts within 24 hours and then passersby started to randomly destroy it.

The car in Palo Alto was not touched for a week, so Zimbardo's team smashed a window. From then on, the car had a similar fate to the car in the Bronx and was destroyed in the same short amount of time by people walking by.

The people taking part in looting and destroying the cars came from across all social classes and included people who were otherwise law-abiding and well-behaved citizens.

This human behavior has become known as the Broken Windows Theory. In my own words:

> *As soon as something looks run-down, damaged, [insert negative adjective here], or generally untended, the human brain feels that it's OK to make it more run-down, damaged, or [insert negative adjective here].*

This theory applies to many areas of life:

- In a neighborhood where vandalism is common, the threshold to loot or damage an untended car is low.

[40]Imagine this sentence in a book about construction engineering or, even scarier, in a book about avionics! Most of us, however, are not building the software equivalent of a skyscraper or an airplane. And software is soft and can be changed more easily than hardware, so sometimes it's actually more economic to (consciously!) take a shortcut first and fix it later (or never).

[41]https://www.theatlantic.com/magazine/archive/1982/03/broken-windows/304465/

- When a car has a broken window, the threshold to damage it further is low, even in a "good" neighborhood.
- In an untidy bedroom, the threshold to throw our clothes on the ground instead of putting them into the wardrobe is low.
- In a classroom where students often disrupt the lesson, the threshold to crack another joke to classmates is low.
- …

Applied to working with code, this means the following:

- When working on a low-quality codebase, the threshold to add more low-quality code is low.
- When working on a codebase with a lot of coding violations, the threshold to add another coding violation is low.
- When working on a codebase with a lot of shortcuts, the threshold to add another shortcut is low.
- …

With all this in mind, is it really a surprise that the quality of many so-called "legacy" codebases has eroded so badly over time?

The responsibility of starting clean

While working with code doesn't really feel like looting a car, we all are unconsciously subject to the Broken Windows psychology. This makes it important to start a project clean, with as few shortcuts and as little technical debt as possible. This is because, as soon as a shortcut creeps in, it acts as a broken window and attracts more shortcuts.

Since a software project often is a very expensive and long-running endeavor, keeping broken windows at bay is a huge responsibility for us as software developers. We may not even be the ones finishing the project and others have to take over. For them, it's a legacy codebase they don't have a connection to, yet, lowering the threshold for creating broken windows even further.

There are times, however, when we decide a shortcut is the pragmatic thing to do, be it because the part of the code we're working on is not that important to the project as a whole, or because we're prototyping, or for economical reasons.

We should take great care to document such consciously added shortcuts, for example in the form of Architecture Decision Records (ADRs), as proposed by Michael Nygard in his blog.[42] We owe that to our future selves and our successors. If every member of the team is aware of this documentation, it will even reduce the Broken Windows effect, because the team will know that the shortcuts have been taken consciously and for good reason.

The following sections each discuss a pattern that can be considered a shortcut in the Hexagonal Architecture style presented in this book. We'll have a look at the effects of the shortcuts and the arguments that speak for and against taking them.

[42]http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions

Sharing models between use cases

In Chapter 5, Implementing a Use Case, I argued that different use cases should have different input and output models, meaning that the types of the input parameters and the types of the return values should be different.

Figure 11.1 shows an example where two use cases share the same input model.

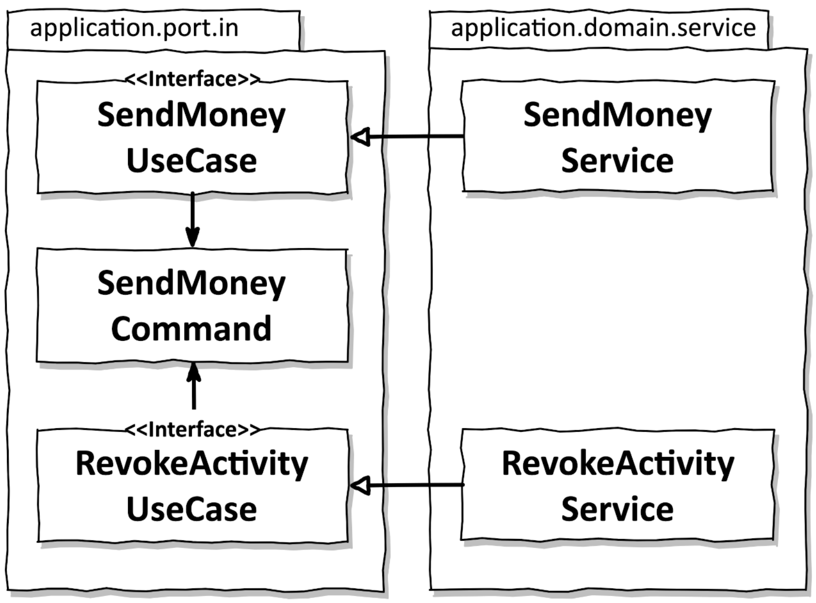

Figure 11.1 - Sharing the input or output model between use cases leads to coupling between the use cases.

The effect of sharing in this case is that `SendMoneyUseCase` and `RevokeActivityUseCase` are coupled to each other. If we change something within the shared `SendMoneyCommand` class, both use cases are affected. They share a reason to change in terms of the Single Responsibility Principle (which should be named the "Single Reason to Change Principle", as discussed Chapter 3, Inverting Dependencies). The same is true if both use cases share the same output model.

Sharing input and output models between use cases is valid if the use cases are functionally coupled, that is, if they share a certain requirement. In this case, *we actually want both use cases to be affected if we change a certain detail.*

If both use cases should be able to evolve separately from each other, however, this is a shortcut. In this case, we should separate the use cases from the start, even if it means duplicating input and output classes if they look the same at the start.

So, when building multiple use cases around a similar concept, it's worthwhile to regularly ask the question of whether the use cases should evolve separately from each other. As soon as the answer becomes "yes" it's time to separate the input and output models.

Using domain entities as the input or output model

If we have an `Account` domain entity and an incoming port `SendMoneyUseCase`, we might be tempted to use the entity as the input and/or output model of the incoming port, as Figure 11.2 shows.

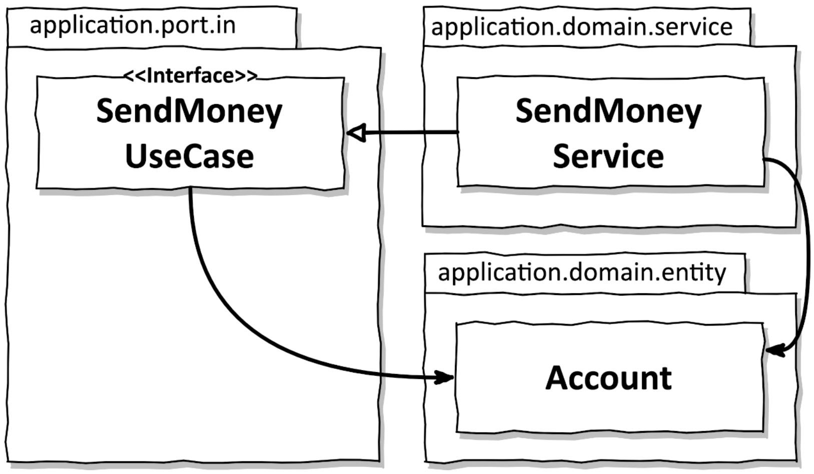

Figure 11.2 - Using a domain entity as the input or output model of a use case couples the domain entity to the use case.

The incoming port has a dependency on the domain entity. The consequence of this is that we've added another reason for the `Account` entity to change.

Wait, the `Account` entity doesn't have a dependency on the `SendMoneyUseCase` incoming port (it's the other way around), so how can the incoming port be a reason to change for the entity?

Say we need some information about an account in the use case that is not currently available in the `Account` entity. This information is ultimately not to be stored in the `Account` entity, however, but in a different domain or bounded context. We're tempted to add a new field to the `Account` entity nevertheless, because it's already available in the use case interface.

For simple create or update use cases, a domain entity in the use case interface may be fine, since the entity contains exactly the information we need to persist its state in the database.

As soon as a use case is not simply about updating a couple of fields in the database, but instead implements more complex domain logic (potentially delegating part of the domain logic to a rich domain entity), we should use a dedicated input and output model for the use case interface, because we don't want changes in the use case to propagate to the domain entity.

What makes this shortcut dangerous is the fact that many use cases start their lives as a simple create or update use case only to become beasts of complex domain logic over time. This is especially true in an agile environment where we start with a minimum viable product and add complexity as we move forward. So, if we used a domain entity as the input model at the start, we must find the point in time to replace it with a dedicated input model that is independent of the domain entity.

Skipping incoming ports

While the outgoing ports are necessary to invert the dependency between the application layer and the outgoing adapters (to make the dependencies point inward), we don't need the incoming ports for dependency inversion. We could decide to let the incoming adapters access our application or domain services directly, without incoming ports in between, as shown in Figure 11.3.

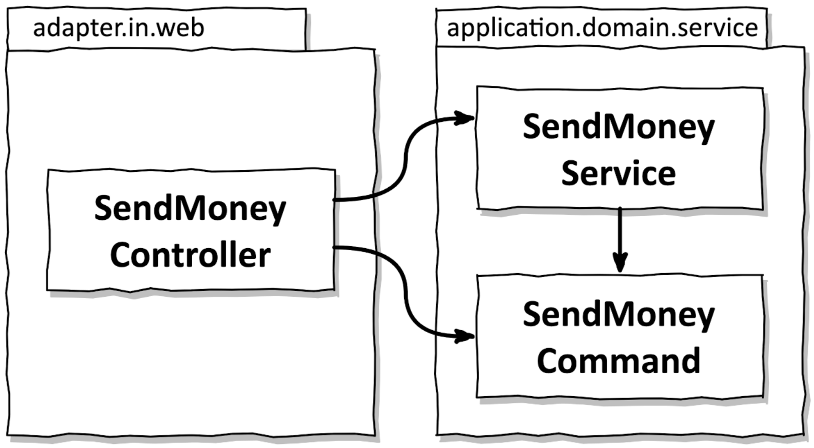

Figure 11.3 - Without incoming ports, we lose clearly marked entry points to the domain logic.

By removing the incoming ports, we have reduced a layer of abstraction between incoming adapters and the application layer. Removing layers of abstraction usually feels rather good.

The incoming ports, however, define the entry points into our application core. Once we remove them, we must know more about the internals of our application to find out which service method we can call to implement a certain use case. By maintaining dedicated incoming ports, we can identify the entry points to the application at a glance. This makes it especially easy for new developers to get their bearings in the codebase.

Another reason to keep the incoming ports is that they allow us to easily enforce architecture. With the enforcement options we'll learn about in Chapter 12, Enforcing Architecture Boundaries, we can make certain that incoming adapters only call incoming ports and not application services. This makes every entry point into the application layer a very conscious decision. We can no longer accidentally call a service method that was not meant to be called from an incoming adapter.

If an application is small enough or only has a single incoming adapter so that we can grasp the whole control flow without the help of incoming ports, we might want to do without incoming ports. However, how often can we say that we know that an application will stay small or will only ever have a single incoming adapter over its whole lifetime?

Skipping services

Aside from the incoming ports, for certain use cases we might want to skip the service layer as a whole, as Figure 11.4 shows.

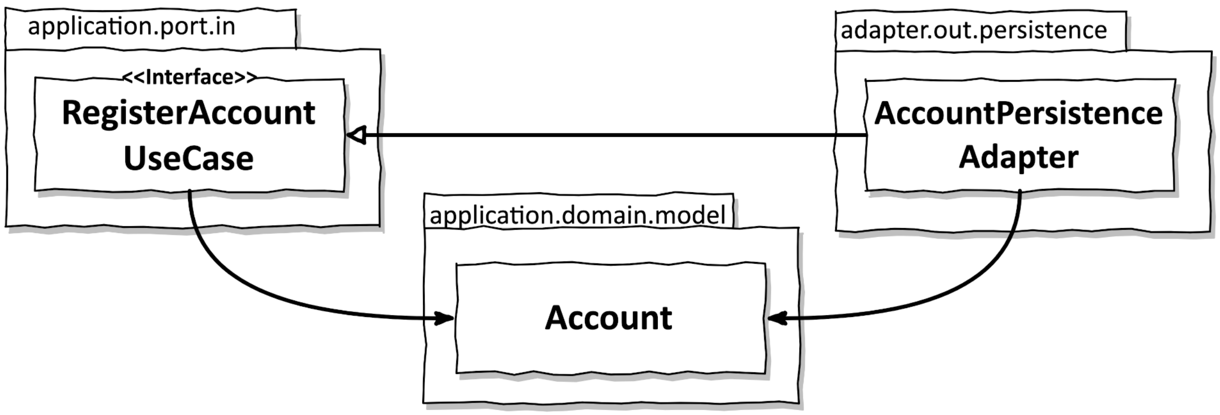

Figure 11.4 - Without services, we don't have a representation of a use case in our codebase anymore.

Here, the `AccountPersistenceAdapter` class within an outgoing adapter directly implements an incoming port and replaces the service that usually implements an incoming port.

It is very tempting to do this for simple CRUD use cases, since in this case a service usually only forwards a create, update, or delete request to the persistence adapter, without adding any domain logic. Instead of forwarding, we can let the persistence adapter implement the use case directly.

This, however, requires a shared model between the incoming adapter and the outgoing adapter, which is the `Account` domain entity in this case, so it usually means that we're using the domain model as input model as described previously.

Furthermore, we no longer have a representation of the use case within our application core. If a CRUD use case grows to something more complex over time, it's tempting to add domain logic directly to the outgoing adapter, since the use case has already been implemented there. This decentralizes the domain logic, making it harder to find and maintain.

In the end, to prevent boilerplate pass-through services, we might choose to skip the services for simple CRUD use cases after all. Then, however, the team should develop clear guidelines to introduce a service as soon as the use case is expected to do more than just create, update, or delete an entity.

How does this help me build maintainable software?

There are times when shortcuts make sense from an economic point of view. This chapter provided some insights into the consequences some shortcuts might have to help decide whether to take them or not.

The discussion shows that it's tempting to introduce shortcuts for simple CRUD use cases, since for them, implementing the whole architecture feels like overkill (and the shortcuts don't feel like shortcuts). Since all applications start small, however, it's very important for the team to agree on when a use case grows out of its CRUD state. Only then can the team replace the shortcuts with an architecture that is more maintainable in the long run.

Some use cases will never grow out of their CRUD state. For them, it might be more pragmatic to keep the shortcuts in place forever, as they don't really entail a maintenance overhead.

In any case, we should document the architecture and the decisions why we chose a certain shortcut so that we (or our successors) can re-evaluate the decisions in the future.

Even though shortcuts may be acceptable at times, we want to make the decision to take a shortcut consciously. That means that we should define one "right" way of doing things and enforce this way, so that we can deviate from that way if there are good reasons to do so. In the next chapter, we'll look at some ways of enforcing our architecture.

12. Enforcing Architecture Boundaries

We talked a lot about architecture in previous chapters and it feels good to have a target architecture to guide us in our decisions on how to craft code and where to put it.

In every above-playsize software project, however, architecture tends to erode over time. Boundaries between layers weaken, code becomes harder to test, and we generally need more and more time to implement new features.

In this chapter, we'll discuss some measures that we can take to enforce the boundaries within our architecture and thus fight architecture erosion.

Boundaries and dependencies

Before we talk about different ways of enforcing architecture boundaries, let's discuss where the boundaries lie within our architecture and what "enforcing a boundary" actually means.

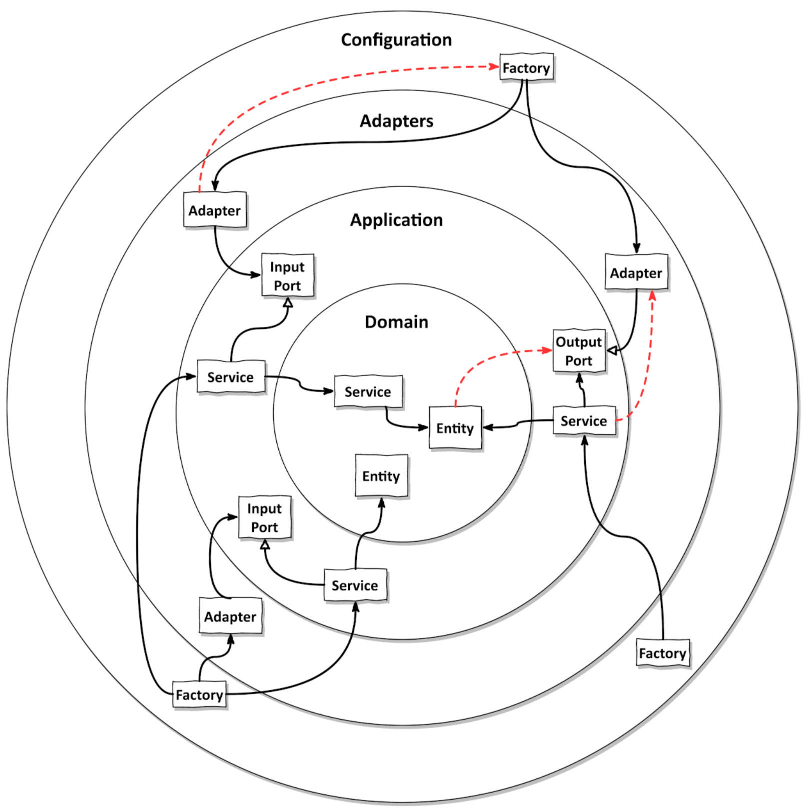

Figure 12.1 - Enforcing architecture boundaries means enforcing that dependencies point in the right direction. Dashed arrows mark dependencies that are not allowed according to our architecture.

Figure 12.1 shows how the elements of our Hexagonal Architecture might be distributed across four layers resembling the generic Clean Architecture approach introduced in Chapter 3, Inverting Dependencies.

The innermost layer contains domain entities and domain services. The application layer around it may access those entities and services to implement a use case, usually through an application service. Adapters access those services through incoming ports or are being accessed by those services through outgoing ports. Finally, the configuration layer contains factories that create adapter and service objects and provides them to a dependency injection mechanism.

In the preceding figure, our architecture boundaries become pretty clear. There is a boundary between each layer and its next inward and outward neighbor. According to the Dependency Rule, dependencies that cross such a layer boundary must always point inward.

This chapter is about ways to enforce the Dependency Rule. We want to make sure that there are no illegal dependencies that point in the wrong direction (dashed arrows in the figure).

Visibility modifiers

Let's start with the most basic tool that object-oriented languages in general, and Java in particular, provide us with to enforce boundaries: visibility modifiers.

Visibility modifiers have been a topic in almost every entry-level job interview I have conducted in the last couple of years. I would ask the interviewee which visibility modifiers Java provides and what their differences are.

Most of the interviewees only list the `public`, `protected`, and `private` modifiers. Only a few of them know the package-private (or "default") modifier. This is always a welcome opportunity for me to ask some questions about why such a visibility modifier would make sense in order to find out whether the interviewee can abstract from their previous knowledge.

So, why *is* the package-private modifier such an important modifier? Because it allows us to use Java packages to group classes into cohesive "modules". Classes within such a module can access each other, but cannot be accessed from outside of the package. We can then choose to make specific classes public to act as entry points to the module. This reduces the risk of accidentally violating the Dependency Rule by introducing a dependency that points in the wrong direction.

Let's have another look at the package structure discussed in Chapter 4, Organizing Code, with visibility modifiers in mind:

```
buckpal
├── adapter
|   ├── in
|   |   └── web
|   |       └── o SendMoneyController
|   └── out
|       └── persistence
|           ├── o AccountPersistenceAdapter
|           └── o SpringDataAccountRepository
├── application
|   ├── domain
|   |   ├── model
|   |   |   ├── + Account
|   |   |   └── + Activity
|   |   └── service
|   |       └── o SendMoneyService
|   └── port
|       ├── in
|       |   └── + SendMoneyUseCase
|       └── out
|           └── + UpdateAccountStatePort
└── common
```

We can make the classes in the `persistence` package package-private (marked with "o" in the tree above), because they don't need to be accessed by the outside world. The persistence adapter is accessed through the output ports it implements. For the same reason, we can make the `SendMoneyService` class package-private. Dependency injection mechanisms usually use reflection to instantiate classes, so they will still be able to instantiate those classes even if they're package-private.

With Spring, this approach only works if we use the classpath scanning approach discussed in Chapter 10, Assembling the Application, however, since the other approaches require us to create instances of those objects ourselves, which requires public access.

The rest of the classes in the example have to be `public` (marked with "+") as defined by our architecture: the `domain` package needs to be accessible by the other layers and the `application` layer needs to be accessible by the `web` and `persistence` adapters.

The package-private modifier is awesome for small modules with no more than a handful of classes. Once a package reaches a certain number of classes, however, it grows confusing to have so many classes in the same package. In this case, I like to create sub-packages to make the code easier to find (and, I admit, to satisfy my sense of aesthetics). This is where the package-private modifier fails to deliver, since Java treats sub-packages as different packages and we cannot access a package-private member of a sub-package. So, members in sub-packages must be `public`, exposing them to the outside world and thus making our architecture vulnerable to illegal dependencies.

Post-compile fitness function

As soon as we use the `public` modifier on a class, the compiler will let any other class use it, even if the direction of the dependency points in the wrong direction according to our architecture.

Since the compiler won't help us out in these cases, we have to find other means to check that the Dependency Rule isn't violated.

One way is to introduce a fitness function - a function that takes our architecture as input and determines its fitness in regard to a specific aspect. In our case, "fitness" is defined as "the Dependency Rule is not violated".

Ideally, a compiler runs a fitness function for us during compilation; but, lacking that, we can run such a function at runtime, after the code has already been compiled. Such runtime checks are best run during automated tests within a continuous integration build.

A tool that supports this kind of architectural fitness function for Java is ArchUnit.[43] Among other things, ArchUnit provides an API to check whether dependencies point in the expected direction. If it finds a violation, it will throw an exception. It's best run from within a test based on a unit testing framework such as JUnit, making the test fail in case of a dependency violation.

With ArchUnit, we can now check the dependencies between our layers, assuming that each layer has its own package, as defined in the package structure discussed in the previous section. For example, we can check that there is no dependency from the domain model on anything outside the domain model:

```
class DependencyRuleTests {

  @Test
  void domainModelDoesNotDependOnOutside() {
    noClasses()
        .that()
        .resideInAPackage("buckpal.application.domain.model..")
        .should()
        .dependOnClassesThat()
        .resideOutsideOfPackages(
                "buckpal.application.domain.model..",
                "lombok..",
                "java..")
        .check(new ClassFileImporter()
            .importPackages("buckpal.."));
  }
}
```

[43] https://github.com/TNG/ArchUnit

This rule validates the dependency rules visualized in Figure 12.2.

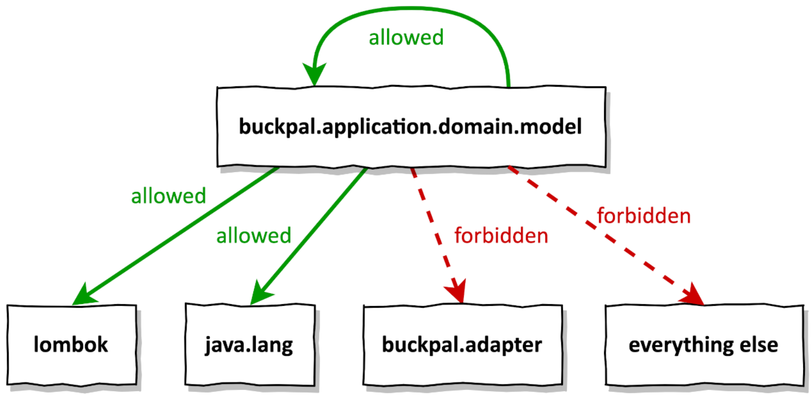

Figure 12.2 - Our domain model may access itself and some library packages, but it may not access code in any other packages, for example the packages containing our adapters. Inspired by the diagrams at https://www.archunit.org/use-cases.

The problem with the preceding rule is that, if we use some library code in the domain model, we have to add an exception to this rule for every dependency we introduce (like I did with `lombok` and `java.lang` in the example). In Chapter 14, A Component-Based Approach to Software Architecture, we will see a rule that doesn't have this problem.

With a little work, we can even create a kind of domain-specific language (DSL) on top of the ArchUnit API that allows us to specify all relevant packages within our Hexagonal Architecture and then automatically checks whether all dependencies between those packages point in the right direction:

```
class DependencyRuleTests {

  @Test
  void validateHexagonalArchitecture() {
    HexagonalArchitecture.basePackage("buckpal")
        .withApplicationLayer("application")
            .domainModel("domain.model")
            .services("domain.service")
            .incomingPorts("port.in")
            .outgoingPorts("port.out")
            .and()
        .withAdaptersLayer("adapter")
          .incoming("in.web")
          .outgoing("out.persistence")
          .and()
        .withConfiguration("configuration")
        .check(new ClassFileImporter()
            .importPackages("buckpal.."));
```

```
  }
}
```

In the preceding code example, we first specify the parent package of our application. We then go on to specify the sub-packages for the domain, adapter, application and configuration layers. The final call to `check()` will then execute a set of checks, verifying that the package dependencies are valid according to the Dependency Rule. The code for this Hexagonal Architecture DSL is available on GitHub if you would like to play around with it.[44]

While post-compile checks like previous one can be a great help in fighting illegal dependencies, they are not fail-safe. If we misspell the package name `buckpal` in the preceding code example, for example, the test will find no classes and thus no dependency violations. A single typo or, more importantly, a single refactoring renaming a package, can make the whole test useless. We should strive to make these tests refactoring-safe, or at least make them fail when a refactoring has broken them. In the preceding example, we could fail the test when one of the mentioned packages does not exist, for example (because it was renamed).

Build artifacts

Until now, our only tool for demarcating architecture boundaries within our codebase has been packages. All of our code has been part of the same monolithic build artifact.

A build artifact is the result of a (hopefully automated) build process. The most popular build tools in the Java world are currently Maven and Gradle. So, until now, imagine we had a single Maven or Gradle build script and we could call Maven or Gradle to compile, test and package the code of our application into a single JAR file.

A main feature of build tools is dependency resolution. To transform a certain codebase into a build artifact, a build tool first checks whether all artifacts the codebase depends on are available. If not, it tries to load them from an artifact repository. If this fails, the build will fail with an error before even trying to compile the code.

We can leverage this to enforce the dependencies (and thus, enforce the boundaries) between the modules and layers of our architecture. For each such module or layer, we create a separate build module with its own codebase and its own build artifact (JAR file) as a result. In the build script of each module, we specify only those dependencies on other modules that are allowed according to our architecture. Developers can no longer inadvertently create illegal dependencies because the classes are not even available on the classpath and they would run into compile errors.

[44] https://github.com/thombergs/buckpal/blob/master/src/test/java/io/reflectoring/buckpal/archunit/HexagonalArchitecture.java

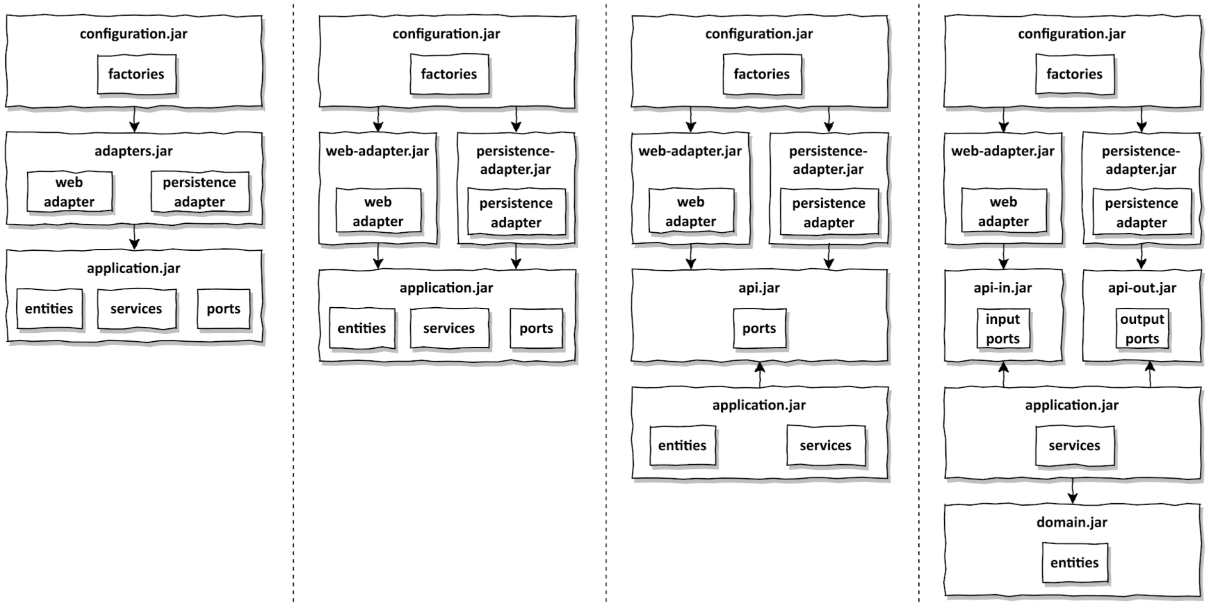

Figure 12.3 - Different ways of dividing our architecture into multiple build artifacts to prohibit illegal dependencies.

Figure 12.3 shows an incomplete set of options to divide our architecture into separate build artifacts.

Starting on the left, we see a basic three-module build with a separate build artifact for the configuration, adapter and application layers. The configuration module may access the adapters module, which in turn may access the application module. The configuration module may also access the application module due to the implicit, transitive dependency between them.

Note that the adapters module contains the web adapter as well as the persistence adapter. This means that the build tool will not prohibit dependencies between those adapters. While dependencies between those adapters are not strictly forbidden by the Dependency Rule (since both adapters are within the same outer layer), in most cases it's sensible to keep adapters isolated from each other. After all, we usually don't want changes in the persistence layer to leak into the web layer and vice versa (remember the Single Responsiblity Principle!). The same holds true for other types of adapters, for example adapters connecting our application to a certain third-party API. We don't want details of that API leaking into other adapters by adding accidental dependencies between adapters.

Thus, we may split the single adapters module into multiple build modules, one for each adapter, as shown in the second column of Figure 12.3.

Next, we could decide to split up the application module further. It currently contains the incoming and outgoing ports to our application, the services that implement or use those ports, and the domain entities that should contain much of our domain logic.

If we decide that our domain entities are not to be used as transfer objects within our ports (i.e. we want to disallow the "No Mapping" strategy from Chapter 9, Mapping between Boundaries), we can apply the Dependency Inversion Principle and pull out a separate "api" module that contains only the port interfaces (third column in Figure 12.3). The adapter modules and the application module may access the api module, but not the other way around. The api module does not have access to the domain entities and cannot use them within the port interfaces. Also, the adapters no longer have direct access to the entities and services, so they must go through the ports.

We can even go a step further and split the api module in two, one part containing only the incoming ports and the other part only containing the outgoing ports (fourth column in Figure 12.3). This way we can make it very clear whether a certain adapter is an incoming adapter or an outgoing adapter by declaring a dependency only on the input or the outgoing ports.

Also, we could split the application module even further, creating a module containing only the services and another containing only the domain model. This ensures that the domain model doesn't access the services and it would allow other applications (with different use cases and thus different services) to use the same domain model by simply declaring a dependency on the domain build artifact.

Figure 12.3 illustrates that there are a lot of different ways to divide an application into build modules, and there are of course more than just the four ways depicted in the figure. The gist is that the finer we cut our modules, the stronger we can control dependencies between them. The finer we cut, however, the more mapping we have to do between those modules, enforcing one of the mapping strategies introduced in Chapter 9, Mapping between Boundaries.

Besides that, demarcating architecture boundaries with build modules has a number of advantages over using simple packages as boundaries.

- First, build tools absolutely hate circular dependencies. Circular dependencies are bad because a change in one module within the circle would potentially mean a change in all other modules within the circle, which is a violation of the Single Responsibility Principle. Build tools don't allow circular dependencies because they would run into an endless loop while trying to resolve them. Thus, we can be sure that there are no circular dependencies between our build modules. The Java compiler, on the other hand, doesn't care at all if there is a circular dependency between two or more packages.
- Second, build modules allow isolated code changes within certain modules without having to take the other modules into consideration. Imagine we have to do a major refactoring in the application layer that causes temporary compile errors in a certain adapter. If the adapters and application layer are within the same build module, some IDEs will insist that all compile errors in the adapters must be fixed before we can run the tests in the application layer, even though the tests don't need the adapters to compile. If the application layer is in its own build module, however, the IDE won't care about the adapters at the moment, and we could run the application layer tests at will. The same goes for running a build process with Maven or Gradle: if both layers are in the same build module, the build would fail due to compile errors in either layer. So, multiple build modules allow isolated changes in each module. We could even choose to put each module into its own code repository, allowing different teams to maintain different modules.
- Finally, with each inter-module dependency explicitly declared in a build script, adding a new dependency becomes a conscious act instead of an accident. A developer who needs access to a certain class they currently cannot access will hopefully give some thought to the question if the dependency is really reasonable before adding it to the build script.

These advantages come with the added cost of having to maintain a build script, though, so the architecture should be somewhat stable before splitting it into different build modules.

Also, build modules tend to be less supple to change over time. Once chosen, we tend to stick with the modules we have initially defined. If the slicing of modules wasn't right from the start, we are less likely to correct it later because of the added effort of refactoring. Refactoring is easier when all the code lies within a single build module.

How does this help me build maintainable software?

Software architecture is basically all about managing dependencies between architecture elements. If the dependencies become a big ball of mud, the architecture becomes a big ball of mud.

So, to preserve the architecture over time, we need to continually make sure that dependencies point in the right direction.

When producing new code or refactoring existing code, we should keep the package structure in mind and use package-private visibility when possible, to avoid dependencies on classes that should not be accessed from outside the package.

If we need to enforce architecture boundaries within a single build module, and the package-private modifier doesn't work because the package structure won't allow it, we can make use of post-compile tools such as ArchUnit.

Anytime we feel that the architecture is stable enough we should extract architecture elements into their own build modules, because this gives explicit control over the dependencies.

All three approaches can be combined to enforce architecture boundaries and thus keep the codebase maintainable over time.

In the next chapter, we'll continue to explore architecture boundaries, but from a different perspective: we'll think about how to manage multiple domains (or bounded contexts) in the same application, while keeping the boundaries between them distinct.

13. Managing Multiple Bounded Contexts

Many applications consist of more than one domain, or, to stick with Domain-Driven Design language, more than one bounded context. The term "bounded context" tells us that there should be boundaries between the different domains. If we don't have boundaries between different domains, there are no restrictions on dependencies between classes in these domains. Eventually, dependencies will grow between the domains, coupling them together. This coupling means that the domains can no longer evolve in isolation, but can only evolve together. We could just as well not have separated our code into different domains in the first place!

The whole reason to separate code into different domains is so that these domains can evolve in isolation. This is an application of the Single Responsibility Principle, discussed in Chapter 3, Inverting Dependencies. Only this time, we're not talking about the responsibilities of a single class, but about the responsibilities of a whole group of classes that make up a bounded context. If the responsibilities of one bounded context change, we don't want to change the code for other bounded contexts!

Managing bounded contexts, that is, keeping the boundaries between them clear, is one of the main challenges of software engineering. Many of the pains developers associate with so-called "legacy software" stem from unclear boundaries. And it turns out that software doesn't need long to become "legacy".

So, unsurprisingly (at least in retrospect), many readers of the first edition of this book asked me how to manage multiple bounded contexts with Hexagonal Architecture. Unfortunately, the answer is not simple. As is so often the case, there are multiple ways to go about it and none of them is right or wrong per se. Let's discuss some ways of separating bounded contexts.

One hexagon per bounded context?

When working with Hexagonal Architecture and multiple bounded contexts, our reflex is to create a separate "hexagon" for each bounded context. The result would look something like Figure 13.1.

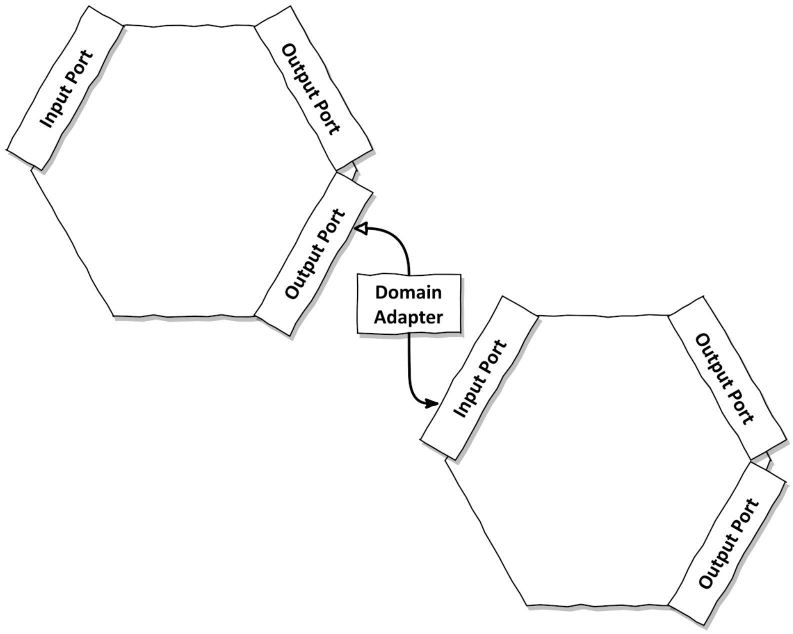

Figure 13.1 - If each bounded context is implemented as its own hexagon, we need an outgoing port, an adapter, and an incoming port for each line of communication between bounded contexts

Each bounded context lives in its own hexagon, providing input ports to interact with it and using output ports to interact with the outside world.

Ideally, the bounded contexts don't need to talk to each other at all, so we don't have any dependencies between the two. In the real world, however, this is rarely the case. Let's assume that the bounded context on the left needs to call some functionality of the bounded context on the right.

If we use the architecture elements that Hexagonal Architecture provides us with, we add an output port to the first bounded context and an input port to the second bounded context. Then, we create an adapter that implements the output port, does any necessary mapping, and calls the input port of the second bounded context.

Problem solved, right?

Indeed, on paper this looks like a very clean solution. The bounded contexts are optimally separated from each other. The dependencies between them are clearly structured in the form of ports and adapters. New dependencies between bounded contexts require us to explicitly add them to the existing ports or to add a new port. It's unlikely that dependencies creep in "by accident" because there is a lot of ritual involved in creating such a dependency.

If we think further than just two bounded contexts, however, it becomes apparent that this architecture doesn't scale very well. For two bounded contexts with one dependency, we need to implement one adapter (the box named "Domain Adapter" in the figure above). If we exclude circular dependencies, we might have to implement three adapters for three bounded contexts, six adapters for four bounded contexts, and so on, as shown in Figure 13.2.[45]

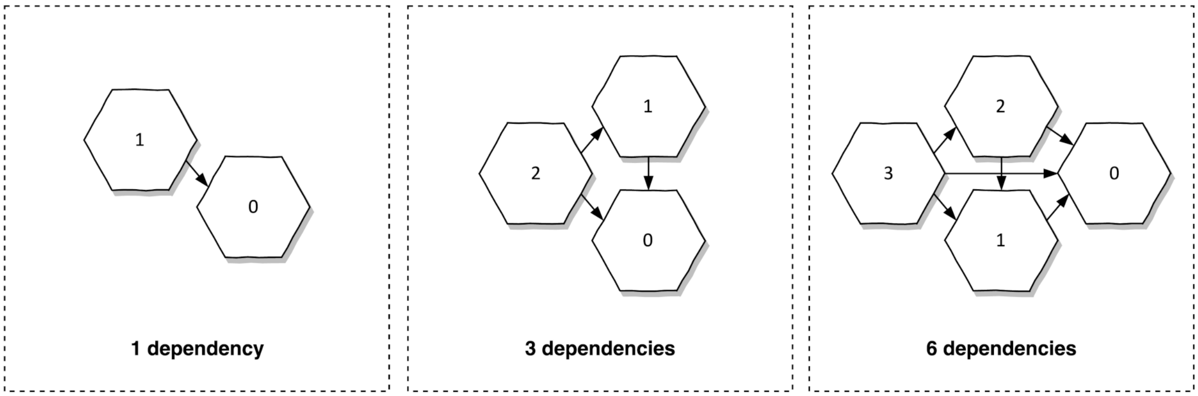

Figure 13.2 - The number of potential dependencies between bounded contexts grows disproportionately to the number of bounded contexts, even if we exclude circular dependencies.

For each dependency, we would have to implement one adapter with at least one associated input and output port. Each adapter would have to map from one domain model to another. This quickly becomes a chore to develop and maintain. If it's a chore and requires more effort than it brings value, the team will take shortcuts to avoid it, resulting in an architecture that looks like a Hexagonal Architecture at first glance but doesn't have the benefits it promises.

If we look at the original article introducing Hexagonal Architecture[46], it was never the intent of Hexagonal Architecture to encapsulate a single bounded context in ports and adapters. Instead, the intent is to encapsulate *an application*. This application may consist of many bounded contexts or none at all.

It *does* make sense to wrap each bounded context in its own hexagon when weâ€™re preparing to extract them into their own applications, that is, their own (micro)services. That means we should be very certain that the boundaries weâ€™re putting between them are the right boundaries, however, and we donâ€™t expect them to change.

The takeaway here is that Hexagonal Architecture doesn't provide a scalable solution for managing multiple bounded contexts in the same application. And it doesn't have to. We can instead take inspiration from Domain-Driven Design to decouple our bounded contexts, because within a hexagon, we can do whatever we like.

Decoupled bounded contexts

In the previous section, we learned that the ports and adapters should encapsulate the whole application, not each bounded context separately. How do we keep the bounded contexts separate from each other, then?

[45]The formula I used to calculate the potential dependencies between *n* bounded contexts is *n-1 + n-2 + … + 1*. The first bounded context has n-1 potential, non-circular dependencies, the second n-2, and so on. The last bounded context cannot have any dependency on another bounded context, because every dependency it can have would be a circular dependency, and we don't want to allow circular dependencies.

[46]https://alistair.cockburn.us/hexagonal-architecture/

In a simple case, we might have bounded contexts that don't communicate with each other. They provide completely separate paths through the code. In this case, we could build dedicated input and output ports for each bounded context like in Figure 13.3.

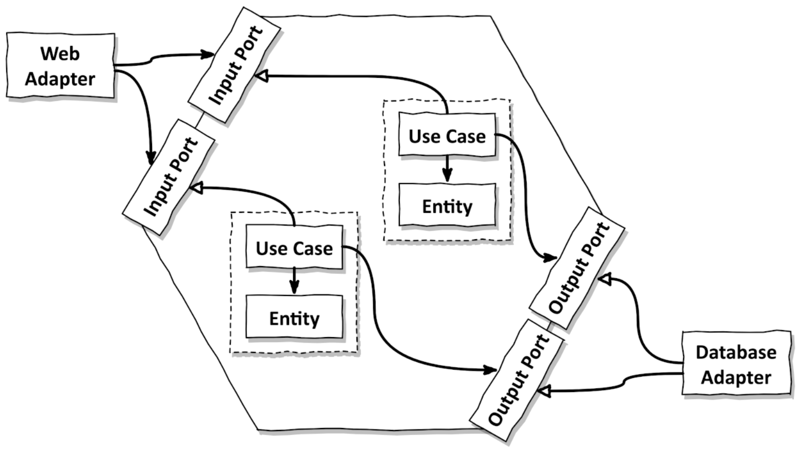

Figure 13.3 - If bounded contexts (dashed lines) don't need to talk to each other, each can implement its own input ports and call its own output ports.

This example shows a Hexagonal Architecture with two bounded contexts. A web adapter is driving the application and a database adapter is driven by the application. These adapters are representative of any other input and output adapters - not every application is a web application with a database.

Each bounded context exposes its own use cases via one or more dedicated input ports. The web adapter knows all input ports and thus can call the functionality of all bounded contexts.

Instead of having dedicated input ports for each of our bounded contexts, we could also implement one "broad" input port through which the web adapter routes requests to *multiple* bounded contexts. In this case, the boundaries between the contexts would be hidden from the outside of our hexagon. This may or may not be desirable depending on the situation.

Furthermore, each bounded context defines its own output port to the database so that it can store and retrieve its data independently of any other bounded context.

While splitting the input ports per bounded context is optional, I would strongly recommend keeping the output ports that store and retrieve the domain data for a bounded context separate from other bounded contexts. If one bounded context is concerned with financial transactions and the other with user registrations, there should be one (or more) output port that is dedicated to storing and retrieving transaction data and another dedicated to storing and retrieving registration data.

Each bounded context should have its own persistence. If bounded contexts share output ports to store and retrieve data, they will quickly become strongly coupled because they both depend on the same data model. Imagine that we need to pull one bounded context out of the Hexagonal application and into its own microservice because we learned that it has different scalability requirements from the rest of the application. If that bounded context shares a database model with another bounded context, it becomes very hard to extract. We wouldn't want the new microservice to reach into another application's database, would we? For the same reason, we want to keep the database model of each bounded context separate.

As long as multiple bounded contexts are executed in the same runtime, they might share a physical database and participate in the same database transactions. But within that database, there should be clear boundaries between the data of different bounded contexts, for example, in the form of a separate database schema, or at least different database tables.

Splitting up the input and output ports like this has the nice effect that the bounded contexts are completely decoupled. Each bounded context can evolve by itself without affecting the others in any way. But they are only decoupled because they're not talking to each other. What if we have use cases that span multiple bounded contexts or if one bounded context needs to speak to another?

Appropriately coupled bounded contexts

If all coupling could be avoided, software architecture would be a lot easier. In real-world applications, a bounded context very likely needs the help of another bounded context to do its work.

An example is again our bounded context that is concerned with money transactions. For security reasons, we'll want to log which user has issued a transaction. That means that our bounded context needs some information about the user, which lives in another bounded context. But our bounded context doesn't need to be tightly coupled to the user management context.

Instead of having to know the whole user object in our "transaction management" bounded context, it might be enough to just know the user's ID. While a user object in the "registration" context is a complex object with many attributes, a representation of a user in the transaction context may only be a wrapper around the user ID. In the "send money" use case we could now just accept the ID of the user executing the transaction as input and then log it. We don't need to couple the transaction context to all the other details of a user.

But we might want to validate that the user is not blocked from transactions. In this case, we can use a domain event.[47] Whenever the status of a user changes in the user management context, we trigger a domain event that can be received by other bounded contexts. Our transaction context might listen to events when a user is newly registered or has been blocked, for example. It can then store that information in its own database for later use in the "send money" use case to validate the status of the user.

Another possible solution is to introduce an application service as the orchestrator between the user management and transaction contexts.[48] The application service implements the "send money" input port. When called, it first asks the user management bounded context for the status of the user and then passes the status into the "send money" use case provided by the transaction context. A different implementation, but the same effect as when using domain events.

[47] *Implementing Domain-Driven Design* by Vaughn Vernon, Pearson, 2013, Chapter 8.
[48] *Implementing Domain-Driven Design* by Vaughn Vernon, Pearson, 2013, Chapter 14.

These were just two examples of how to "appropriately" couple bounded contexts. If you haven't yet, I recommend reading through the Domain-Driven Design literature to get inspired.

Coming back to Hexagonal Architecture, appropriately coupling multiple bounded contexts may look something like in Figure 13.4.

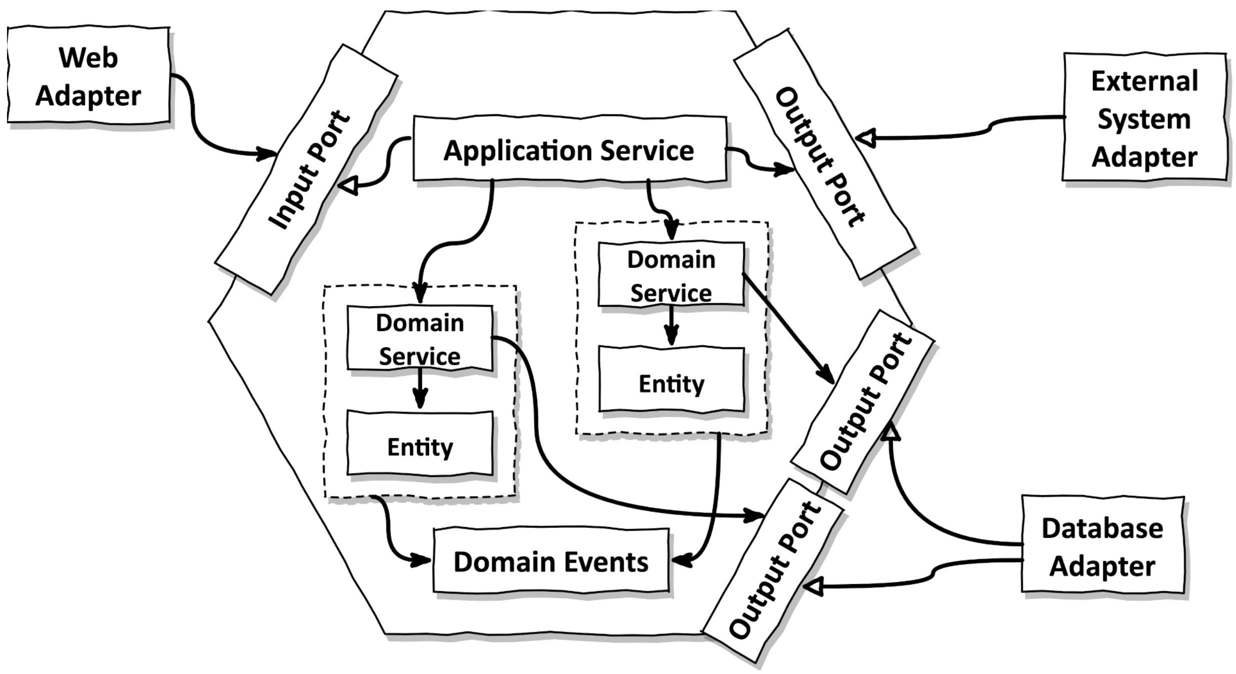

Figure 13.4 - If we have use cases spanning multiple bounded contexts, we can introduce an application service to orchestrate and domain events to share information between contexts.

We have introduced an application service as the orchestrator above our bounded contexts. The input ports are now implemented by this service instead of by the bounded contexts themselves. The application service may call output ports to get the required information from other systems and then calls one or more domain services provided by the bounded contexts. In addition to orchestrating the calls to the bounded contexts, the application service also acts as a transaction boundary so that we can call multiple domain services in the same database transaction, for example.

The domain services within the bounded contexts each still use their own database output ports to keep the data model between the bounded contexts separated. We may decide that this separation is not necessary and use a single database output port instead (but we should be aware that sharing a data model leads to very tight coupling).

The bounded contexts have access to a set of shared domain events that they can emit and listen to, respectively, to exchange information in a loosely coupled fashion.

How does this help me build maintainable software?

Managing boundaries between domains is one of the hardest parts of software development. In a small codebase, boundaries might not be necessary because the mental model of the whole codebase still fits into our brain's working memory. But as soon as the codebase reaches a certain size, we should make sure to introduce boundaries between domains, so we can reason about each domain in isolation. If we don't do this, dependencies will creep in, turning our codebase into one of those dreaded "big balls of mud".

Hexagonal Architecture is all about managing a boundary *between an application and the outside world*. The boundary is made up of certain input ports provided by the application and certain output ports expected by the application.

Hexagonal Architecture does *not* help us to manage finer-grained boundaries within our application. Inside our "hexagon", we can do whatever we want. If the codebase gets too big for our working memory, we should fall back to Domain-Driven Design or other concepts to create boundaries within our codebase.

In the next chapter, we will explore a lightweight method of creating boundaries that we can use with or without Hexagonal Architecture.

14. A Component-Based Approach to Software Architecture

When we're starting a software project, we never know all the requirements that the users will throw at us once they are actually using the software. A software project is always associated with taking chances and making educated guesses (we like to call them "assumptions" to make it sound more professional). The environment of a software project is just too volatile to know in advance how everything will play out. This volatility is why the Agile movement was born. Agile practices make organizations flexible enough to adapt to change.

But how can we create a software architecture that can cope with such an agile environment? If everything can change at any time, should we even bother with architecture?

Yes, we should. As discussed in Chapter 1, Maintainability, we should make sure that our software architecture enables maintainability. A maintainable codebase can evolve over time, adapting to external factors.

Hexagonal Architecture takes a big step toward maintainability. It's creating a boundary between our application and the outside world. On the inside of our application (within the hexagon), we have our domain code, which provides dedicated ports to the outside world. These ports connect the application to adapters, which talk to the outside world, translating between the language of our application and the languages of outside systems. This architecture enhances maintainability because the application can mostly evolve independently of the outside world. As long as the ports don't change, we can evolve anything within the application to react to changes in the agile environment.

But, as we learned in Chapter 13, Managing Multiple Bounded Contexts, Hexagonal Architecture doesn't help us to create boundaries *within* our application core. We might want to apply a different architecture within our application core that helps us in this regard.

Also, I've heard quite a few times that Hexagonal Architecture feels hard, especially for a software project just starting out. It's hard to get the team on board because not everyone understands the value of dependency inversion and the mapping between the domain model and the outside world. Hexagonal Architecture might just be overkill for a fledgling application.

For cases like this, we might want to start out with a simpler architecture style that still provides the modularity we need to evolve into something else in the future but that's simple enough to get everyone on board. I propose that a component-based architecture is a good starting point, and we'll use this chapter to discuss this architecture style.

Modularity through components

One of the drivers of maintainability is modularity. Modularity allows us to conquer the complexity of a software system by dividing it into simpler modules. We don't have to understand the whole system to be able to work on one specific module. Instead, we can focus on that one module and potentially the modules it interfaces with. Modules can evolve mostly independently of each other, as long as the interfaces between modules are clearly defined. We're probably able to fit a mental model of one module into our working memory, but good luck with creating a mental model if there are no modules in the codebase. We would jump around in the code rather helplessly.

Only modularity allows us humans to create complex systems. In his book "Modern Software Engineering", Dave Farley talks about the modularity of the Apollo space program[49]:

> *This modularity had lots of advantages. It meant that each component could be built to focus on one part of the problem and would need to compromise less in its design. It allowed different groups - in this case completely different companies - to work on each module largely independently of the others. As long as the different groups agreed on how the modules would interface with each other, they could work to solve the problems of their module without constraint.*

Modularity allowed us to go to the moon! Modularity allows us to build cars, aircraft, and buildings. It should be no surprise that it also helps us build complex software.

But what is a "module"? I feel the term is overloaded in (object-oriented) software development. Everything and its cat is called a "module", even if it's just a bunch of classes that were haphazardly thrown together to do a useful thing. I prefer the term "component" to describe a group of classes that were thoughtfully engineered to implement certain functionality that can be *composed* together with other groups of classes to build a complex system. The composition aspect implies that components can be composed to form a bigger whole and potentially even re-composed to react to changes in the environment. Composability requires a component to define a clear interface that tells us what it provides to and needs from the outside world (input and output ports, anyone?). Think of LEGO bricks. A LEGO brick provides a certain layout of studs for other bricks to attach to, and it requires a certain layout of studs to attach to other bricks. All that said, I won't judge you if you use the term "module", but I'll refer of "components" in the rest of this chapter.

For the sake of this chapter, a component is a set of classes that has a dedicated namespace and a clearly defined API. If another component needs this component's functionality, it can call it via its API, but it may not reach into its internals. A component may be made up of smaller components. By default, these sub-components live inside the internals of the parent component, so that they are not accessible from the outside. They can, however, contribute to the parent component's API if they implement functionality that should be accessible from the outside.

Like any other architecture style, component-based architecture is all about which dependencies are allowed and which are discouraged. This is illustrated in Figure 14.1.

[49] *Modern Software Engineering* by Dave Farley, Pearson, 2022, chapter 6.

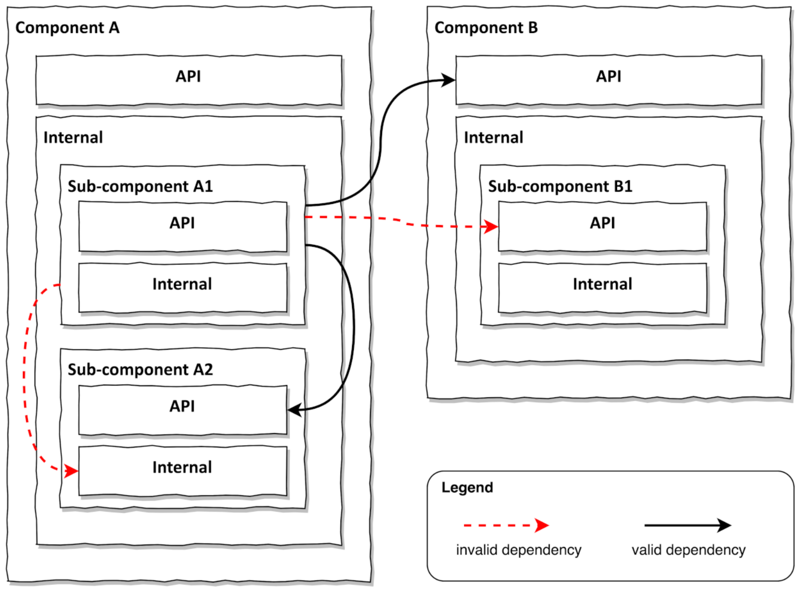

Figure 14.1 - Dependencies on an internal package are invalid. Dependencies on an API package are valid, provided that API package is not nested in an internal package.

Here, we have two top-level components, A and B. Component A is made up of two sub-components, A1 and A2, while component B only has a single sub-component B1.

If A1 needs access to B's functionality, it can get it by calling B's API. It cannot, however, access B1's API, because, as a sub-component, it's part of its parent's internals and thus hidden from the outside. B1 can still contribute functionality to its parent's API, though, by implementing an interface in the parent API. We will see this in action in the case study later.

The same rules apply between the sibling components A1 and A2. If A1 needs access to A2's functionality, it can call its API, but it cannot call into A2's internals.

And that's all there is to component-based architecture. It can be summarized in four simple rules:

1. A component has a dedicated namespace to be addressable.
2. A component has a dedicated API and internals.
3. A component's API may be called from the outside, but its internals may not.
4. A component may contain sub-components as part of its internals.

To make the abstract concrete, let's see a component-based architecture in real code.

Case study: Building a "Check Engine" component

As a case study for the component-based architecture presented in this chapter, I extracted a component from a real software project I worked on into a standalone GitHub repository.[50] The

[50]https://github.com/thombergs/components-example

fact alone that I extracted the component with relatively little effort and that we can reason about this component *without knowing anything about the software project it comes from* shows that we have successfully conquered complexity by applying modularity!

The component is written in object-oriented Kotlin, but the concepts apply to any other object-oriented language.

The component is called "check engine". It was meant to be a kind of web scraper that goes through web pages and runs a set of checks against them. These checks can be anything from "check that the HTML on that web page is valid" to "return all spelling errors on that web page".

Since a lot can go wrong when scraping web pages, we decided to run the checks asynchronously. That means that the component needs to provide an API to schedule checks and an API to retrieve the results of a check after it has been executed. This implies a queue in which to store incoming check requests and a database in which to store the results of these checks.

From the outside, it doesn't matter if we build the check engine "in one piece" or split it up into sub-components. As long as the component has a dedicated API, these details are hidden from the outside. The requirements above, however, outline certain natural boundaries for sub-components within the check engine. Breaking the check engine up along these boundaries allows us to manage complexity *within* the check engine component because each sub-component will be simpler to manage than the whole problem.

We came up with three sub-components for the check engine:

- A queue component that wraps the access to a queue to queue and de-queue check requests.
- A database component that wraps the access to a database to store and retrieve check results.
- A checkrunner component that knows which checks to run and runs them whenever a check request comes in from the queue.

Note that these sub-components introduce mostly technical boundaries. Very similarly to output adapters in Hexagonal Architecture, we're hiding away the specifics of accessing an external system (queue and database) in sub-components. But then, the check engine component is a very technical component with little to no domain code. The only component that we could consider "domain code" is the checkrunner, which acts as a controller of sorts. Technical components lend themselves very well to a component-based architecture because the boundaries between them are clearer than the boundaries between different functional domains.

Let's take a look at an architecture diagram of the check engine component to dig into the details (Figure 14.2).

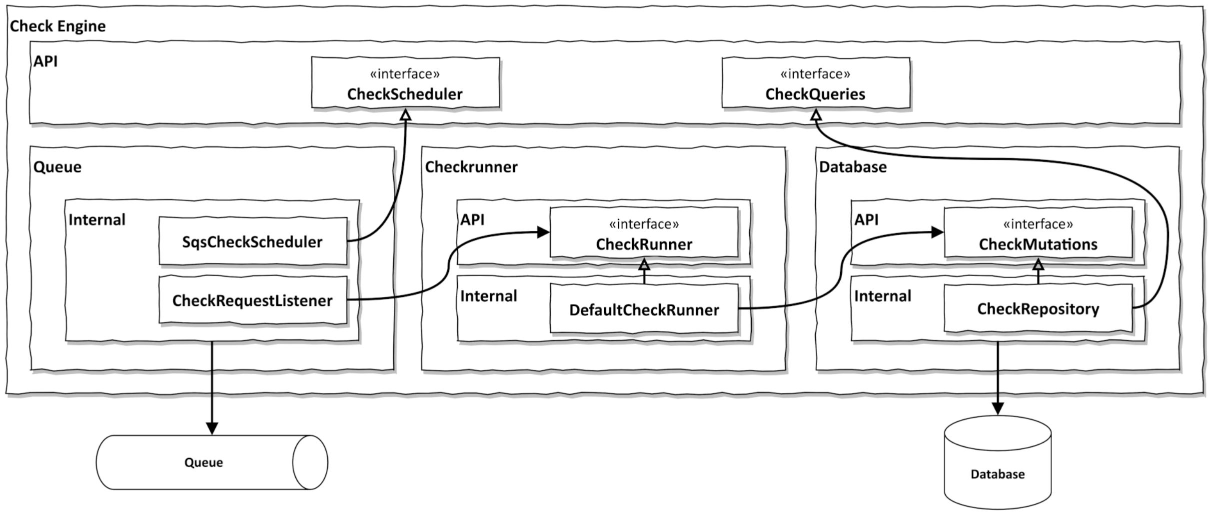

Figure 14.2 - The check engine component is made up of three sub-components that contribute to the parent component's API.

The diagram mirrors the structure of the code. You can think of each box as a Java package (or a simple source code folder in other programming languages). If a box is within a larger box, it's a sub-package of that larger box. The boxes at the lowest level, finally, are classes.

The public API of the check engine component consists of the `CheckScheduler` and `CheckQueries` interfaces which allow scheduling a web page check and retrieving the check results, respectively.

`CheckScheduler` is implemented by the `SqsCheckScheduler` class living in the queue component internals. This way, the queue component contributes to the parent component's API. Only when we look at the name of this class does it tell us that it's using Amazon's "Simple Queue Service" (SQS). This implementation detail is not leaked to the outside of the check engine component. Not even the sibling components know which queue technology is used. You might notice that the queue component doesn't even have an API package, so *all* of its classes are internal!

The `CheckRequestListener` class, then, listens to incoming requests from the queue. For each incoming request, it calls the `CheckRunner` interface in the checkrunner sub-component's API. `DefaultCheckRunner` implements that interface. It reads the web page URL from the incoming request, determines which checks to run against it, and then runs those checks.

When a check has finished, the `DefaultCheckRunner` class stores the results in the database by calling the `CheckMutations` interface of the database sub-component's API. This interface is implemented by the `CheckRepository` class which handles the details of connecting and talking to a database. Again, the database technology is not leaked to the outside of the database sub-component.

The `CheckRepository` class also implements the `CheckQueries` interface, which is part of the check engine's public API. This interface provides methods to query for check results.

By splitting up the check engine component into three sub-components, we have divided the complexity. Each sub-component solves a simpler part of the overall problem. It can evolve mostly by itself. A change in queue or database technologies because of costs, scalability, or other reasons doesn't leak into other sub-components. We could even replace the sub-components with simple in-memory implementations for tests if we wanted.

All this we get by structuring our code into components, following the convention of having dedicated API and internal packages.

Enforcing component boundaries

Conventions are good to have, but if that's all there is, someone will break them, and the architecture will erode. We need to enforce the conventions of the component architecture.

The nice thing about the component architecture is that we can apply a relatively simple fitness function to make sure that no accidental dependencies have crept into our component architecture:

> *No classes that are outside of an "internal" package should access a class inside of that "internal" package.*

If we put all the internals of a component into a package called "internal" (or a package marked as "internal" in some other way), we just have to check that no class in that package is called from outside of that package. For JVM-based projects, we can codify this fitness function with ArchUnit[51]:

```kotlin
fun assertPackageIsNotAccessedFromOutside(internalPackage: String) {
    noClasses()
        .that()
        .resideOutsideOfPackage(packageMatcher(internalPackage))
        .should()
        .dependOnClassesThat()
        .resideInAPackage(packageMatcher(internalPackage))
        .check(analyzedClasses)
}
```

We just need a way to identify internal packages during each build and feed them all into the function above, and the build will fail if we have accidentally introduced a dependency on an internal class.

The fitness function doesn't even need to know anything about the components in our architecture. We just need to follow a convention for identifying internal packages and then feed those packages into the function. This means that we don't need to update the test that's running the fitness function whenever we add or remove a component to/from the codebase. Very convenient!

Note that this fitness function is an inverted form of the fitness function we introduced in Chapter 12, Enforcing Architecture Boundaries. In Chapter 12, we verified that classes from a certain package don't access classes *outside* of that package. Here, we verify that classes from outside the package

[51]You can see this fitness function in action in the class `InternalPackageTests` at https://github.com/thombergs/components-example/blob/-main/server/src/test/kotlin/io/reflectoring/components/InternalPackageTest.kt.

are not accessing classes *inside* the package. This fitness function is much more stable, as we don't have to add exceptions for every library we're using.

We can still introduce unwanted dependencies by just not following our convention for internal packages, of course. And the rule still allows a loophole: if we put classes directly into the "internal" package of a top-level component, the classes of any sub-components may access it. So, we might want to introduce another rule that disallows any classes directly in the "internal" package of a top-level component.

How does this help me build maintainable software?

Component-based architecture is very simple. As long as each component has a dedicated namespace, dedicated API and internal packages, and classes within an internal package are not called from the outside, we get a very maintainable codebase consisting of many composable and re-composable components. If we add the rule that components may be composed of other components, we can build a whole application out of smaller and smaller parts where each part solves a simpler problem.

Even though there are loopholes to get around the rules of the component architecture, the architecture itself is so simple that it's very easy to understand and communicate. If it's easy to understand, it's easy to maintain. If it's easy to maintain, the loopholes are less likely to be exploited.

Hexagonal Architecture cares about boundaries at the application level. Component-based architecture cares about boundaries at the component level. We can use this to embed components within a Hexagonal Architecture, or we can choose to start out with a simple component-based architecture and evolve it in any other architecture should the need arise. A component-based architecture is modular by design and modules are easy to move around and refactor.

In the next and last chapter, we'll close the discussion around architecture and try to answer the question of when we should choose which architecture style.

15. Deciding on an Architecture Style

So far, this book has provided an opinionated approach to building a web application in a Hexagonal Architecture style. From organizing code to taking shortcuts, we have answered many questions that this architecture style confronts us with.

Some of the answers in this book can be applied to the conventional layered architecture style. Some answers can only be implemented in a domain-centric approach such as the one proposed in this book. And some answers you might not even agree with, because they haven't worked in your experience.

The final questions we want answers for, however, are these: When should we actually use the Hexagonal Architecture style? And when should we rather stick with the conventional layered style (or any other style for that matter)?

Start simple

An important point that took me far too long to realize is that software architecture isn't just something we define at the beginning of a software project that will take care of itself after. We can't know everything we need to know to design a great architecture at the beginning of a project! The architecture of a software project can and should evolve over time to adapt to changing requirements.

This means that we won't know which architecture style will be the best for the software project in the long run, and we might need to change the architecture style in the future! To make this possible, we need to make certain that our software is supple to change. We need to plant a seed of maintainability.

Maintainability means that we need to make our code modular so that we can work on each module in isolation and move it around in the codebase should the need arise. Our architecture needs to make the boundaries between those modules as clear as possible so that unwanted dependencies between those modules don't accidentally creep in, reducing maintainability.

The start of a project might only involve a collection of CRUD use cases and a domain-centric architecture such as Hexagonal Architecture might be overkill, so we opt for something simpler such as the component-based approach. Or we might know enough about the project already that we start building out a rich domain model, in which case the Hexagonal Architecture style might be the right one to start with.

Evolve the domain

Over time, we learn more and more about the requirements of our software, and we can make better and better decisions about the best architecture style. The application might evolve from a collection of simple CRUD use cases to a rich domain-centric application with a lot of business rules. At this point, the Hexagonal Architecture style becomes a good option.

It should have become clear in the previous chapters that the main feature of a Hexagonal Architecture style is that we can develop domain code free from diversions such as persistence concerns and dependencies on external systems. In my opinion, evolving domain code free from external influence is the single most important argument for the Hexagonal Architecture style.

This is why this architecture style is such a good match for DDD practices. To state the obvious, in DDD the domain drives the development, and we can best reason about the domain if we don't have to think about persistence concerns and other technical aspects simultaneously.

I would even go so far as to say that *domain-centric architecture styles such as the Hexagonal style are enablers of DDD*. Without an architecture that puts the domain at the center of things, and without inverting dependencies toward the domain code, we have no chance of really doing DDD. The design will always be driven by other factors.

So, as a first indicator of whether to use the architecture style presented in this book or not: *if the domain code is not the most important thing in your application, you probably don't need this architecture style.*

Trust your experience

We're creatures of habit. Habits automate decisions for us, so we don't have to spend time on them. If there's a lion running toward us, we run. If we build a new web application, we use the layered architecture style. We have done it so often in the past that it has become a habit.

I'm not saying that habitual decisions are necessarily bad decisions. Habits are just as good at helping to make a good decision as they are at helping to make a bad one. I'm saying that we're doing what we're experienced in. We're comfortable with what we've done in the past, so why should we change anything?

Therefore, the only way to make an educated decision about an architecture style is by having experience in different architecture styles. If you're unsure about the Hexagonal Architecture style, try it out on a small module of the application that you're currently building. Get used to the concepts. Get comfortable. Apply the ideas in this book, modify them, and add your own ideas to develop a style that you're comfortable with.

This experience can then guide your next architecture decision.

It depends

I would love to provide a list of multiple-choice questions to decide on an architecture style just like all those "Which personality type are you?" and "If you were a dog, what kind of dog would you be?" tests that regularly swirl around on the social media.[52]

[52] In case you wanted to know, I'm the "Defender" personality type and if I were a dog, I would apparently be a Pit Bull.

However, it isn't as easy as that. My answer to the question of which architecture style to choose remains the professional consultant's "It depends…". It depends on the type of software to be built. It depends on the role of the domain code. It depends on the experience of the team. And finally, it depends on being comfortable with a decision.

I hope, however, that this book has provided some sparks of inspiration to help with the architecture question. If you have a story to tell about architecture decisions, with or without Hexagonal Architecture, I'd love to hear about it.[53]

[53]You can drop me an email to tom@reflectoring.io.

Changelog

A list of changes in the revisions of this book.

First edition

Revision	Date	Changes
0.1	2019-04-02	First published version, containing chapters 1 through 7.
0.2	2019-05-29	Added chapter 9 "Assembling the Application". Fixed some typos. Streamlined all 28 occurrences of "codebase" and "code base" into "codebase". I'm talking a lot about code, apparently.
0.3	2019-06-20	Added chapter 10 "Enforcing Architecture Boundaries". Added a paragraph in chapter 9, section "Assembling via Spring's Java Config" discussing the visibility of @Bean classes. Fixed an embarrassing typo in the very first sentence of the book.
0.4	2019-07-12	Added chapter 11 "Taking Shortcuts Consciously". Reduced the use of quotemarks for a better reading experience. Streamlined all occurrences of "input" and "output" ports / adapters into "incoming" and "outgoing" ports / adapters.
1.0	2019-08-19	Added chapter 7 "A Testing Strategy". Moved the testing chapter right after the implementation chapters, so that chapter numbers for the following chapters have changed. Added chapter 12 "Deciding on an Architecture Style". Replaced the example application from the book writing domain with an example application from the financial domain (everything can be explained better in terms of money). Updated most code examples and many figures to refer to the new example application.
1.1	2019-09-04	Added reference to SOLID principles in foot note 11. Added code example with `@Transactional` annotation in chapter 6. Removed paragraph referring to agile and legacy in chapter 1, as it might spark unproductive discussion. Fixed some typos and phrasing. Thanks to Ludwig Richter for the feedback.
1.2	2019-09-30	Updated link to primary source on Hexagonal Architecture, as it has been made available again Polished some phrases and code snippets
1.3	2019-11-08	Fixed some typos and errors in a figure (thanks to Petromir Dzhunev for the feedback). Added a note about the print version of the book in the preface.

Revision	Date	Changes
1.4	2020-01-30	Fixed an error in a code example and re-phrased "no mapping strategy should … " to "no single mapping strategy should …" to avoid confusion with the "no-mapping strategy" (thanks to Emil Gelev for the feedback).
1.5	2020-12-01	Changed package `buckpal.domain` to `buckpal.account.domain` in a couple of places. Fixed some typos.
1.6	2021-02-20	Changed some package names in the code examples to make the code easier to find in the example application oin GitHub.
1.7	2021-12-15	Fixed a logical error in the section "Different Input Models for Different Use Cases" in the chapter "Implementing a Use Case" (thanks to Soeun Park for the feedback).

Second edition

Revision	Date	Changes
2.0	2023-07-14	Second edition. New first chapter "Maintainability". New chapter about "Managing Multiple Bounded Contexts". New chapter "A Component-Based Approach to Software Architecture".