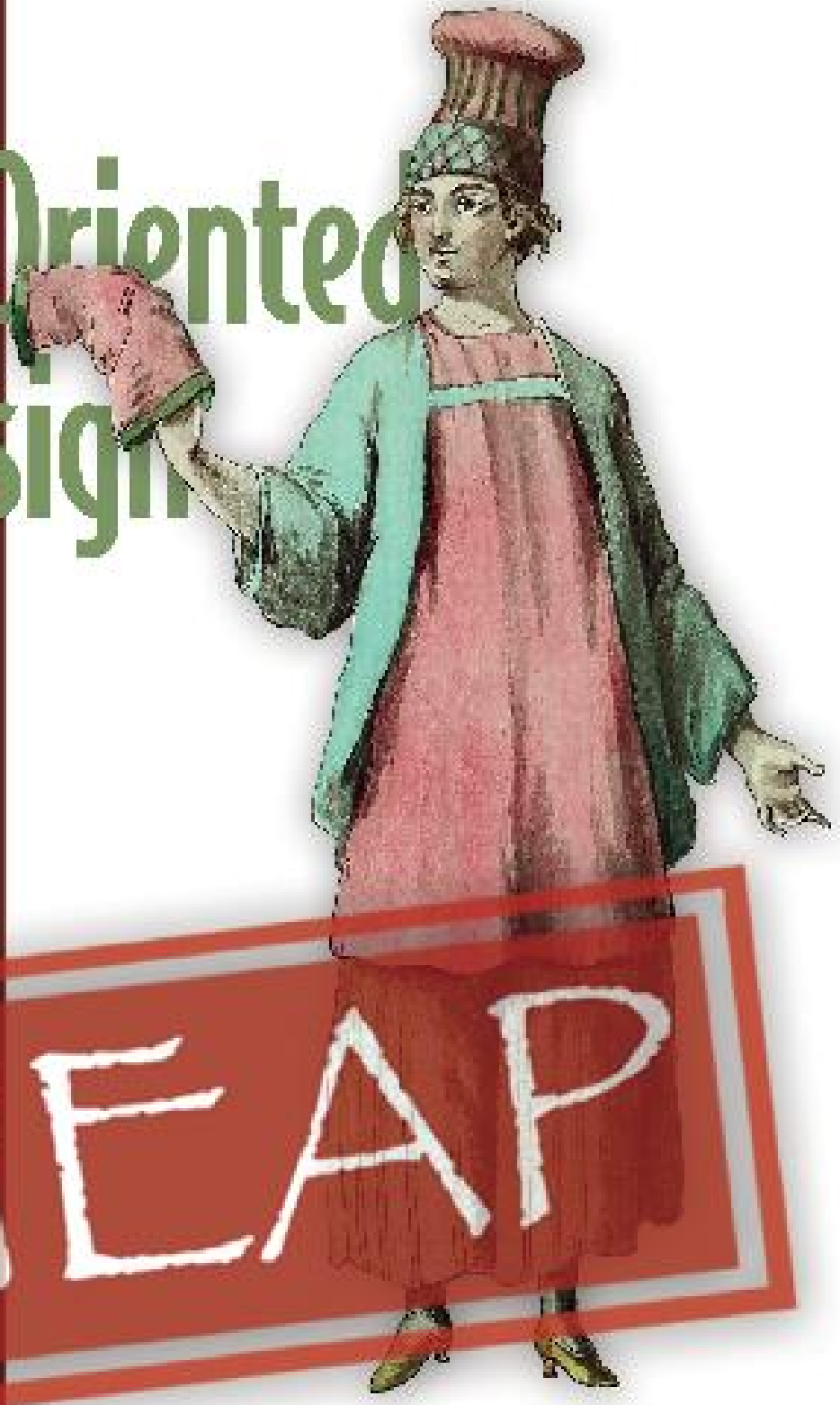
A guide to creating clean, maintainable code

SIMPLE Object Oriented Design



Maurício Aniche

MEAP

MANNING

Simple Object Oriented Design

1 It’s all about managing complexity

This chapter covers

- Why software systems get more complex over time
- The different challenges in object-oriented design
- Why we should keep improving our design over time

Information systems have been simplifying so much of our lives. Imagine going to the doctor and having them manually search for the patient’s history. Or going to any store and waiting for the cashier to sum up the price of the items we’re buying. Fortunately, this isn’t the case for a long time now. We use software systems to help us organize complex information and have them available at the snap of our fingers. We, software engineers, must develop such systems and ensure that they can keep evolving.

Building a highly maintainable information system requires good design. Modern businesses tend to be quite extensive, as companies often operate on different fronts and offer different products; to be complex, with business rules full of variations; and evolve fast, requiring developers to keep implementing or changing existing functionality.

Luckily, we don’t have to invent best practices for object-oriented systems from scratch, as our community already has extensive knowledge on the topic. Think of the Gang of Four’s *Design Patterns*, *SOLID principles*, *Clean Architecture* by Robert Martin, Martin Fowler’s refactoring techniques, *Hexagonal Architecture* by Alistair Cockburn, *Domain-Driven Design* by Eric Evans, object-oriented design styles by Matthias Noback, or at an even higher level of abstraction, service decomposition patterns by Sam Newman.

Throughout the years, I learned a lot from my good and bad decisions, especially when applying these principles to information systems. If you don’t know these principles, no worries. They aren’t a pre-requisite to read this book. But I recommend you read about them once you finish this book. If you already know these principles, this book will give you a different and more pragmatic view of them.

Applying any best practice or principle by the book is always difficult. Trade-offs have to be made. Where should I code this business rule? Is this code simple enough, or should I propose a more elegant abstraction? How should I model the interaction between my system and this external web app? Should I make this class depend on this other class, or is that bad coupling? These are all common questions that emerge in the minds of any developer that’s building an information system.

In this book, I’ll share with you my set of patterns that have been helping me deliver high-quality information systems.

Why is this book called "*Simple Object-Oriented Design*," you may ask? Because simple object-oriented designs are always easier to maintain. The main challenge is not to come up with a simple design but keep it this way. As I'll discuss throughout this book, too many forces push complexity down your throat. If you don't explicitly work toward keeping design simple, you'll end up in no time with a big ball of mud (http://www.laputan.org/mud/mud.html) that's hard to maintain.

1.1 Designing simple object-oriented systems

"As software systems evolve, their complexity increases unless work is done to maintain or reduce it." Evolving software systems of any type isn't that straightforward. We know that code tends to decay over time, requiring effort to maintain since the 1980s. This insight comes from Lehman's famous paper on the "laws of software evolution" (https://www.sciencedirect.com/science/article/abs/pii/0164121279900220). And despite 40 years of progress, the maintainability of software systems remains a challenge.

In essence, maintainability is the effort you need to complete tasks like modifying business rules, adding features, identifying bugs, and patching the system. Highly maintainable software enables developers to perform such tasks with reasonable effort, whereas low maintainability makes tasks too difficult, time-consuming, and bug-prone.

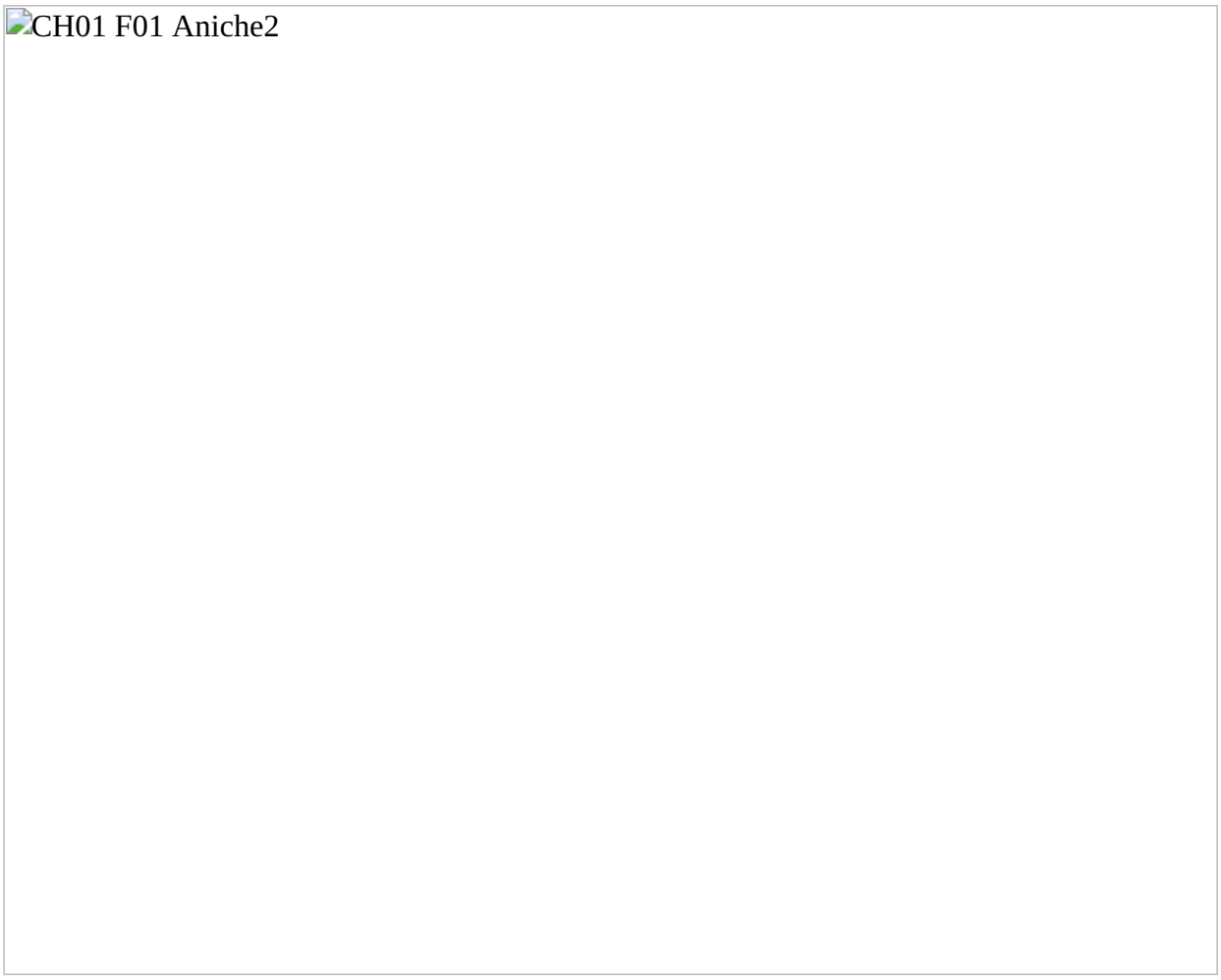
Many factors affect maintainability, from overly complex code to dependency management, poorly designed abstractions, and bad modularization. Systems naturally become more complex over time, so continually adding code without considering its consequences on maintenance can quickly lead to a messy codebase.

Consistently combatting complexity growth is crucial, even if it seems more time-consuming. And I know: it's much more effort than simply "dumping code." But trust me, developers feel way worse when handling big balls of mud the entire day. You may have worked on codebases that are hard to maintain. I did. Doing anything in such systems takes a lot of time. You can't find where to write your code, all code you write feels like it's a workaround, you can't write an automated test for it because the code is untestable, and you are always afraid something will go wrong because you never feel confident about changing it, and on it goes.

What constitutes a simple object-oriented design? Based on my experience, it's a design that presents the following six characteristics, also illustrated in figure 1.1:

- small units of code
- consistent objects
- proper dependency management
- good abstractions
- infrastructure properly handled
- well modularized

Figure 1.1 Characteristics of a simple object-oriented design



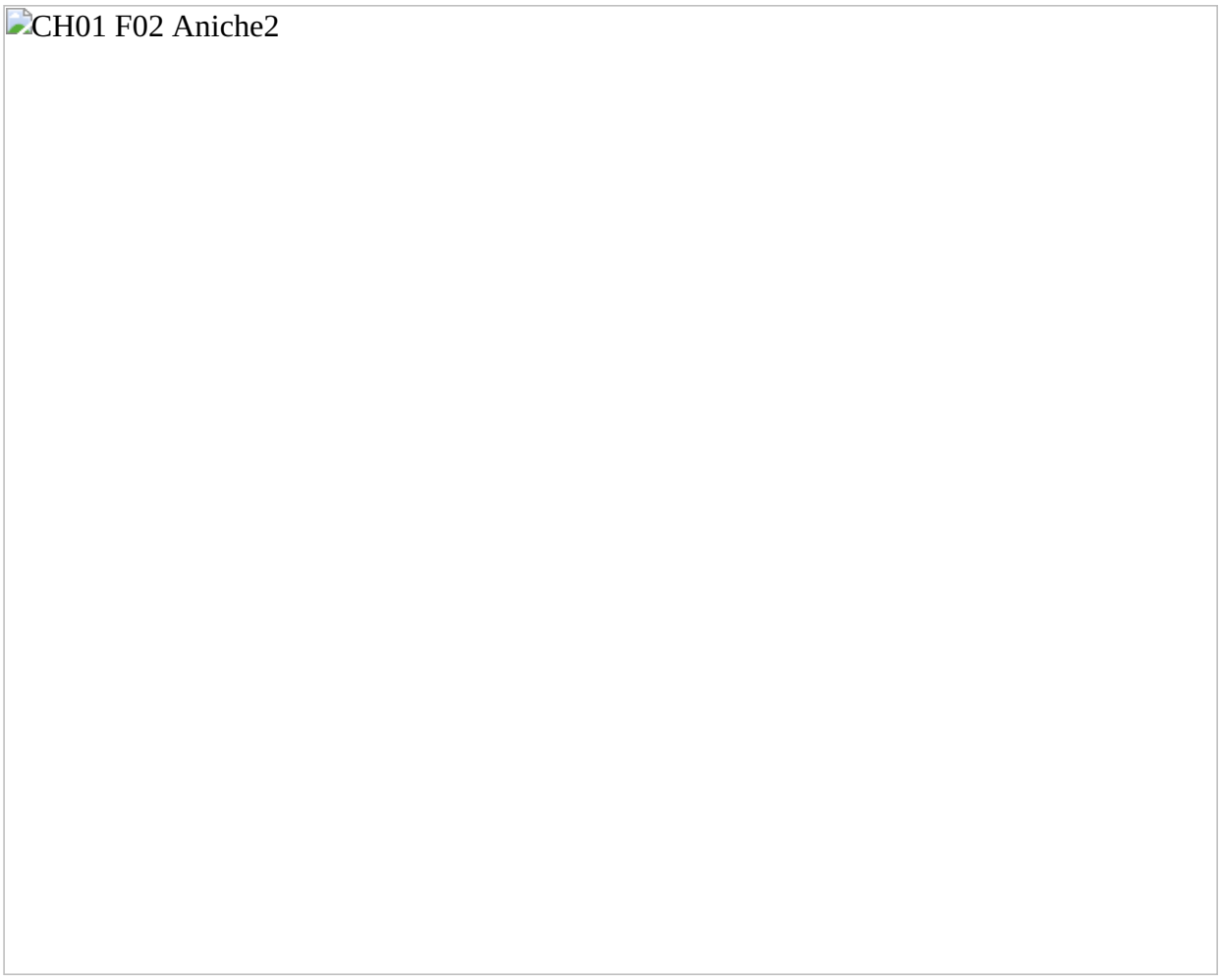
These ideas may sound familiar to you. They are all popular in object-oriented systems. Let’s look at what I mean by each of them and what happens when we lose control, all in a nutshell.

1.1.1 Small units of code

Keeping implementing methods and classes that are simple in essence is a great way to start your journey toward maintainable object-oriented design. Consider a method that began as a few lines with a few conditional statements but grew over time and currently has hundreds of lines and `if`s inside of `if`s. Maintaining such a method is just tricky.

Interestingly, classes and methods usually start simple and manageable. But if we don’t work to keep them like this, they become hard to understand and maintain, like in figure 1.2. Complex code tends to result in bugs, as they are drawn to complex implementations that are difficult to understand. Complex code is also challenging to maintain, refactor, and test, as developers fear breaking something and struggle to identify all possible test cases.

Figure 1.2 Simple code becomes complex over time and, consequently, very hard to maintain.



There are many ways to reduce the complexity of a class or method. For example, clear and expressive variable names help developers to better understand what's going on. However, what I'm going to argue in this part of the book is that the number one rule to keep classes and methods simple is to keep them small. A method shouldn't be too long. A class shouldn't have too many methods. Smaller units of code are always easier to be maintained and evolved.

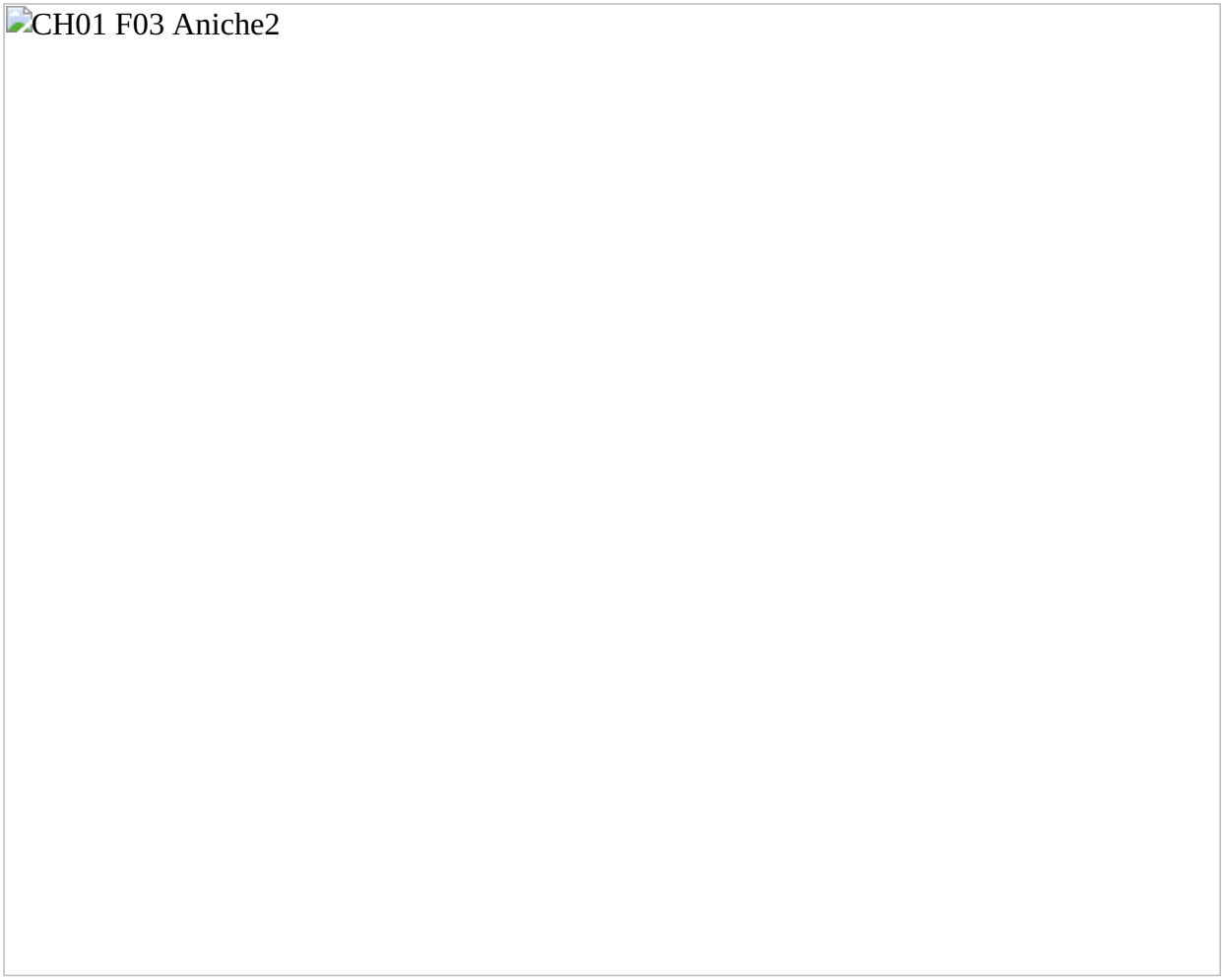
1.1.2 Consistent objects

It's much easier to work on a system where you can trust that objects are always in a consistent state and any attempt to make them inconsistent is denied. When consistency isn't accounted for in the design, objects may hold invalid states, leading to bugs and maintainability issues.

Consider a Basket class in an e-commerce system that tracks the products a person is buying and their final value. The total value must be updated whenever we add or remove a product from the basket. The basket should also reject invalid client requests, like adding a product -1 times or removing a product that isn't there.

In figure 1.3, the left side shows a protected basket, where items can only be added or removed by asking the basket itself. The basket is in complete control and ensures its consistency. On the right side, the unprotected basket allows unrestricted access to its contents. Given the lack of control, that basket can’t always ensure consistency.

Figure 1.3 Two baskets, one that has control over the actions that happen on it, another one that doesn’t. Managing state and consistency is fundamental.



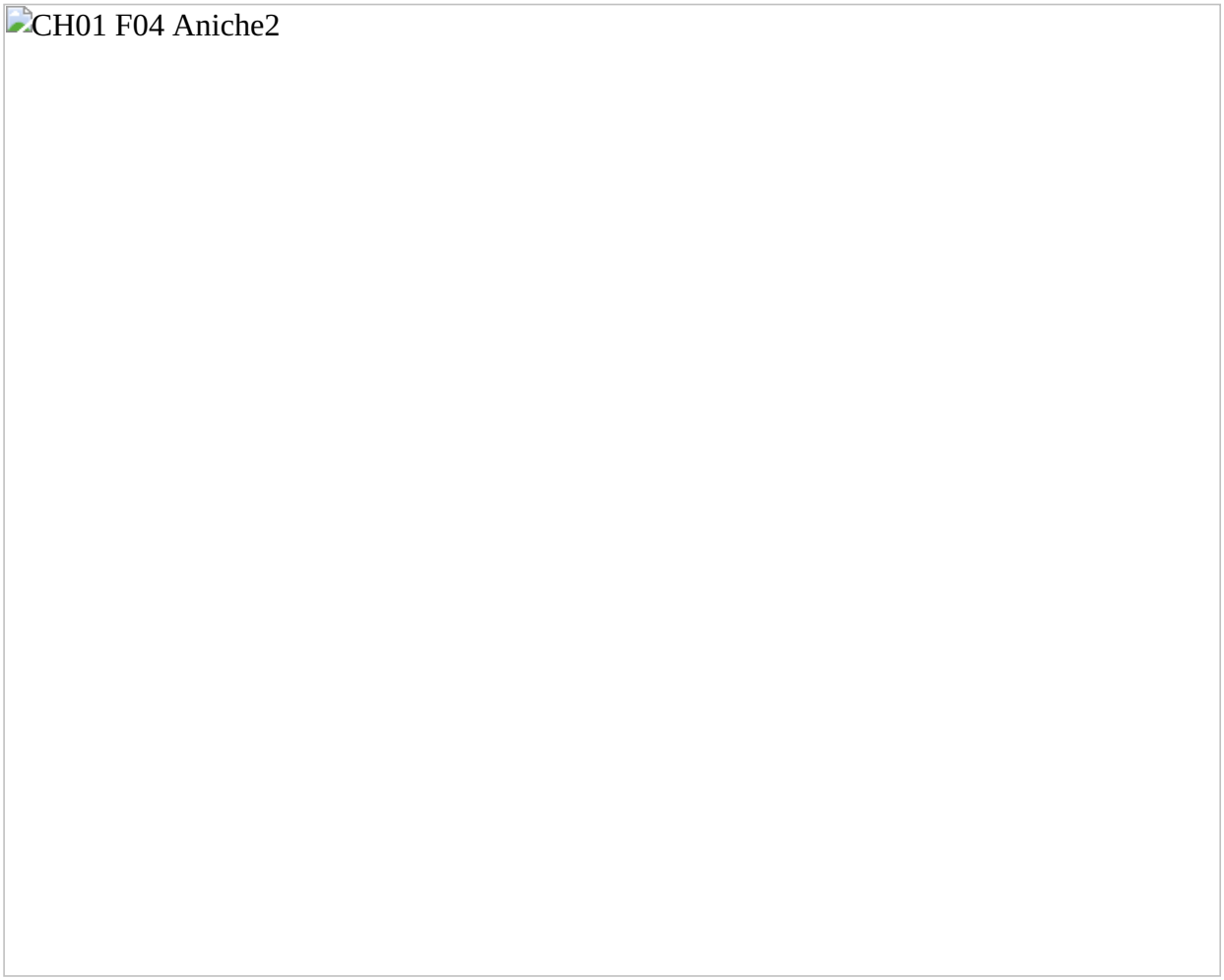
We’ll see that a good design ensures objects can’t ever be in an inconsistent state. Consistency mishandling can happen in many different ways, such as improper setter methods that bypass consistency checks or the lack of flexible validation mechanisms, which we’ll discuss in more detail later.

1.1.3 Proper dependency management

In large-scale object-oriented systems, dependency management becomes key to maintainability. In a system where the coupling is high and no one cares about "which classes are coupled to which classes," any simple change may have unpredicted consequences.

Figure 1.4 shows how the `Basket` class may be impacted by changes in any of its dependencies: `DiscountRules`, `Product`, `Customer`. Even a change in `DiscountRepository`, a transitive dependency, may impact `Basket`. If, say, class `Product` frequently changes, `Basket` is always at risk of having to change as well.

Figure 1.4 Dependency management and change propagation



Simple object-oriented designs aim to minimize dependencies among classes. The less they depend on each other and the less they know about each other, the better. Good dependency management also ensures that our classes depend as much as possible on stable components that are less likely to change and, therefore, less likely to provoke cascading changes.

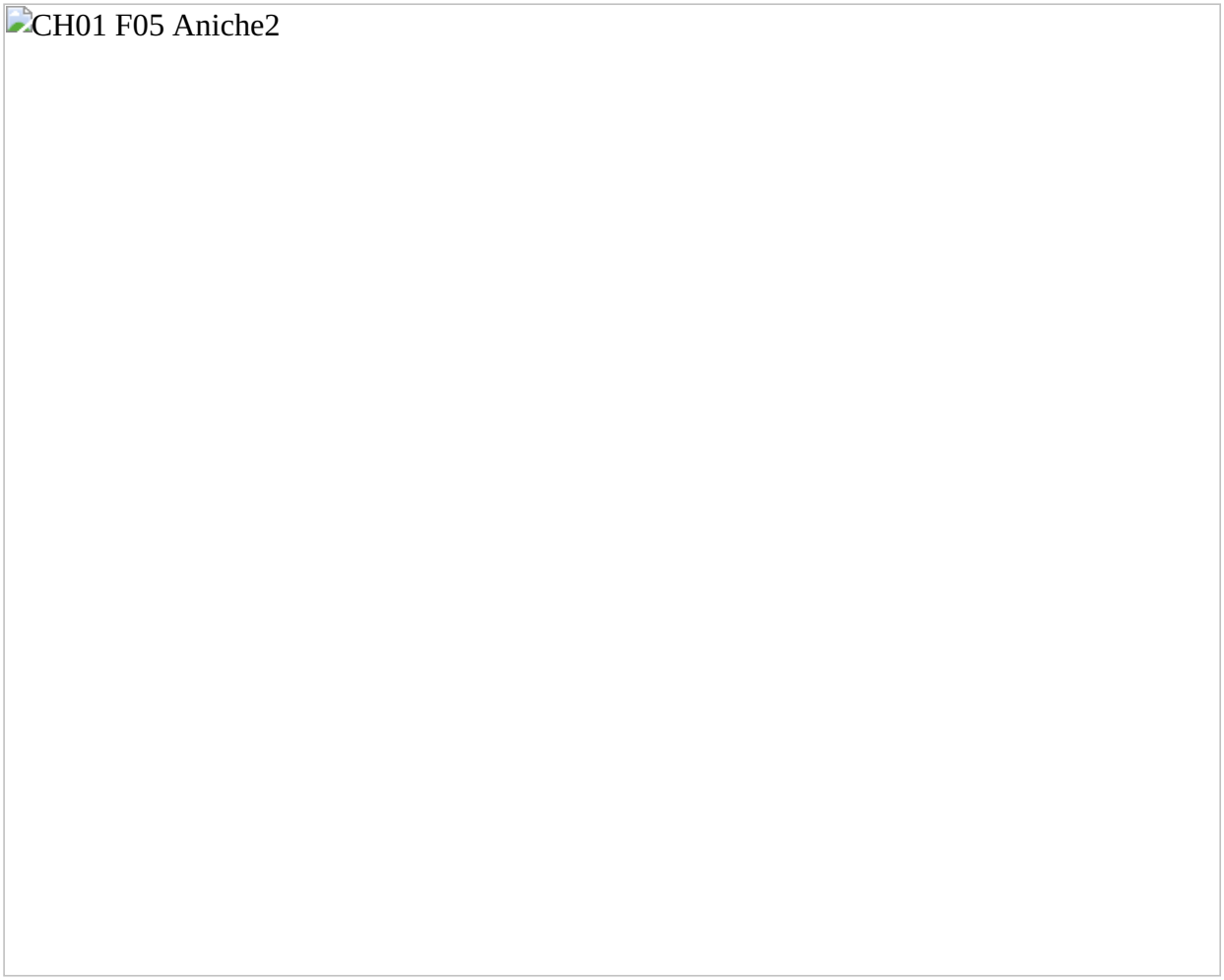
1.1.4 Good abstractions

Simple code is always preferred, but it may not be sufficient for extensibility. Extending a class by adding more code at some point stops being effective and becomes a burden.

Imagine implementing 30 or 40 different business rules in the same class or method. I illustrate that in figure 1.5. Note how the `DiscountRule` class, a class that's responsible for

applying different discounts in our e-commerce system, grows as new discount rules are introduced, making the class much harder to maintain.

Figure 1.5 A class that has no abstractions grows in complexity indefinitely.

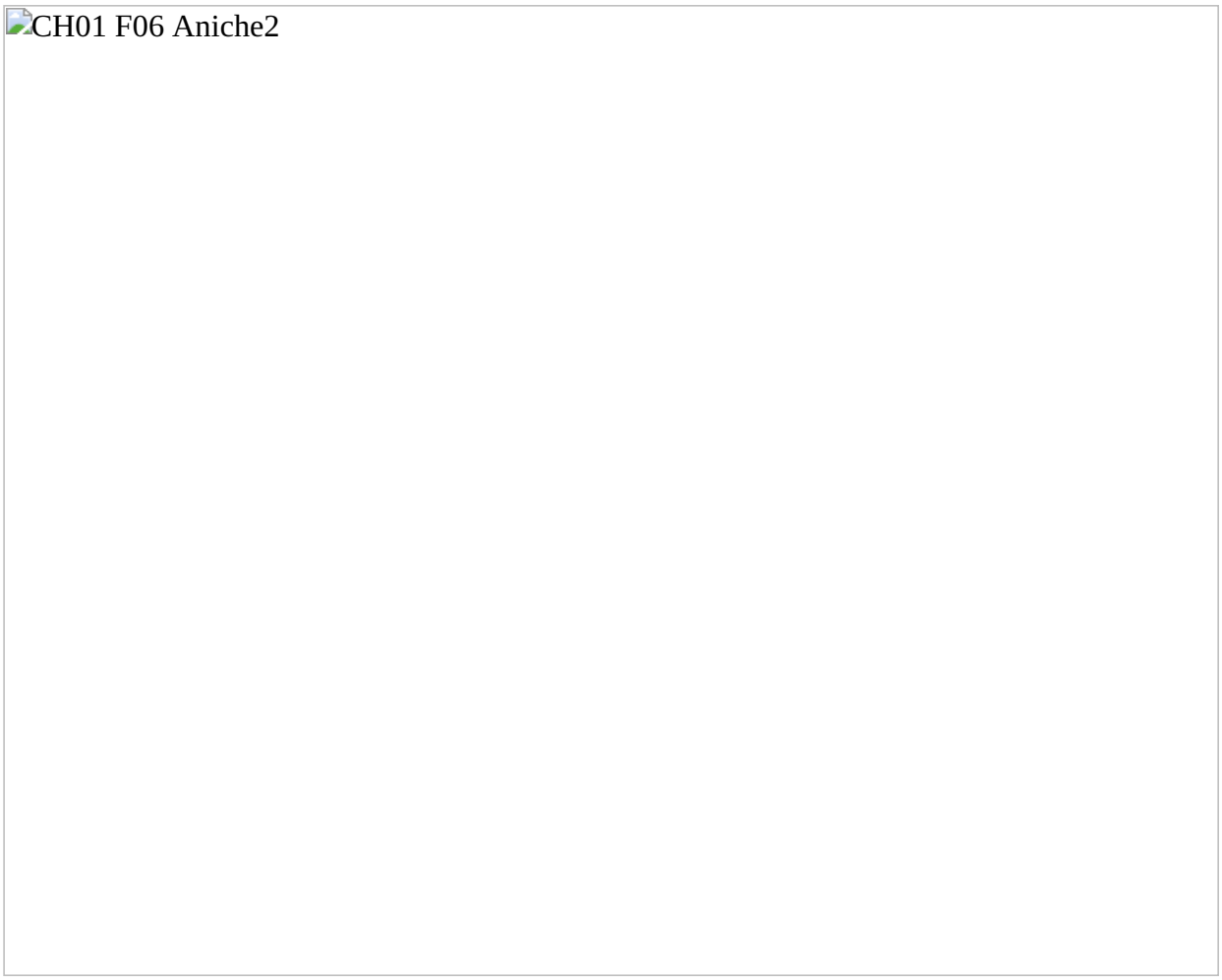


A good design provides developers with abstractions that help them evolve the system without making existing classes more complex.

1.1.5 Infrastructure properly handled

Simple object-oriented designs separate domain code, that contains business logic, from infrastructure code required for communication with external dependencies. Figure 1.6 shows domain classes on the left and infrastructure classes on the right.

Figure 1.6 The architecture of a software system that separates infrastructure from domain code.



Letting infrastructure details leak into your domain code may hinder your ability to make changes in the infrastructure. Imagine all the code to access the database is spread through the code base. Now, you decide you need to add a caching layer to speed up the application's response time. You may have to change the code everywhere for that to happen.

The challenge is to find the right abstraction for the infrastructure code. For instance, if you are using a relational database like Postgres, you may want to completely hide its presence from the domain code, but doing so could limit access to its unique features that enhance productivity or performance. The key is to abstract irrelevant aspects while leveraging valuable features that are provided by your infrastructure.

1.1.6 Well modularized

As software systems grow, fitting everything into a single component or module is challenging. Simple object-oriented designs divide large systems into independent components that interact to achieve a common goal.

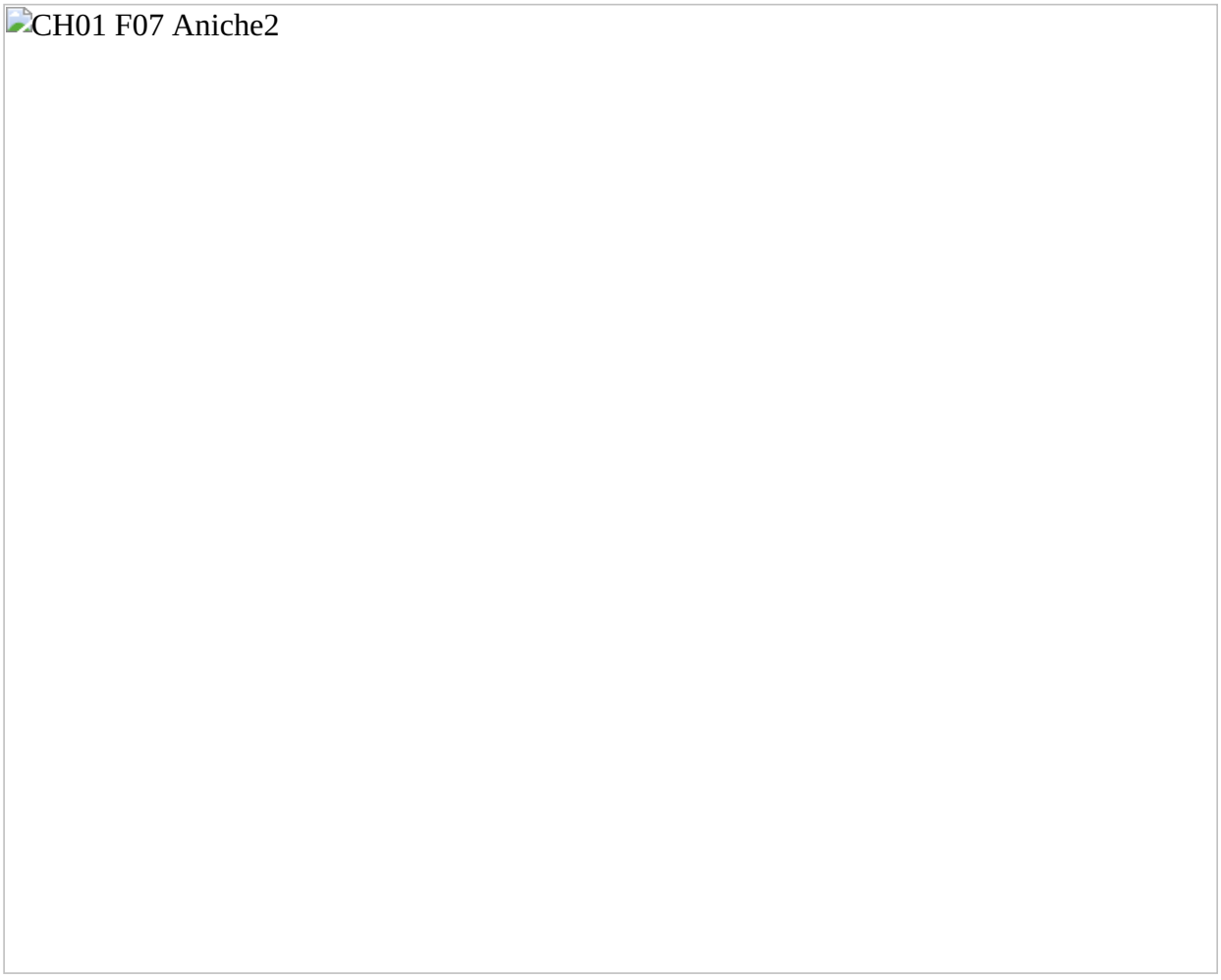
Dividing systems into smaller components makes them easier to maintain and understand. It also helps different teams work on separate components without conflicts. Smaller

components are more manageable and testable.

Consider a software system with three domains: Invoice, Billing, and Delivery. These domains must work together, with Invoice and Delivery requiring information from Billing.

Figure 1.7 shows a system without modules on the left, where classes from different domains mix freely. As complexity increases, this becomes unmanageable. The right side of the figure shows the same system divided into modules—Billing, Invoice, and Delivery. Modules interact through interfaces, ensuring clients only use what's needed without understanding the entire domain.

Figure 1.7 Two software systems with different modularization approaches.



It's not easy to identify the right level of granularity for a module or what its public interface should look like, which we'll talk more about later.

1.2 Simple design as a day-to-day activity

As I said before, the hard part is often not to design a simple design but to keep it that way. We must keep focusing on improving and simplifying our designs as we learn about the system and the more it evolves.

1.2.1 Reducing complexity is similar to personal hygiene

Constantly working towards simplifying the design can be compared to brushing your teeth. While not particularly exciting, it’s necessary to avoid discomfort and costly problems in the future. Similarly, investing a little time in code maintenance daily helps prevent more significant issues down the line.

1.2.2 Consistently addressing complexity is cost-effective

Regularly addressing complexity keeps both the time and cost associated with it within reasonable limits. Delaying complexity management can result in significantly higher expenses and make refactoring more difficult and time-consuming.

The "technical debt" metaphor helps us understand this. Coined by Ward Cunningham, the idea is to see coding issues as financial debt. The additional effort required for maintenance due to past poor decisions represents interest on this debt. This concept relates closely to the book’s focus; complexity escalates if code structure isn’t improved, leading to excessive interest payments.

I’ve been to code bases where everyone knew that parts of it were very complex and hard to maintain, but no one dared to refactor it. Trust me, you don’t want to get there.

1.2.3 High-quality code promotes good practices

When developers work with well-structured code with proper abstractions, straightforward methods, and comprehensive tests, they are more likely to maintain its quality. Conversely, messy code often leads to further disorganization and degradation in quality. This concept is similar to the broken-window theory, which explains how maintaining orderly environments can prevent further disorder.

1.2.4 Controlling complexity isn’t as difficult as it seems

The key to ensuring the complexity doesn’t grow out of hand is recognizing patterns and addressing them early on. With experience and knowledge, developers can detect and resolve most issues in the initial stages of development.

1.2.5 Keep the design simple is a developer’s responsibility

Creating high-quality software systems that are easy to evolve and maintain can be challenging but necessary. As developers, managing complexity is part of our job and contributes to more efficient and sustainable software systems.

Striking the right balance between manageable complexity and overwhelming chaos is challenging. Starting with complex abstraction might prevent issues, but it adds system complexity. A simpler method with two `if` statements is easier to understand than a convoluted interface. On the other hand, at some point, simple code isn't enough anymore. Striving for extensibility in every code piece would create chaos.

It's our job to find the right balance between simplicity and complexity.

1.2.6 Good enough designs

John Ousterhout, in "A Philosophy of Software Design" (https://web.stanford.edu/~ouster/cgi-bin/book.php), says that it takes him at least three rewrites to get to the best design for a given problem. I couldn't agree more.

Often, the most effective designs emerge after several iterations. In many cases, it's more practical to focus on creating "good enough designs" that are easily understood, maintained and evolved rather than striving for perfection from the outset. Again, the key is to identify when simple isn't enough anymore.

1.3 A short dive into the architecture of an information system

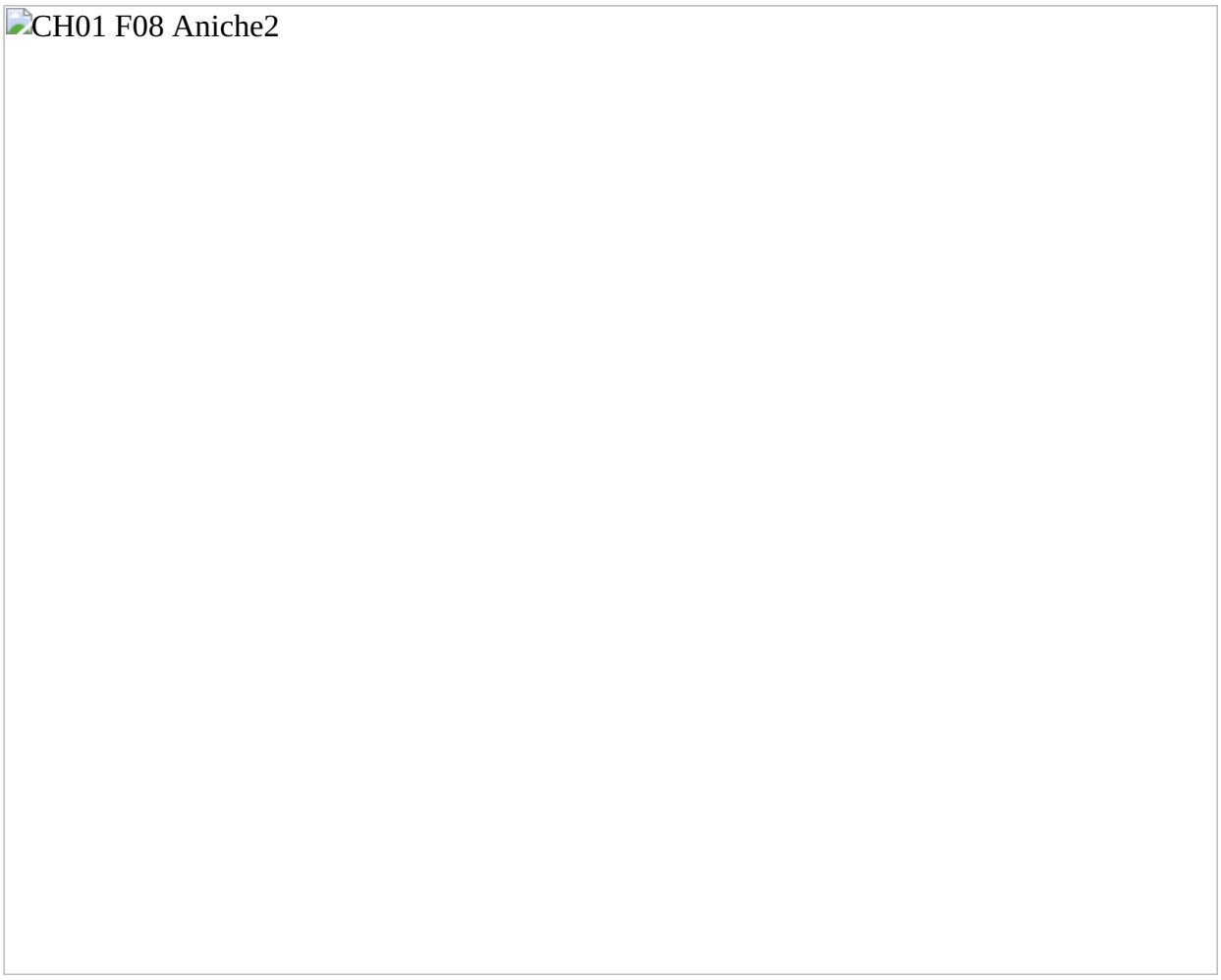
Before discussing the different patterns to help you keep your design simple, let me define a few terms.

First, while I started this chapter by talking about information systems, I'll focus on the back-end part of such systems. I'm using the term back-end to identify the software that runs behind the scenes of every information system.

Figure 1.8 illustrates a traditional information system. Back-end systems are often characterized by the following:

- Having a front-end that displays all the information to the user. This front-end is often implemented as a web page. Developers can use many technologies to build modern front-ends, such as React, Angular, VueJS, or even plain vanilla JavaScript, CSS, and HTML.
- Having a back-end that handles the requests from the front-end. The back-end is the place where most, if not all, business logic lives. The back-end may communicate with other software systems to achieve its tasks, such as external or internal web services.
- Having a database that stores all the information. Back-end systems are strongly database-centric. This means most back-end actions involve retrieving, inserting, updating, or deleting information from the database.

Figure 1.8 A traditional information system with front-end, back-end, and database.



Let me also define the inside of a back-end system, so we can agree on a few terms. See figure 1.9. A back-end system receives requests via any protocol, but usually HTTP, from any other system, for example, a traditional web-based front-end.

This request is first received by a Controller. The Controller's primary responsibility is to convert the user request into a series of commands to the domain model that knows the business rules.

The domain model is composed of different types of classes. This depends on the architectural patterns your application is following, but you often see:

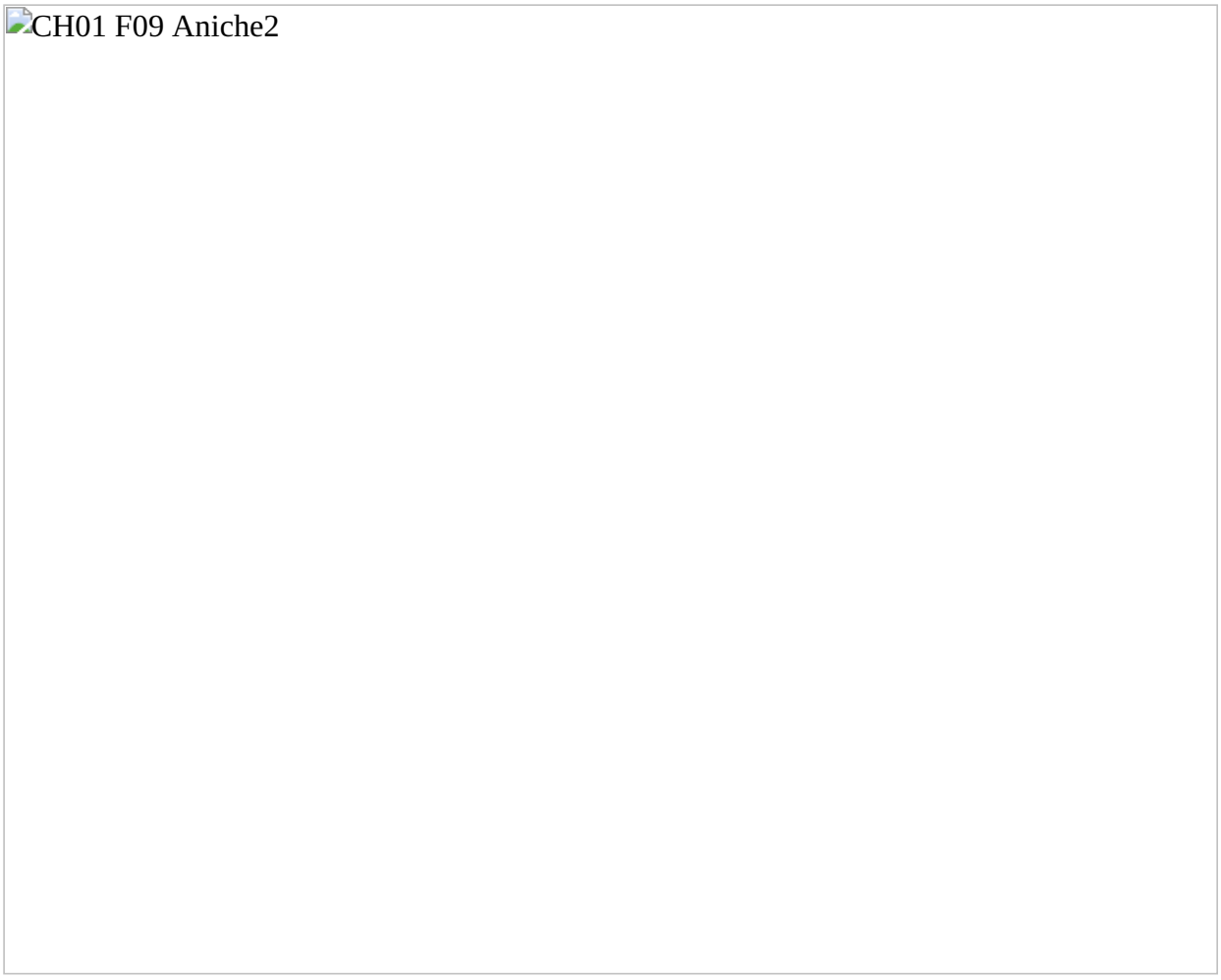
- Entities, that model the business concepts. Think of an `Invoice` class that models what invoices mean to the system. Entity classes contain attributes that describe the concept and methods that consistently manipulate these attributes.
- Services, that encapsulate more complex business rules that involve one or more entities. Think of a `GenerateInvoice` service responsible for generating the final invoice for a user that just decided to pay for all the products in their basket.
- Repositories, which contain all the logic to retrieve and persist information. Behind the scenes, their implementation talks to a database.

- Data Transfer Objects (DTOs), that are classes that hold some information and are used to transfer information from different layers.
- Utility classes, which contain a set of utility methods that are not offered by your programming language or framework of choice.

Back-ends also commonly have to communicate with other external applications, usually via remote calls or specific protocols. Think of a web service that enables the application to send a request to a governmental system. Or to an SMTP server that enables the application to send e-mails. Anything that's outside and somewhat out of control of the back-end, I'll call infrastructure.

A large-scale back-end may also be organized in modules. Each module contains its own domain model with entities, repositories, services, etc. Modules may also send messages to each other.

Figure 1.9 The internal design of a back-end system.



If you're familiar with architectural patterns such as Clean Architecture, Hexagonal Architecture, Domain-Driven Design, or any other layered architecture, you may have

opinions on how things should be organized inside. This diagram is meant to be generic enough so that anyone can fit it into their favorite architecture.

The figure also shows the two modules inside the same back-end, which may lead you to think I'm proposing monoliths over a service-distributed architecture. Again, this figure is meant to be generic. Different modules can live in the same distributed binary and communicate via simple method calls or distributed over the network and communicate via remote procedure calls. It doesn't matter at this point.

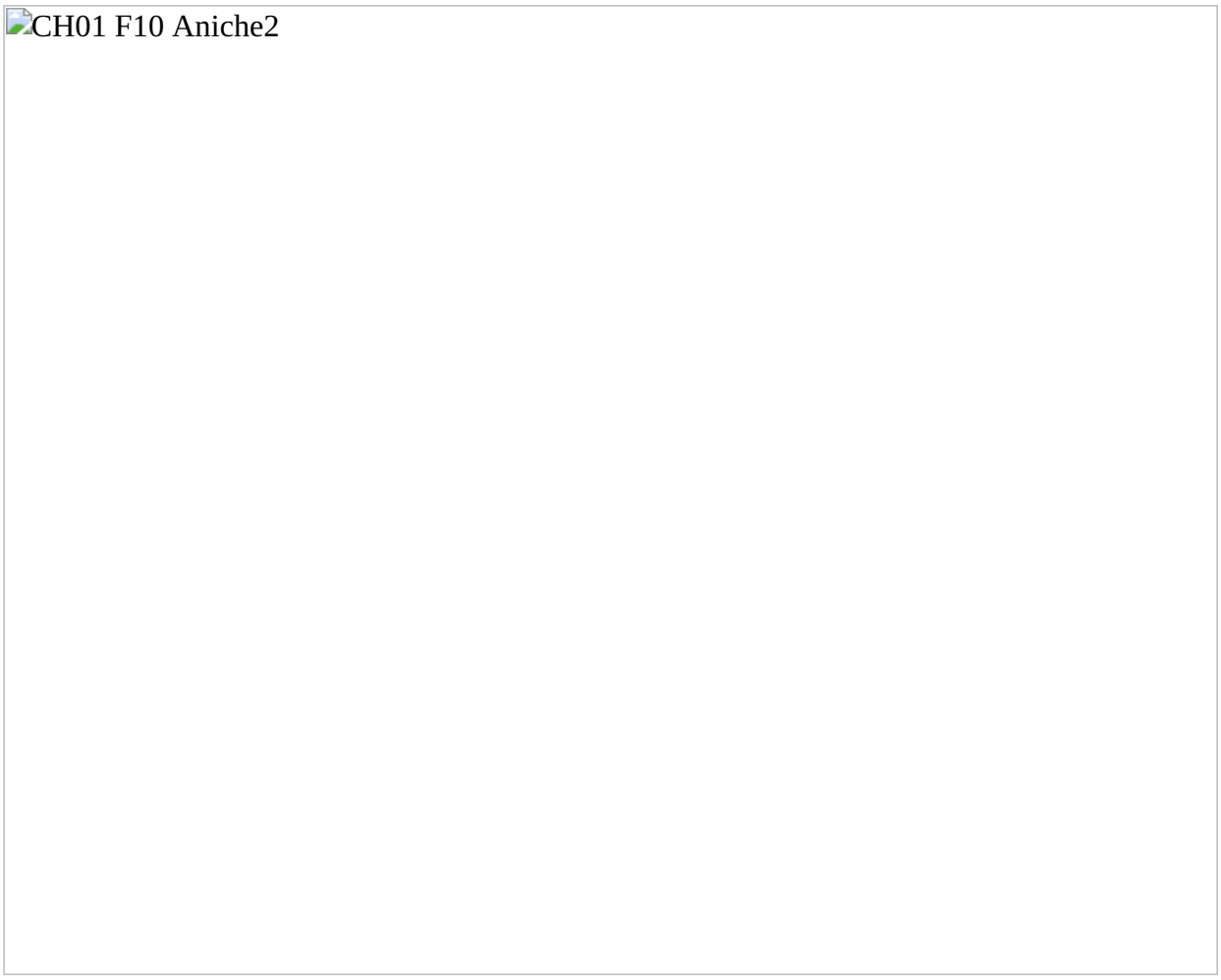
1.4 The example project: PeopleGrow!

To illustrate the design patterns throughout the book, I'll use an imaginary back-end system called PeopleGrow!, illustrated in figure 1.10, an information system that manages employees and their growth through trainings.

The system handles different features. I highlight a few of them below. In bold, domain terms that will often appear in the following chapters:

- The list of **trainings** and **learning paths** that are collections of trainings.
- The **employees** and the trainings that they took or still have to take.
- Trainings are offered multiple times a year. Each **offering** contains the training date and the maximum number of participants allowed.
- Participants can **enroll** for the offerings themselves or be enrolled by the administrator.
- The trainings and the **trainers** that deliver these trainings.
- All sorts of **reports**, such as which trainings miss instructors, which trainings are full, etc.
- All sorts of convenient functionality, such as calendar invites through the company's calendar system, automatic messages via the company's internal chat, and e-mail notifications.
- A front-end available for **administrators** to add new trainings, and learning paths, and see the reports.
- Various APIs available for any internal system to use. For example, employees can enroll in trainings via the company's internal wiki, which uses PeopleGrow!'s APIs.

Figure 1.10 The high-level architecture of PeopleGrow!



Architecturally speaking, PeopleGrow! comprises a front-end, a back-end, and a database. As I said, it also can connect to external systems of the company, such as the internal chat and the calendar system.

The back-end is implemented using object-oriented languages like Java, C#, or Python. It makes use of current frameworks for web development and database access. For example, think of Spring Boot and JPA, if you are a Java developer, ASP.Net MVC / Core and Entity Framework if you are a C# developer, or Django if you are a Python developer. Internally, the back-end models the business. Imagine classes such as `Employee`, `Trainer`, `Enrollment`, and `LearningPath` in the codebase.

The team that's building PeopleGrow! is now facing maintenance challenges. Bugs are emerging. Whatever change is requested by the product team takes days to be done. Developers are always afraid of making changes, and an innocent change often impacts areas of the system one wouldn't expect.

Let's dive into the design decisions of PeopleGrow! and improve them!

1.5 Exercises

Discuss the following questions with a colleague:

1.1. In your opinion, what constitutes a simple object-oriented design? What's the difference between your point of view and the point of view presented in this chapter?

1.2. What types of object-oriented design problems have you faced in your life as a developer? What were their consequences? Do they fit in any of the six categories presented in this chapter?

1.3. Do you think it's possible to keep the design always simple as the system evolves? What are the main challenges in keeping it simple?

1.6 Summary

- Building a highly maintainable software system requires a good object-oriented design. Simplicity is the key factor for a highly maintainable software system.
- Building simple object-oriented designs is often simple, but it's hard to keep the design simple as the complexity of the business grows. Managing complexity is essential to maintain and develop software systems effectively.
- Simple and maintainable object-oriented designs have six characteristics: simple code, consistent objects, proper dependency management, good abstractions, infrastructure adequately handled, and well modularized.
- Managing complexity and keeping designs simple is a continuous process that requires daily attention, like brushing your teeth.
- Writing good code is easier when the codebase is already good. Productivity increases, developers feel confident modifying the code, and business value is delivered faster.

2 Making code small

This chapter covers

- Breaking large units of code into smaller pieces
- Moving new complexity away from existing units of code
- Documenting your code to improve understanding

Making code simple and small is the first thing you should always consider. Even with a well-designed software system, losing sight of code complexity is easy. Lines of code tend to overgrow if we don't actively work to prevent it, leading to oversized classes. Long classes and methods happen more often and more naturally than we like to admit. It's just too easy to open an existing class in the system and add more code instead of reflecting on the impact of these new lines and re-design the code.

Uncontrolled complexity hinders code evolution. Complex code is harder to read and understand than simple code. As a developer, you've likely experienced the mental strain of deciphering a 200-line method compared to a 20-line one. You feel tired even before starting to read it. It's also known that highly complex code is more prone to bugs. It's easier to make mistakes in code that's difficult to grasp. It's also hard to write tests for complex code, as it has too many different cases and corner cases to be explored, and it's just too easy to forget one of them. And we can't blame the developer. It's hard to develop good test cases for complex code.

The gist of this chapter is that code that's small is easier to be maintained. In the next sections, I'll dive into patterns that'll help you make your code small.

2.1 Make units of code small

Classes and methods should be small. That improves code readability, maintainability, reusability, and reduces the likelihood of bugs.

Business rules in information systems keep evolving and increasing in complexity. The popular quick and dirty way to evolve a business rule is to add more code to existing methods and classes. Either adding lines of code to an existing method or adding methods to existing classes. Suddenly, you have a long class that requires a lot of energy from any engineer to maintain it.

No matter how well-crafted a 2000-line method or class is, it remains challenging to understand. Such methods do too much for any developer, regardless of seniority, to quickly grasp. This may seem like a basic pattern, but trust me, the primary strategy for reducing code complexity is to decrease the unit's size—-it's that simple.

Developers sometimes debate whether having more classes is a drawback. Some argue that it's easier to follow code if they are all on the same file. While software design has trade-offs, academic studies like the 2012 paper "An exploratory study of the impact of antipatterns on class change- and fault-proneness", by Khomh and colleagues, show that longer methods are more susceptible to changes and defects.

It takes little to convince someone that breaking complex code into smaller units is a good approach. Smaller units are always better than oversized units.

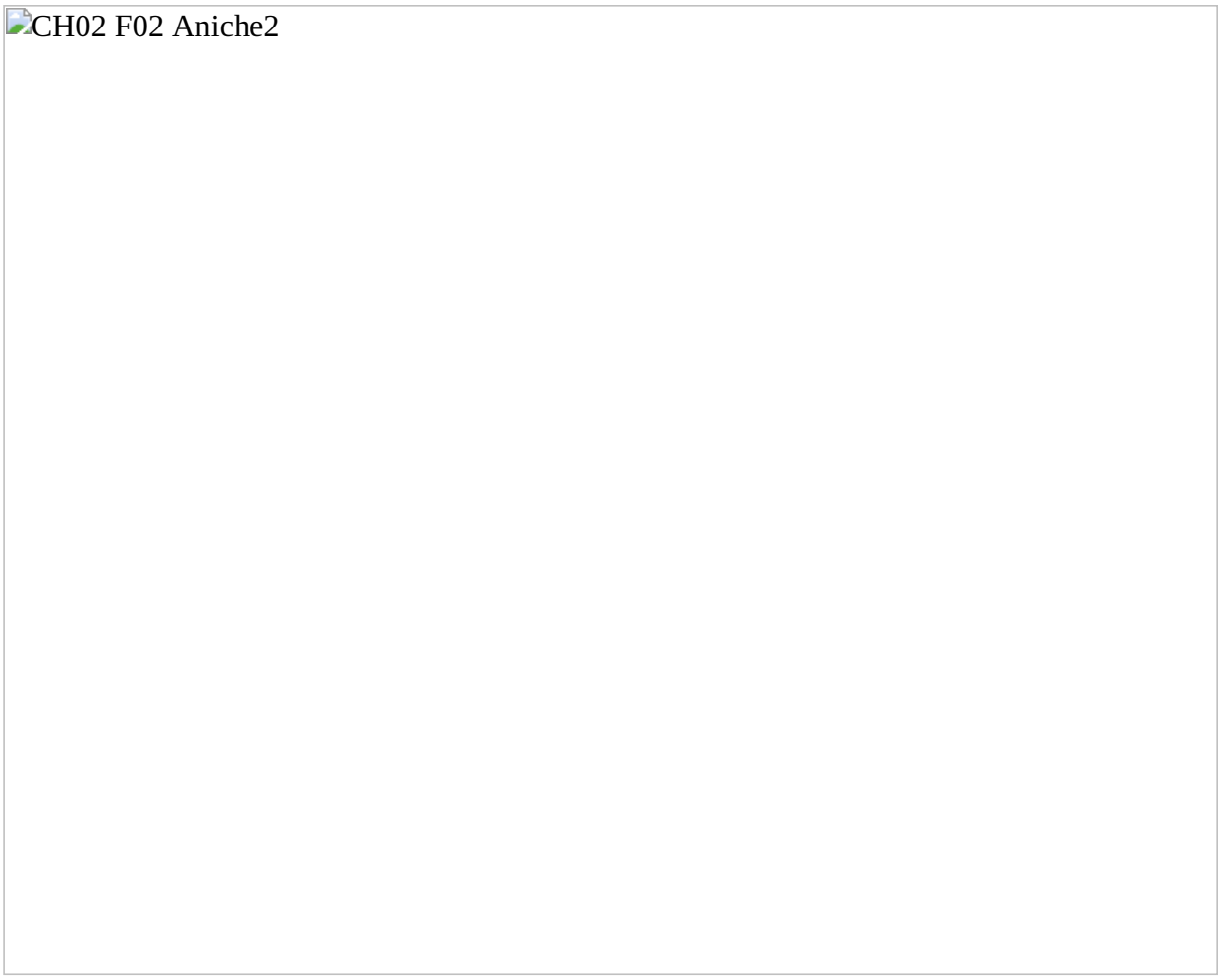
First, small classes or units of code allow developers to read less. If the whole implementation is visible, they will likely read it, whether necessary or not. But with a method call to another class, they'll only open that code when needed. Although one may argue that navigation becomes trickier when jumping between classes, modern IDEs make navigation easy once mastered.

Second, it enables extensibility from day one. Refactoring complex behavior into smaller classes often involves seeing the code as a set of pieces that, all together, compose a puzzle. Once modeled, each piece can be replaced if needed.

Lastly, testability improves: having smaller classes lets developers decide whether to write entirely isolated unit tests for specific parts of the business logic. Sometimes we want to exercise one piece in isolation; other times, all pieces together. You don't have that choice if the behavior is all in one place.

In practice, we should build complex behavior through the composition of smaller methods or classes, as illustrated in figure 2.1. Classes and methods there are all small and do one thing only. Anyone can easily understand, and testing is easy.

Figure 2.1 Smaller units are always better than large units



In the following sub-sections, I'll discuss a few heuristics of when to break up code into methods and classes. We'll also discuss the exceptional cases where you don't want to break code into small units.

What does cohesion mean?

A cohesive component (class or method) has a single, clear responsibility within the system —-it does one thing only. A class that does one thing is undoubtedly more petite than a class that does multiple things. If we strive for cohesive code, we naturally strive for simple code.

2.1.1 Break the complex method into private methods

Breaking a large method into a few smaller ones is an excellent and easy way to reduce complexity. All you need to do is to identify a piece of code within the large method that can be moved to a private method.

Private methods

Private methods can only be called from within the same it's declared and are a perfect solution for when you want to isolate a piece of code from the rest, but you don't want it to be visible and possibly called from the outside of the class.

An excellent way to determine if a new private method makes sense or if the code segment can be an independent unit is by evaluating the following:

- Can you assign a clear name to this private method that explains its purpose?
- Does this new method perform a cohesive, small action that the public method can easily use?
- Is this new method dependent on numerous parameters or class dependencies, or is it concise enough for a developer to understand its requirements quickly?
- When the method is called, is its name sufficient to explain its function without examining the implementation?
- Could this private method be made static? Such methods often make good candidates for extraction, as they don't rely on the original class. I don't want you to make the method static, but this is a nice trick to see if the method is independent.

2.1.2 Move the complex unit of code to another class

Private methods might not be the ideal location for extracted code, mainly if it's unrelated to the main goal of the large unit.

When should we move code to another class instead of a private method?

- Does this piece of code do something different from the rest of the class?
- Does it do something important enough for the domain that it deserves its own name and class?
- Do you want to test this piece of code in isolation?
- Does this code depend on classes you don't want the rest of the code to depend on?
- Is it too big that you must break this method into too many other private methods?

Believe me or not, one of the most significant challenges in moving behavior to a new class is to name this new class. If you can quickly come up with a good name and know precisely in which package this class should live, then you should do it.

2.1.3 When not to divide code into small units?

Every rule indeed has its exceptions. When should code be kept together?

- When two or more puzzle pieces can't live entirely apart. Forcing separation often results in complex method signatures.
- When a puzzle piece is unlikely to be replaced.
- When there's little value in testing a part in complete isolation.
- When there are few puzzle pieces. If you only need two to four private methods, why complicate it?

Pragmatism is critical, as always.

Careful with "classitis"

In the insightful book "A Philosophy of Software Design," John Ousterhout argues that having too many small classes can hinder maintainability. He refers to creating too many small classes as "classitis."

He has a point. Again, you don’t want to have micro classes as much as you don’t want to have huge blocks of code.

2.1.4 Get a helicopter view of the refactoring before you do it

In more complex refactorings, I try to picture what the final code look like after I’m done. What will the classes look like and how do they relate to each other? Do I like what I see? Do I see any design issues with it?

I don’t do this in any formal way. If I can’t see the final result in my mind, I resort to drawing diagrams in a piece of paper or whiteboard, usually in a UML-like form.

2.1.5 Example: Importing employees

Employees are imported into PeopleGrow! in batches. The administrator uploads a CSV (comma-separated values) file containing the employee’s name, e-mail, role, and starting date. If the employee is already in the database, PeopleGrow! updates their information.

The initial implementation looked like what you see in listing 2.1. The code parses the CSV using a third-party library, and then for each employee in the import data, the system either creates a new employee or updates the existing one in the database.

Although this code isn’t complicated, remember that this is just an illustration. The importing service could have hundreds of lines in a real software system.

Listing 2.1 A large method that should be broken down

```
class ImportEmployeesService {

  private EmployeeRepository employees;

  public ImportEmployeesService(EmployeeRepository employees) {
    this.employees = employees;
  }

  public ImportResult import(String csv) {

    var result = new ImportResult();

    var csvParser = new CsvParserLibrary();
    csvParser.setMode(IGNORE_ERRORS);
    csvParser.setObjectType(EmployeeParsedData.class);
```

```
    List<EmployeeParsedData> importedList = csvParser.parse(csv); #1

    for(var employee in importedList) {
      var maybeAnEmployee = employees.findByEmail(employee.email()); #2
      if(maybeAnEmployee.isEmpty()) { #3
        var newEmployee = new Employee(
          employee.name(),
          employee.email(),
          employee.startingDate(),
          employee.role());

        employees.save(newEmployee);
        result.addedNewEmployee(newEmployee);

      } else { #4
        var currentEmployee = maybeAnEmployee.get();
        currentEmployee.setName(name);
        currentEmployee.setStartingDate(startingDate);
        currentEmployee.setRole(role);

        employees.update(currentEmployee);
        result.updatedEmployee(currentEmployee);
      }
    }

    return result;
  }
}

record EmployeeParsedData(String name, String email,
  LocalDate startingDate, String role) { } #5
```

The `import` method does too much. It's too easy to get lost in the code. Let's reduce its complexity by simply moving code away from it. First, let's move the CSV parsing logic to another class. Even though these few lines of code only delegate the real work to the `CsvParserLibrary` class, it's a different responsibility and will look lovely in its own class.

Note

In future chapters, I'll also discuss that wrapping up calls to third-party libraries is in general a good idea as well.

See listing 2.2. The `EmployeeImportCSVParser` class offers a `parse` method that returns a list of `EmployeeParsedData`. The implementation is the same as before.

Listing 2.2 CSV parser in its own class

```
class EmployeeImportCSVParser {

  public List<EmployeeParsedData> parse(String csv) { #1
    var csvParser = new CsvParserLibrary();
    csvParser.setMode(IGNORE_ERRORS);
    csvParser.setObjectType(EmployeeParsedData.class);
    return csvParser.parse(csv);
  }
```

```
}
```

There's still more we can do back in the `ImportEmployeesService`. We can make the `import` method to only control the flow and let other classes or methods implement the actions. For example, the two blocks of code in the `if` statement, one that creates a new employee and the other that updates them, can be extracted to a private method.

I don't see these two methods going to different classes. They seem related, and leaving them in the `ImportEmployeesService` looks fine for now. I may change my mind in the future, but I like to take simpler and smaller steps first.

See listing 2.3. Look how the class is much smaller and the methods are more cohesive. The `import` method only coordinates the task. It calls the new `EmployeeImportCSVParser`, gets the parsed results, calls the `EmployeeRepository` and sees if the employee is already in the database, and based on that, decides which action to take, with each action in a separate private method.

Listing 2.3 The ImportEmployeesService is much smaller now

```
class ImportEmployeesService {

  private EmployeeRepository employees;
  private EmployeeImportCSVParser parser;

  public ImportEmployeesService(EmployeeRepository employees,
    EmployeeImportCSVParser parser) {
    this.employees = employees;
    this.parser = parser;
  }

  public ImportResult import(String csv) {

    var result = new ImportResult();

    var importedEmployees = parser.parse(csv); #1

    for(var importedEmployee : importedEmployees) {
      var maybeAnEmployee =
employees.findByEmail(importedEmployee.getEmail()); #2

      if(maybeAnEmployee.isEmpty()) { #3
        createNewEmployee(importedEmployee, result);
      } else {
        updateEmployee(importedEmployee, maybeAnEmployee.get(), result);
      }
    }

    return result;
  }

  private void createNewEmployee(
    EmployeeParsedData importedEmployee,
    ImportResult result) { #4
```

```
    var newEmployee = new Employee(
      importedEmployee.getName(),
      importedEmployee.getEmail(),
      importedEmployee.getStartingDate(),
      importedEmployee.getRole());

    employees.save(newEmployee);
    result.addedNewEmployee(newEmployee);
  }

  private void updateEmployee(
    EmployeeParsedData importedEmployee,
    Employee currentEmployee,
    ImportResult result) { #5
    currentEmployee.setName(name);
    currentEmployee.setStartingDate(startingDate);
    currentEmployee.setRole(role);

    employees.update(currentEmployee);
    result.updatedEmployee(currentEmployee);
  }
}
```

It takes less time for any developer to read this class and figure out what it does. It also takes less time for any developer to understand each small block.

I dislike that I have to pass `ImportResult` to the private methods. Not bad but also not elegant in my eyes. I'd consider making `result` a field of the class so that all private methods could access it. For that to happen safely, I'd have to ensure that instances of `ImportEmployeesService` are never reused. This can be achieved if you are using a dependency injection framework.

Having methods that focus on the "what" and letting other methods implement the "how" is a good practice, which we'll revisit in chapter 5.

Good job, `ImportEmployeesService` is much better now!

2.2 Make code readable and documented

Improve the readability of the code and document it when necessary to minimize the time developers spend trying to understand its purpose and functionality.

Consider how much time you've spent reading code to fix bugs or implement new features in unfamiliar areas. In the Clean Code book, Robert Martin estimates the ratio of time spent reading to writing code is about 10 to 1. Academic research indicates programmers spend around 60% of their time reading code (see the paper by Xin Xia and colleagues from 2017). The less time developers spend reading code, the more productive developers become.

Your goal should be to write code that others can easily understand. There are many different patterns and principles that you can apply. The Clean Code book is the canonical reference for

writing readable code. Although I don’t agree with 100% of Uncle Bob’s suggestions in that book, it’s a great read nevertheless.

In this section, I’ll focus on three ideas that increase code legibility and that I think we should be doing more: looking for good variable names, explaining complex decision points, and writing code comments.

2.2.1 Keep looking for good names

Naming is a critical aspect of writing maintainable code. The closer your code resembles how business people communicate, the better. Good variable names enable developers to quickly grasp a method’s purpose and are particularly essential in information systems, as code should mirror business terminology.

Choosing an initial name for a variable, method, or class is tough. In a controlled experiment conducted by Feitelson, Mizhari, and Noy (in "How Developers Choose Names", published in 2022), researchers noticed that if they ask two developers to name the same variable, it’s likely that they’ll pick different names. That’s why this pattern isn’t about coming up with good variable names; it’s about continually searching for the right name and refactoring until you find it.

When implementing a method, you may only have a vague idea of how a new variable will be used. Will it be combined with another variable, passed to another method, or returned to the caller? Since giving first names is difficult, I prefer not to dwell on them. I move forward, wait until I have more concrete code, and then focus on finding better variable and method names.

I often consider these questions when naming:

- Is the class name fitting and representative of the concept?
- Does the attribute name reveal its information while aligning with the class name?
- Does the method name clearly describe its function, expectations, and return?
- Is the interface name indicative of its concrete implementations' actions?
- Does the service name specify the actions it performs?

This list isn’t exhaustive but aims to guide your naming considerations. You might initially answer yes to these questions, only to change your mind later. This is natural, especially during early development stages. Because of that, don’t hesitate to rename variables, methods, or classes repeatedly. Investing time in renaming saves future developers considerable effort.

Ubiquitous language

Popularized by Domain Driven Design, the ubiquitous language refers to a shared and consistent language used by all development team members to communicate and understand the domain concepts. This language should be reflected in the code. This helps to eliminate confusion and ensure that everyone has a clear understanding of the problem domain. This is highly related to what we just discussed.

2.2.2 Document decisions

A characteristic of information systems is that they make complex decisions based on lots of data. Because of that, decision points in the code can quickly become complex.

It's not uncommon to see `if` statements with multiple conditions. These `if` statements are crucial to the system and so developers should be able to easily and quickly understand what they mean.

There are a few things you can do to clarify decisions:

- You can introduce extra variables in the code to better explain the meaning of complex `if` statements
- You can break the decision process into a set of smaller steps, each taking one part of the decision (which we'll talk more about later)
- You can write a code comment that explains it (which we'll also talk more about later)

Regardless of how you do it, remember that clarifying the decisions a unit of code may take is vital for code legibility, as understanding the decision-making process eases maintenance and debugging.

2.2.3 Add code comments

Code comments written in natural language can be powerful, and you should want to write a code comment for many reasons.

Some developers argue that the code isn't clear enough if a code comment is needed. Others claim comments become deprecated quickly, or they see pointless code comments everywhere. Although all these points are valid, and you should always try first to improve the legibility of the code through refactoring, there are still cases where refactoring isn't enough.

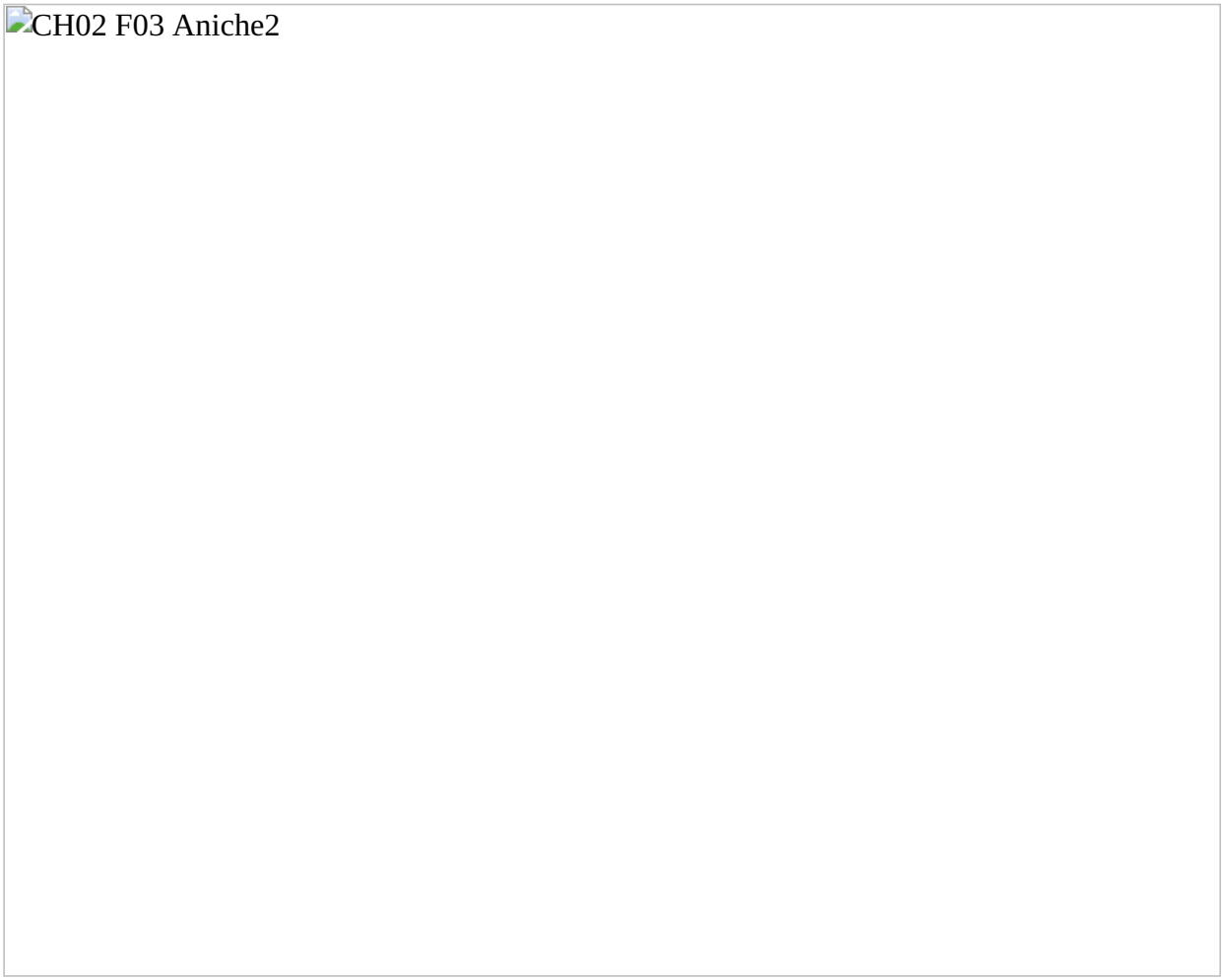
The first reason you may want code comments is to explain things that go beyond the code. Clean code is excellent for explaining implementation details, but they aren't great for explaining the why's. For example, what was the business reason behind the decisions of this code? Why did we take approach A instead of B?

Some may argue that this information fits best on an internal wiki page. Wikis are great tools as you can write text with formatting. I still prefer to write code comments as developers are always in their IDEs closer to the code. I've seen teams adding comments that link to a page in their internal wikis. This is also a great solution that combines the power of both tools.

Another reason you may prefer comments is that sometimes breaking the code into smaller pieces reduces rather than increases its readability. For example, sometimes, an extract method refactoring can make the code look cumbersome, especially if the code in the middle relies on numerous variables from preceding or subsequent sections. This can be evident when using the automated refactoring in your IDE, which may propose a private method with a strange signature or even fail.

Having a code comment separating the different blocks of code is a great way to explain the code that's about to come. Given that it's easy for a developer to identify the different blocks of comments, it's also easy for them to skip to the next block if they don't need to read that part of the code. Figure 2.2 illustrates this.

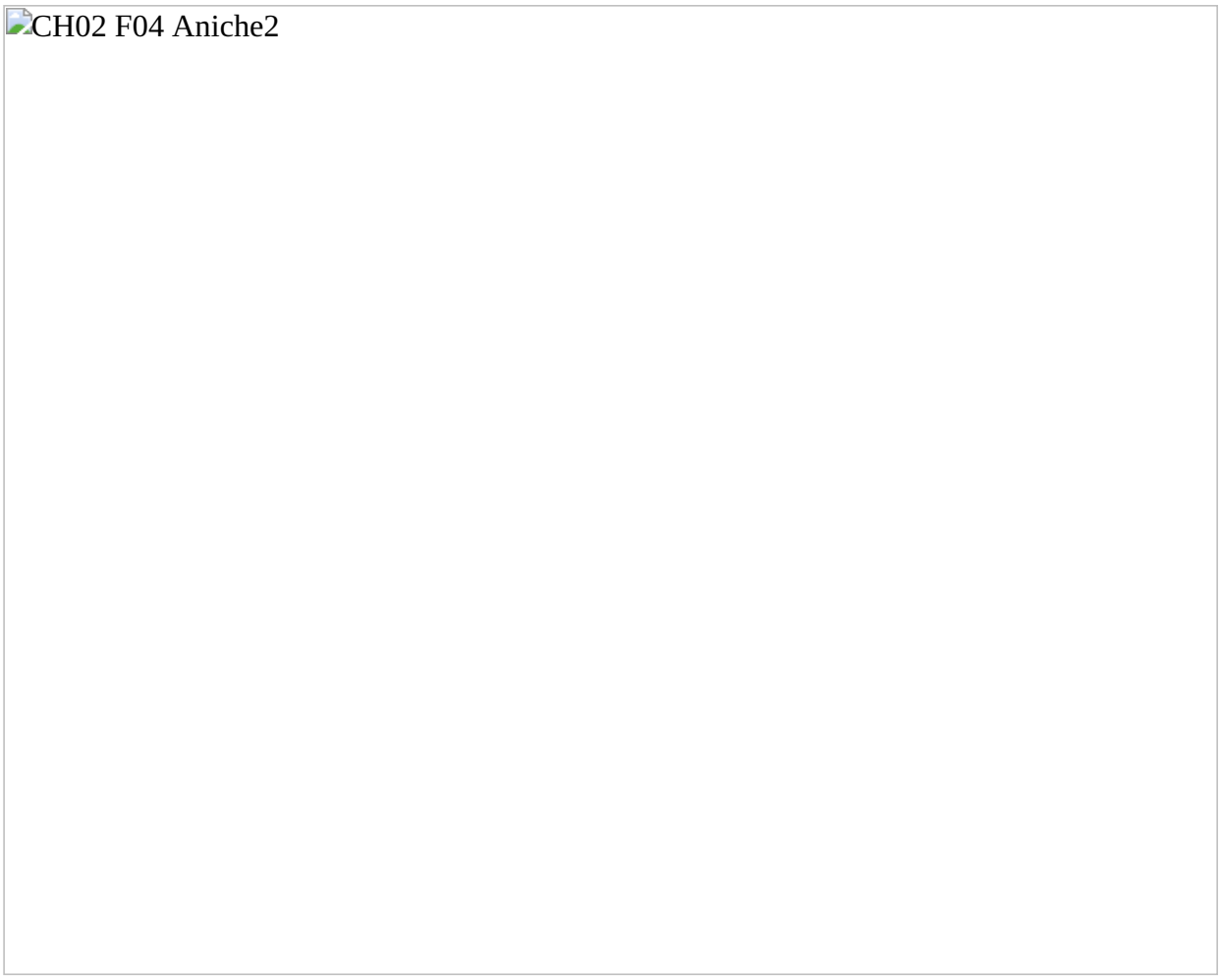
Figure 2.2 Comments separating code blocks



Finally, another reason is that you want to save time from developers. In many cases, just reading the name of the method and its parameters isn't enough to understand all its details. Think of all the times you had a question about a method from any library you use. The name of that method was clear enough, but you still needed more information. Did you read its code, or did you read its documentation? I'd guess the documentation.

Figure 2.3 shows a screenshot of an Apache Commons Lang library method. Note how its documentation is clear and saves you from the hassle of reading the implementation.

Figure 2.3 The implementation of swapCase in Apache Commons Lang



Reading a summary of what the method does, its main caveats, what it does in case pre-conditions aren’t met, and so on, is much faster than reading code. Remember that in most cases, you don’t need to know the in-depth details of a method, only high-level information.

While you may want to document every single public method of an open-source library that’s going to be used by millions of developers, you don’t want to document every single method of your software system, but only the ones that contain complex business rules or the ones that have been through interesting decision making processes. Understanding the "why"s behind coding decisions is crucial when it comes to maintaining that piece of code in the future.

Let’s talk about the problems of comments now. A significant disadvantage is that they can become deprecated without no one noticing it. However, deprecated comments occur when the comment isn’t essential. Crucial comments are usually updated by developers. If you identify a comment that no one bothers to update, delete it.

Writing code with newcomers in mind is essential. A "useless code comment" might seem pointless to you, but it could help someone who’s never seen the code before. Nevertheless, if you find a useless comment, delete it.

Note

In "A Philosophy of Software Design" by John Ousterhout, he offers a refreshing perspective on code comments. His views on their importance and usefulness are worth exploring. I highly recommend reading his ideas on this topic.

2.2.4 Example: Deciding when to send an update e-mail

Whenever an administrator updates an offering in PeopleGrow!, all employees that are enrolled for that offering should receive an e-mail. Or at least that's how the feature started. The business later noticed that employees were getting spammed by the number of e-mails. If an administrator changes the number of available positions for a trainer, a participant doesn't have to know about it.

It was then agreed that updates would only be sent if the offering's dates or specific information (imagine a text field where admins write the room where the training will happen or the Zoom link) changed. Nevertheless, employees can still opt-in if they want to receive all updates. See the current code in listing 2.4.

Listing 2.4 Deciding whether an employee should receive the update email

```
public void update(UpdatedOffering updatedOffering) {

  // ...
  // logic to update the offering
  // ...

  if(employee.wantsAnyEmailUpdates() || (updatedOffering.isDateUpdated() ||
updatedOffering.isInformationUpdated())) { #1

    // send an update email to the employee

  }
}
```

Note how the `if` statement is hard to understand. You would need to dive into the code to get what this decision point is all about.

Let's refactor this snippet.

- First, let's separate the decision from the code that sends the e-mail. Let's move the entire decision process to a method.
- Then, let's introduce variables to explain the different parts of the `if` statement to simplify its legibility.

See listing 2.5 for the refactored version. Note how much easier it is to understand what the method does. You just read the comments and understand how the implementation works as the complex `if` is broken down.

Listing 2.5 A better version of the algorithm that decides if an employee receives an update email

```
/**
 * An employee should receive an e-mail in case #1
 * they opt-in for it or in case important information was updated.
 */
boolean shouldReceiveAnEmail(Offering updatedOffering,  Employee employee) {

  boolean datesUpdated = updatedOffering.isUpdated();
  boolean informationUpdated = updatedOffering.isUpdated();
  boolean importantInfoUpdated = datesUpdated || informationUpdated; #2

  boolean employeeWantsAllUpdates = employee.wantsAllEmailUpdates();

  return employeeWantsAllUpdates || importantInfoUpdated;
}

...

if(shouldReceiveAnEmail(offering, employee)) { #3
  // send an update email to the student
}
```

Where to place the shouldReceiveAnEmail method?

The more we refactor the more we learn about the code. It feels to me that the `shouldReceiveAnEmail` shouldn't be placed in the implementation of the service.

Should we move this method to either the `Offering` or the `Employee` entities? Should we give this behavior an entirely new class just so this business logic can evolve independently from the rest of the service logic? These are the questions I'd be asking myself if I were implementing this in a real system.

I won't do a second round of refactor here as the example already illustrates the point of this section.

Don't be afraid of documenting your code. Documentation is key to productive maintenance.

2.3 Move new complexity away from existing classes

Move any new complexity arising from feature requests or code evolution to a separate location. This pattern fosters simplicity and cohesiveness within existing units, making them easier to maintain and comprehend.

Deciding when to create a new class while evolving code can be challenging. It's tempting to keep adding code where the rest of it already exists. However, continuously avoiding this question leads to lengthy and complex code.

Methods and classes that grow indefinitely eventually become unmaintainable. They should expand as business complexity increases, but this growth must be controlled. Overgrown code units eventually become too complex for developers to maintain with reasonable effort, resulting in complex classes.

In many cases, classes grow indefinitely due to a lack of good abstraction or extension points that enable developers to keep adding new behavior without changing existing code. I'll cover these topics in chapter 4 and 5. In this section, I'll focus on two recurrent design decisions I take to avoid classes growing forever: move complex business rules to their own class, and break down large business flows into multiple steps.

2.3.1 Give the complex business logic a class of its own

Whenever there's a new complex business logic to implement, trying to fit it into an existing class can be a burden. You may never be able to find the suitable class because the rule spans multiple classes. You may also have to add too many other dependencies to an existing class because the rule will likely require interaction with many other classes.

In such cases, it's best to create another class that isolates the feature, keeping the other classes free from growing in complexity. There are a few advantages to doing that:

- First, dedicating an entire class to a complex feature makes it easier to see all its dependencies. In this case, email-related classes and different repositories are explicitly listed in the service's constructor.
- Second, the feature's code is isolated from the rest of the system. When reading the code, there's no need to separate what belongs to this feature and what doesn't, reducing cognitive overload.
- Third, isolation also makes testing easier. You can write tests for this feature without worrying about other behaviors.
- Fourth, it simplifies reusing the functionality. This class can be called from any part of the system that needs the same functionality.
- Finally, the class is highly cohesive, doing one thing only.

A caveat is that, at the same time, you should keep your business logic as close as possible to the class it acts on and not on a class "far away," like I just said. For example, marking the `Enrollment` as canceled should happen inside the `Enrollment` class and not outside. We'll discuss encapsulation, state, and consistency in chapter 3.

2.3.2 Break down large business flows

Some long, complex methods exist because they control sizeable multiple-step business flows. Without a proper abstraction to help you model the flow, you may end up with a huge class where each method is one step of the flow. Imagine a business flow composed of 10, 15, or 20 steps. Putting the implementation of all steps in a single class isn't a good idea.

You should break down large complex business flows into simple, small, cohesive units of code and use intelligent design mechanisms to deliver the entire flow. If each step is in one class, you get all the benefits of small classes again:

- They are simple and easier to be understood.
- They are easier to be maintained.

- They are more easily testable.
- They can be better reused.

While there are frameworks for organizing workflows, a simple abstraction often suffices. If you need patterns for breaking up multi-step business flows, consider Gang of Four's design patterns like Chain of Responsibility, Decorator, or Observer. For highly complex business flows, you might explore domain events.

Another alternative is to build an event-based system, where each process step generates an event consumed by the next step. Consider this option for complex workflows requiring flexibility or when steps should occur in different services. Otherwise, stick to simpler workflow mechanisms. I won't delve into event-based system implementation details, as many books cover this topic (choose any microservices book).

2.3.3 Example: waiting list for offerings

PeopleGrow! offers a waiting list feature. If an offering is full, employees can join the waiting list. If someone unenrolls, the entire waiting list is notified, and the first employee that enrolls gets that spot. An employee that enrolls in an offering is automatically removed from the waiting list.

Let's focus on the part where someone unenrolls, and we need to send an e-mail to all employees on the waiting list. The first implementation proposed by the developer was to implement this functionality inside the `UnenrollEmployeeFromOfferingService` service. The service would retrieve the employees on the waiting list and then loop through this list and create an e-mail for every one of them. Listing 2.6 shows the initial implementation.

Listing 2.6 Notifying the waiting list when an employee unenrolls

```
class UnenrollEmployeeFromOfferingService {

  private Emailer emailer;
  private OfferingRepository offerings;

  public UnenrollEmployeeFromOfferingService(...,
    OfferingRepository offerings,
    Emailer emailer) {
    this.offerings = offerings;
    this.emailer = emailer; #1
  }

  public void unenroll(int enrollmentId) {
    // ...
    // logic to unenroll the employee
    // ...

    Offering offering = offerings.getOfferingFrom(enrollmentId);
    notifyWaitingList(offering); #2
  }

  private void notifyWaitingList(Offering offering) { #3
```

```
    List<Employee> employees = offering.getWaitingList();
    for(Employee employee : employees) {
      emailer.sendWaitingListEmail(offering, employee);
    }
  }
}
```

Adding this implementation to the existing `UnenrollEmployeeFromOfferingService` isn't a good idea, as it makes the class more complex. As soon you add the `notifyWaitingList`, you might be forced to review all your automated tests for the `UnenrollEmployeeFromOfferingService` and see if they still work. It's also harder for you to write tests for the new feature in isolation.

A better option would be to move the waiting list notification logic to a separate class, say `WaitingListNotifier`. Move the complexity away from existing classes, as we said.

That's what we'll do. See listing 2.7. The `UnenrollEmployeeFromOfferingService` now depends on the new `WaitingListNotifier` and calls when it's time to notify the employees. The waiting list notifier depends on the `Emailer` and has the same logic as before.

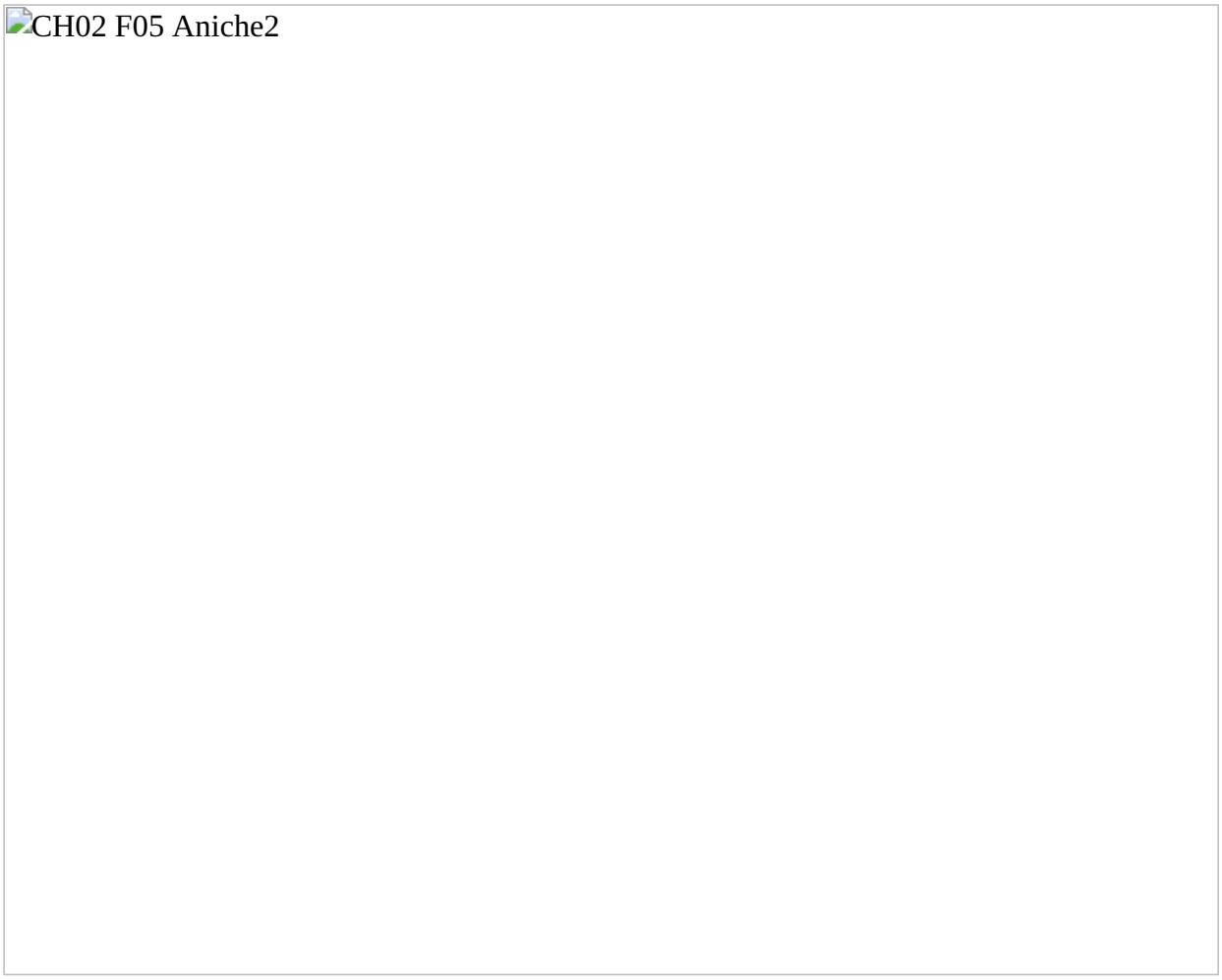
Listing 2.7 Waiting list notification in another class

```
class UnenrollEmployeeFromOfferingService {

  private OfferingRepository offerings;
  private WaitingListNotifier notifier; #1

  public UnenrollEmployeeFromOfferingService(...,
    OfferingRepository offerings,
    WaitingListNotifier notifier) {
    this.offerings = offerings;
    this.notifier = notifier;
  }

  public void unenroll(int enrollmentId) {
    // ...
    // logic to unenroll the employee
    // ...

    Offering offering = offerings.getOfferingFrom(enrollmentId);
    notifier.notify(offering); #2
  }
}

class WaitingListNotifier {

  private Emailer emailer;

  public WaitingListNotifier(Emailer emailer) { #3
    this.emailer = emailer;
  }

  public void notify(Offering offering) { #4
    List<Employee> employees = offering.getWaitingList();
    for(Employee employee : employees) {
```

```
      emailer.sendWaitingListEmail(offering, employee);
    }
  }
}
```

Figure 2.4 depicts a class diagram that illustrates the new class design. Note how we managed to move new complexity away from existing classes. And we did it by creating a new class and delegating this new behavior to it. The new class is also small, easily testable, and reusable. This simple pattern works more often than you'd expect.

Figure 2.4 A set of smaller classes work together to deliver the unenrolling employee feature



We ended up with a simpler design by simply forcing ourselves to move new complexity away from existing code.

2.4 Exercises

Discuss the following questions with a colleague:

2.1. If you were to define a hard threshold for the maximum lines of code for a method, what would this threshold be? And why?

2.2. How do you currently document your software systems? Does it work? Do you see room for improvement?

2.3. Have you ever encounter bad comments in the code? And what about good code comments? What did both look like?

2.4. This chapters strongly argues for small classes. What are your thoughts on it? Do you also prefer having many small classes instead of a larger class, or do you see advantages in larger classes?

2.5 Summary

- Code complexity arises when code units become too large and difficult to understand, maintain, and extend, leading to bugs.
- To reduce code complexity, break down complex methods and classes into smaller ones, such as splitting a large method into smaller private methods or moving code to different classes.
- Use explanatory variables, clear method names, and natural language comments to make code easy to read for other developers. Code comments should explain why decisions were made, while easy-to-read code should explain the implementation logic.
- Move any new complexity away from existing units. You can do that by, for example, moving the new feature to a new class.

3 Keeping objects consistent

This chapter covers

- Keeping classes consistent
- Emerging problems with inconsistent objects
- Implementing validation mechanisms that ensure consistency at all times

A well-designed class encapsulates its data and provides operations to access or manipulate it. These operations ensure the object remains in a valid state without inconsistencies. Better yet, they do so in a way that the clients of the class don’t even need to know about it.

One of the greatest advantages of object-oriented programming is the ability to ensure that objects are always in a consistent state. Compare it with, say, procedural programming languages like C. In C, you can define data structures (known as structs). However, there’s no way to control who changes the values inside the struct. Any piece of code, anywhere in the codebase, can change it.

When code is not appropriately encapsulated, developers feel they can never find where to patch the code and can never fully fix a bug in one shot. When code is spread and not encapsulated, developers have to search for where things are all the time. Just going to the class that defines the abstraction isn’t enough. They do it as soon as they find the place to fix the code, but then the same bug appears in another place.

Encapsulation, the idea of keeping data internally in the object and allowing users to manipulate them only through elegant operations, is the cornerstone of object-oriented programming. It’s due to encapsulation that we can change details that are internal to a class without affecting the rest of the code base.

In this chapter, I’ll present some patterns that will help you design classes that’ll stay consistent no matter what.

Consistency or integrity?

In object-oriented programming, the word "consistency" is commonly used to indicate that an object has accurate and reliable information. In other computer science domains, such as databases, the term *"_integrity*"_ is the one that refers to the accuracy of the information (while "consistency" is often related to the availability of the data).

3.1 Ensure consistency at all times

Ensure that objects are in a consistent state at all times. This improves reliability as objects are always in a valid state regardless of where and how they are being used.

When objects are consistent, it means that their internal state is synchronized and coherent with the program’s requirements and the users' expectations. Maintaining consistency ensures that objects always behave correctly and produce accurate results, leading to reliable and trustworthy software.

The rule of thumb when it comes to consistency is: does the client of the class need to do some work to ensure the object is in a consistent state? If yes, you likely have a poorly designed class that may haunt you soon.

Maintaining a consistent state should take minimal, if not zero, effort from clients. In the following sections, we’ll explore patterns that make encapsulation a breeze.

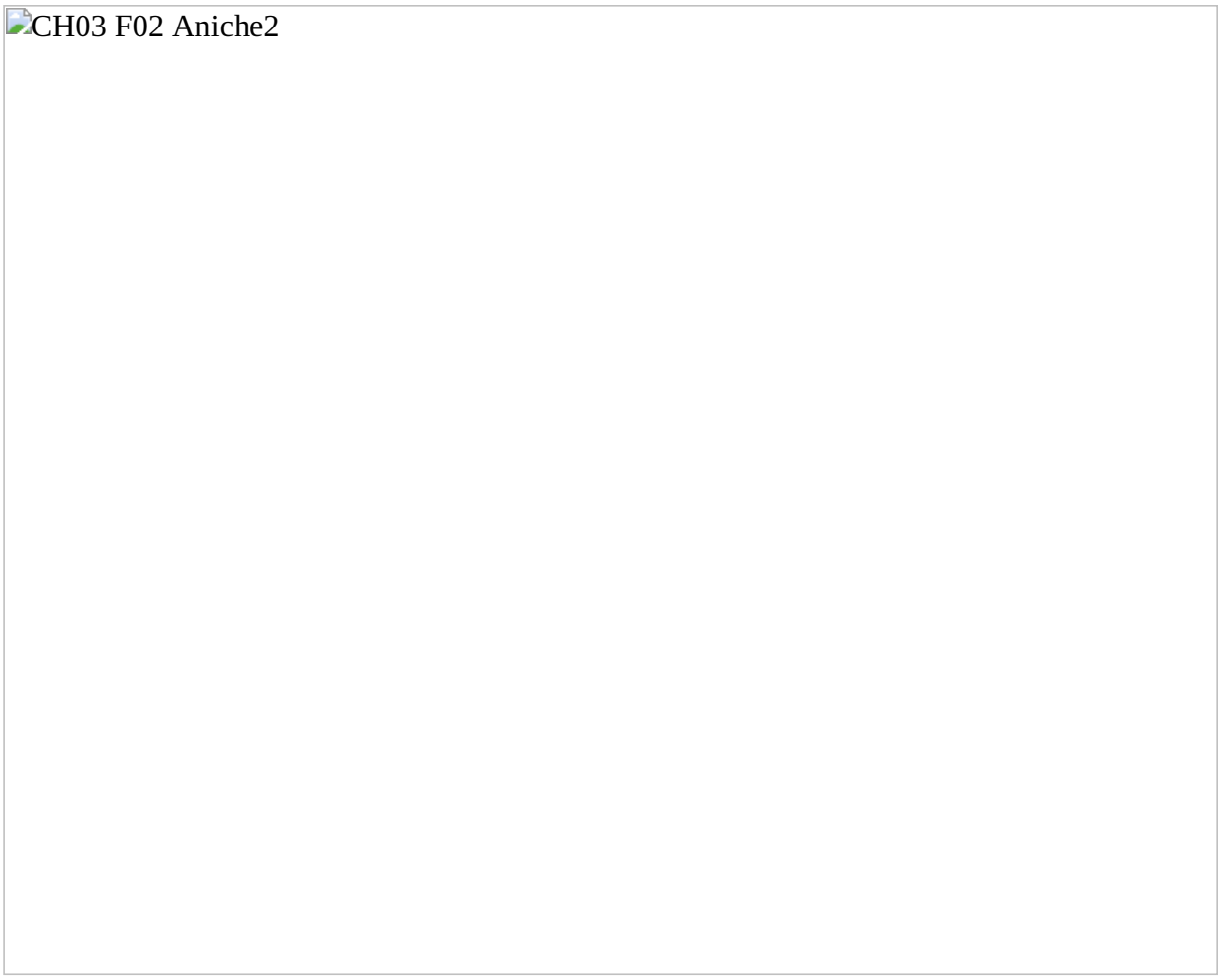
3.1.1 Make the class responsible for its consistency

You should ensure consistency checks are coded in the class itself. If unsure which class should maintain data consistency, consider the one containing the data.

In information systems, entities are usually the classes containing the data. For example, ensuring that an `Offering` never accepts more participants than its maximum allowed limit should happen inside the `Offering` class. Or that the number of empty seats gets reduced by one as soon as someone enrolls. The clients of these classes shouldn’t be doing things like "Ok, I just added one more participant to the offering; let me reduce the number of available spots by 1 now".

Figure 3.1 illustrates the idea. Stay alert. If you find yourself coding consistency checks outside the class, pause and reconsider. Can consistency always be ensured inside the class? Unfortunately, not.

Figure 3.1 Entities ensuring their own consistency



3.1.2 Encapsulate entire actions and complex consistency checks

Some consistency checks need information beyond the ones in the object, or they could be too complex, requiring the code to live somewhere outside the class. For example, you may need to consult the information in the database to decide whether an operation is valid. You can't do (and shouldn't want to do) that from inside the entity.

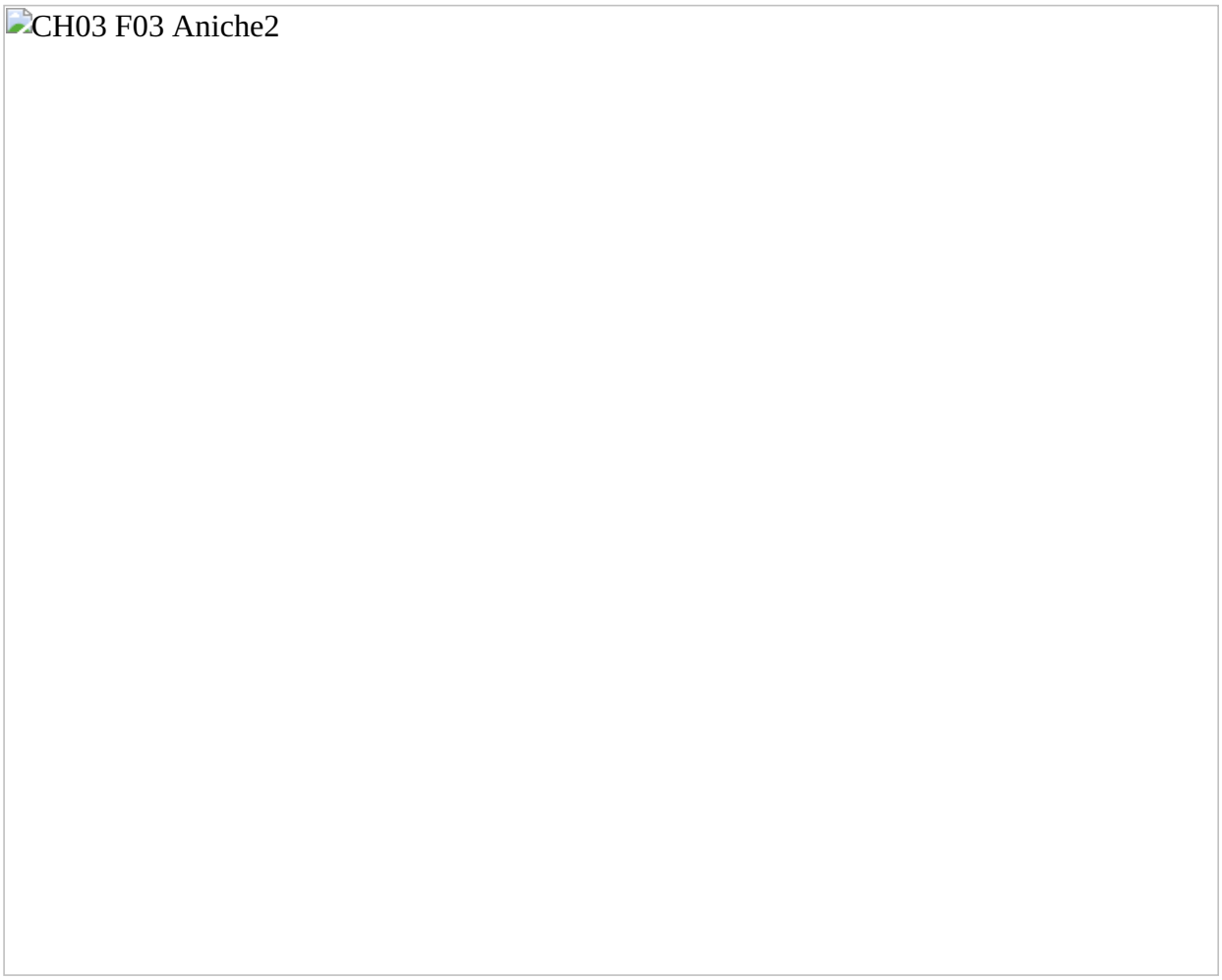
Note

You can technically access the database from an entity, and some frameworks, especially the ones that support Active Record, like Ruby on Rails, make it easy. However, this is not the standard in most architectures and something we usually avoid.

In these cases, you want to encapsulate the consistency checks and the action in one class. For example, in Service classes, as described in chapter 1.

Services ensure the consistency that the entities can't do it by themselves. Entities still ensure as much as they can on their own. Figure 3.2 illustrates the overall idea of this pattern.

Figure 3.2 Services and entities working together to achieve consistency



The advantage of moving the entire consistency logic to a new class is that we can implement complex consistency checks without making the original (entity) class more complex. The clear disadvantage is that you moved this logic away from the class in which it's supposed to be, and now the clients of the classes need to know that they should use the Service.

Documentation plays an important role here. You should write code comments in the most natural places a developer would look for if they want that behavior. For example, the `Offering` entity may deserve a comment saying that in case some client wants to add an employee to a training, it should use the service class.

Although it's sad to see behavior separated from data, sometimes it's necessary. Clients should never be responsible for consistency checks; the class itself should manage it by default. If checks are too complex for the class, move the entire operation, including consistency checks, to a service class, and require clients to use this class for the desired behavior.

Note

Later in this chapter, I'll talk more about aggregate roots, as moving behavior to another class becomes tricky once the class is actually an aggregate of multiple classes and has many invariants to take care of.

The complexity of the consistency checks is an excellent reason to move the entire operation to another class. There might be other good reasons too. Avoiding unwanted dependencies is one of them. I'll talk more about class dependencies in future chapters, but I like my domain classes to be "as pure as possible" and contain only data and methods that operate on this data. I especially avoid as much as possible coupling them to classes that access things like databases or any other external software system. As a rule of thumb, the object should take care of all the consistency checks it can without relying on additional dependencies.

3.1.3 Example: The Employee entity

An offering in PeopleGrow! contains the date it will happen, the list of employees attending the training, the maximum number of allowed participants, and an open description text field where administrators can write information like the room where the training will happen, Zoom link, and so on.

The first version of the `Offering` class is shown in listing 3.1. Note how the class doesn't ensure consistency. This is a poorly designed class, as it's easy to put it in an inconsistent state. Clients have to check whether there are available spots before adding an employee and then, later, subtract the number of available spots by one (see 3.2). Imagine a client forgetting to update the number of available spots. Suddenly the object is invalid.

Listing 3.1 The Offering entity

```
class Offering {

  private int id; #1
  private Training training;
  private Calendar date;
  private List<Employee> employees;
  private int maximumNumberOfAttendants;
  private int availableSpots;

  public Offering(Training training, Calendar date,  int
maximumNumberOfAttendants, int availableSpots) { #2
    this.training = training;
    this.date = date;
    this.maximumNumberOfAttendants = maximumNumberOfAttendants;
    this.availableSpots = maximumNumberOfAttendants; #3
  }

  public List<Employee> getEmployees() { #4
    return this.employees;
  }

  public int getAvailableSpots() { #5
    return this.availableSpots;
  }
```

```
  public void setAvailableSpots(int availableSpots) { #6
    this.availableSpots = availableSpots;
  }
}
```

Listing 3.2 Clients using the Offering entity

```
Offering offering = getOfferingFromDatabase(); #1

if(offering.getNumberOfAvailableSpots() > 0) { #2
  offering.getEmployees().add(employeeThatWantsToParticipate); #3
  offering.setAvailableSpots(offering.getAvailableSpots() - 1); #4
}
```

The first refactoring we must do is to ensure that the `Offering` class ensures its internal consistency. This means the class should make sure the number of available spots gets subtracted by 1 if someone is added to the training. The class should also not allow a new employee to join the training if it's already full.

See listing 3.3. This new version of the class has a much better `addEmployee` method. The method ensures no one can be added to the training without empty spots. The method also keeps tabs on the number of available spots. The clients of this class now don't need to know anything about how offerings work. Whenever they want to add an employee, they call `addEmployee()` and move forward.

Listing 3.3 The Offering entity with the new addEmployee() method

```
class Offering {

  private int id;
  private Training training;
  private Calendar date;
  private List<Employee> employees;
  private int maximumNumberOfAttendants;
  private int availableSpots;

  public Offering(Training training, Calendar date,  int
maximumNumberOfAttendants) {
    // basic constructor
  }

  public void addEmployee(Employee employee) { #1
    if(availableSpots == 0)
      throw new OfferingIsFullException();

    employees.add(employee);
    availableSpots--;
  }

  public int getAvailableSpots() { #2
    return this.availableSpots;
  }
}
```

It’s also important to notice that the class doesn’t offer the `getEmployees` method anymore. We don’t want clients to be able to handle an internal data structure of the class by themselves without any control. Only `Offering` should handle the list of employees.

Returning a copy of the data structure

One way to avoid giving clients full access to an internal data structure of the object, like in this case where we don’t want to give access to the internal list of employees, is to give clients a copy of that data structure. In this case, whatever changes they do in the copy won’t affect the original data structure that’s only available internally to the object.

I’ll discuss when and how to offer getters and setters later in this chapter.

We can also improve the construction of the class. The class shouldn’t allow an `Offering` to be created with an empty training or a negative number of maximumNumberOfAttendants. I’ll talk more about this when we talk about validation in the next section.

Concurrency and design

Don’t forget that your non-functional requirements may be a force that influences your design decisions. For example, if you are expecting multiple simultaneous requests to add employees to trainings, the code above may not work, as there’ll be concurrent access to the `availableSpots`. To solve this particular problem, you could ensure that requests for the same training are processed linearly through, say, some smart queueing. Or, class design-wise, you could get rid of `availableSpots` and completely avoid the concurrent access to this field.

I won’t dive into concurrency patterns in this book, but the overall message is that you should never forget about all your functional and non-functional requirements when designing your classes.

3.2 Design effective data validation mechanisms

Validate client data to prevent unexpected errors and reduce the risk of odd behavior in the system. Clearly define the consequences of invalid data. This pattern improves code reliability and user experience by ensuring that the system can handle and communicate issues better.

Validating data throughout your system may seem tedious, but it pays off in the long run. Consider what would happen if a user requests the creation of a training offering with an empty date. Will your software crash, or will it handle the invalid input gracefully?

I’ll discuss two design approaches to handle data validation in the following sections. One focuses on explicitly defining pre- and post-conditions of your methods, while the other validates input data from a business perspective.

3.2.1 Make pre-conditions explicit

Many bugs and inconsistencies in software systems occur when methods call other methods in invalid ways. This can happen for various reasons, such as not knowing how to use a class or method or input data cascading from previous code layers without pre-checks.

The second reason is harder to spot in code. Systems have complex data flows, and you can't be aware of them. By actively ensuring pre- and post-conditions and explicitly documenting them, you reduce the chances of inconsistent program execution and increase the likelihood of others using your classes as intended.

Let's start with pre-conditions. Methods should make it clear what valid values are for each input. For example, the `Offering` class offers a method called `addEmployee` that receives an `Employees` as a parameter. The method adds the employee to that training offering.

What if the client passes null to the method? Null is clearly an invalid input parameter. If nothing is done, the system will likely crash at some point. It's the responsibility of the developer to decide what the method should do if a client doesn't respect the pre-conditions of a method.

You may design different actions when your pre-conditions aren't met. You can take harsh measures against invalid input values, such as throwing an exception. Throwing an exception has an advantage: it halts the program right away. Halting the program is better than continuing its execution when you don't know how to proceed with the invalid data. However, this decision increases the workload for client classes because they now have to handle this possible exception. If that's your intention for that class, it's a perfect decision.

In other cases, you may choose a more lightweight way of handling pre-conditions. For example, the method can accept nulls and "do nothing" in these cases. If a null comes in, the method returns early. This option requires less effort from clients since the method doesn't throw exceptions or stop working in case of invalid input (which is no longer invalid because the method can handle it).

Handling invalid input was easy in this example, but it may require more lines of code in practice. The trade-off is that if you make your code more tolerant toward bad input, saving clients from handling possible exceptions themselves, the method's developer must code a bit more.

Overall, explicitly thinking of pre-conditions is a fundamental design activity that'll save your code from crashing unexpectedly. This is one of the few practices discussed in this book that focuses more on quality, as in "this code works" rather than quality, as in "this code is easy to maintain." However, code that works is easier to maintain.

Additionally, the maintenance aspect is related to how you decide to handle the pre-condition. In his book *A Philosophy of Software Design*, John Ousterhout says that we should "define errors out of existence." For example, imagine a method that returns all employes that are enrolled in a training. Instead of throwing an exception in case no employees are yet enrolled, you may return an empty list. Or, imagine a method that marks an invoice as paid. If the invoice is already paid, instead of throwing an exception in case someone tries to mark it as

paid again, the code simply does nothing. By simplifying the pre-conditions of the code, you simplify the lives of yours clients as they have less corner cases to handle.

If you can design your code so that it can’t fail, that’s better for maintenance. If you consider that you develop the class once, but it’ll be used multiple times by different clients, saving them effort is also a good thing to do in the long run.

3.2.2 Create validation components

In enterprise systems, actions often require more than just meeting method pre-conditions. Take the enrolling in a training offering feature as an example: the `addEmployee` has a simple pre-condition, don’t accept nulls. However, from a business perspective, there might be additional validations, like employees can’t take the same training more than 3 times, or the employee must be from a specific office, and so on. These rules aren’t pre-conditions per se, but we must ensure the request complies with them.

Business validation rules are pervasive in enterprise systems, so you should handle validation explicitly in your code by giving rules their own classes. This enables reusability and clarity when clients call validation methods.

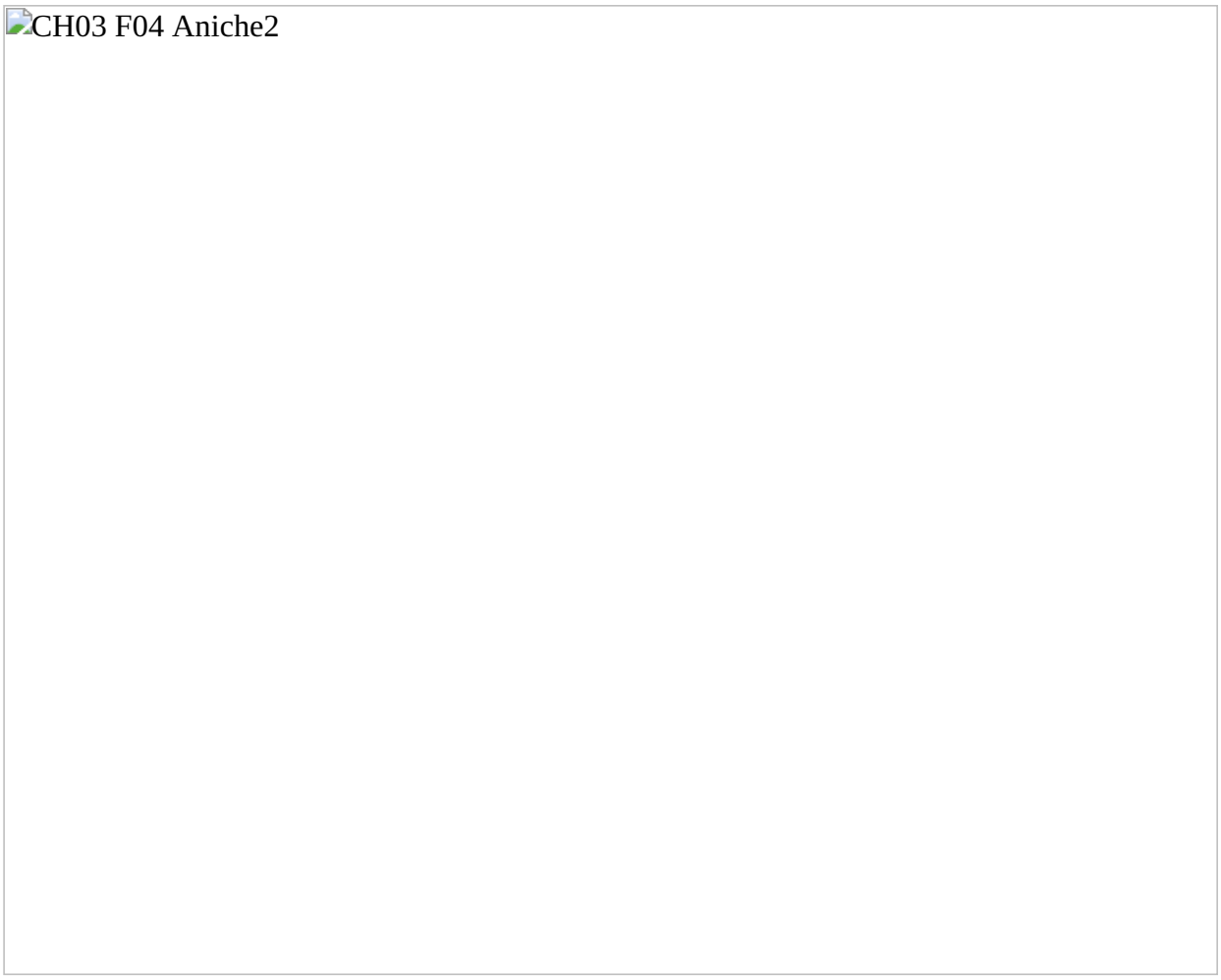
Pre-conditions versus validation rules

Pre-conditions are the minimum requirements for a unit of code to execute correctly, such as "this attribute can’t be null" or "this value should be either A, B, or C." Validation rules are more business-related and often require more code for checks, like "employees can’t take the same training more than 3 times."

You don’t want clients to handle consistency or pre-conditions checks, and you don’t want them to know that validation rules should be called before an action. In this case, it’s also wise to design a Service class to control the flow and ensure that validations and consistency checks happen before the action.

Figure 3.3 enhances our previous figure. Services do consistency and validation checks that entities can’t. Services may get the help of validation components. Entities still perform all the consistent checks they can.

Figure 3.3 Validation components helping keep consistency



Let me make a few final remarks about validation classes:

- If you are sure you'll need to reuse the same validation rules for other use cases or service classes, design it so that it can be reusable elsewhere. If you are looking for a way to build flexible validation rules, you should search for the Specification pattern, which became popular after the Domain-Driven Design book. The pattern allows you to define rules and compose them in different ways. However, most validation rules are specific to the feature or a specific service. Don't overdesign validation classes.
- Should we call validation components from inside a domain class? Since many validation rules require external dependencies, including databases, handling pre-conditions and internal consistency checks within the class and moving business validation rules to other classes is preferable. Using a Service class to coordinate is a worthwhile trade-off for complex business actions.
- If you try to instantiate an entity that does all the proper consistency checks directly with the data from the user, you may get some pre-condition violation right away without even having the chance to do other checks. To avoid this, you can introduce intermediate classes like `OfferingForm`, which are data structures to hold the data from the user without validation. Once the data is valid, you convert it to a proper `Offering` class.

3.2.3 Use nulls carefully or avoid them if you can

Sir Tony Hoare introduced nulls in 1965 as a convenient solution, later calling it his "one billion dollar mistake." Nulls can be tricky, causing unexpected null pointer exceptions and hindering readability with excessive null checks.

The possibility of a class returning null forces clients to perform null checks everywhere, which makes hampers readability. In listing 3.4, note what would happen if we had to check for nulls at every point of our code.

Listing 3.4 Null checks everywhere

```
var obj1 = method1(); #1
if(obj1!=null) {
  var obj2 = method2();

  if(obj2!=null) {
    var obj3 = method3();

    if(obj3!=null) { #2
      // ... code continues ...
    }
  }
}
```

This example might be an extreme one, but I hope you get the point. Adding null checks everywhere is something you don’t want to do.

The best you can do is to ensure your methods never return null. How can you do it? Consider the following:

- If your method has a path that should return "nothing," consider creating an object that "represents nothing." For example, if your method returns a list, is it a problem if you return an empty list if the operation isn’t successful?
- If you want to return null because there was a problem in the execution of the method, should this method then throw an exception? Can we make this method return a class that describes the problem?
- Can we design the error out of existence? For example, if the client passes a null to a list parameter, can we assume that this list is empty, instead of returning null back?
- If your method returns null because a library you don’t control returns nulls, can you wrap this library call and transform the result?

A system without null returns is easier to work with, but avoiding nulls requires time and effort. You have to invest some time in designing the null away.

What to do if you need to represent "absence of information?" Isn’t what 'null' is all about? Languages like Java offer the `Optional` type, which can be helpful in such situations. Clients must check for the presence of a value before proceeding.

I also don’t want to discard the need for a null return you can’t control. Life is hard, and you may be in one of these situations. In this case, I suggest you document this behavior so that clients know they have to be prepared for it! Avoiding surprises is the best you can do when you can’t model the null away.

Tools can also help you identify pieces of code that may suffer from null pointers. For example, the Checker Framework or IntelliJ’s null detection capabilities are excellent in detecting places where you should handle a possible null return.

3.2.4 Example: Adding an employee to a training offering

PeopleGrow! has many business rules around adding an employee to an offering:

- An employee can’t register if there are no spots available
- An employee can’t take the same training more than 3 times
- An employee can’t be registered more than once to the same offering

These rules can’t be implemented all inside the `Offering` class, as a single instance of offering doesn’t have access to whether the employee already took this training in the past. Given that adding an employee to an offering became more complex, the entity isn’t the best place to code this rule. We need a service and a validation class.

Let’s start with the service class. Its implementation should be straightforward. We validate the request. If valid, we add the employee to the offering (listing 3.5).

As a design choice, I opted for receiving the IDs of the offering and the employee, and let the service retrieve them from the database. Another option would be to receive the `Offering` and `Employee` entities directly, forcing the client to retrieve them both before calling the Service. Both have advantages and disadvantages, but no one is better.

The method then makes sure both of them exist in the database; otherwise, it throws an error. Then, it calls the `AddEmployeeToOfferingValidator` validator to ensure that this is a valid request from a business point of view. Could we have implemented the validation rules in the service directly? If they are simple, sure. If they are more complex, I prefer a dedicated class, as discussed before. If validation goes fine, we add the employee to the offering. If not, we will throw an exception.

Listing 3.5 The AddEmployeeToOfferingService class

```
class AddEmployeToOfferingService {

  private OfferingRepository offerings;
  private EmployeeRepository employees;
  private AddEmployeeToOfferingValidator validator;

  public void addEmployee(int offeringId, int employeeId) {

    var offering = offerings.findById(offeringId);
    var employee = employees.findById(employeeId);
```

```
    if(offering == null || employe == null) #1
      throw new InvalidRequestException(
        "Offering and employee IDs should be valid");

    var validation = validator.validate(offering, employee); #2
    if(validation.hasErrors()) { #3
      throw new ValidationException(validation);
    }

    offering.addEmployee(employee); #4
  }
}
```

Note that the service may also have to handle other aspects of your system, depending on your architecture. If you use an object-relational mapper (ORM) such as Hibernate, you should open a transaction scope. If you aren't using an ORM, you may have to explicitly persist the change in the `offering` instance.

I'm also not explicitly showing how the service should be instantiated in this snippet. Depending on how you architected everything, you may use a dependency injection framework or do it manually.

The implementation of the `AddEmployeeToOfferingValidator` class itself should also be straightforward. See listing 3.6. It contains a sequence of ifs, each checking a business rule, and if the request is invalid, it takes note of it.

Listing 3.6 The AddEmployeeToOfferingValidator class

```
class AddEmployeeToOfferingValidator {

  private TrainingRepository trainings;

  public ValidationResult validate(Offering offering,  Employee employee) {

    var validation = new ValidationResult();

    if(!offering.hasAvailableSpots()) { #1
      validation.addError("Offering has no available spots.");
    }

    var timesParticipantTookTheTraining =
      trainings.countParticipations(employee, offering.getTraining())

    if(timesParticipantTookTheTraining >= 3) { #2
      validation.addError("Participant can't take the training again.");
    }

    if(offering.isEmployeeRegistered(employee)) { #3
      validation.addError("Participant already in this offering.");
    }

    return validation;
  }
}
```

I won’t go too much into the details of the `ValidationResult`. Just imagine a simple class that stores the list of possible errors that the validator identifies. The service can then ask if there were errors (via `validation.hasErrors()`) and decide what to do with them. For example, repassing them to the client that called the service.

What matters in this snippet is that the service coordinates the action and doesn’t allow an invalid request to proceed. It makes sure offerings are always consistent.

This means clients shouldn’t be able to call `addEmployee()` directly; only the service should be able to do that. In many programming languages, you can play with visibility modifiers, module systems, or even static analysis to prevent that from happening.

Regarding the generic validation mechanism, you can implement it in a million different ways. Don’t get too attached to my `ValidationResult` example. As I said before, make it simple and evolve it over time.

Domain services and application services

Domain-Driven Design and Clean Architecture distinguish between domain services and application services. Application services should only coordinate the work and have no business rules, while domain services contain the business rules. The `AddEmployeToOfferingService` acts as an application service as it only coordinates and has no business rule.

Separating application and domain services or separating control flow from business logic helps you simplify your code and domain. As always, I take a pragmatic approach. I start with a simple service, and I’m lenient at the beginning if it’s doing both. However, as soon as the complexity starts to grow, I refactor.

3.3 Encapsulate state checks

Encapsulate state checks regardless of their complexity. This ensures that clients keep ignorant about other classes' internal details, enabling classes to change their internal implementation without breaking the clients.

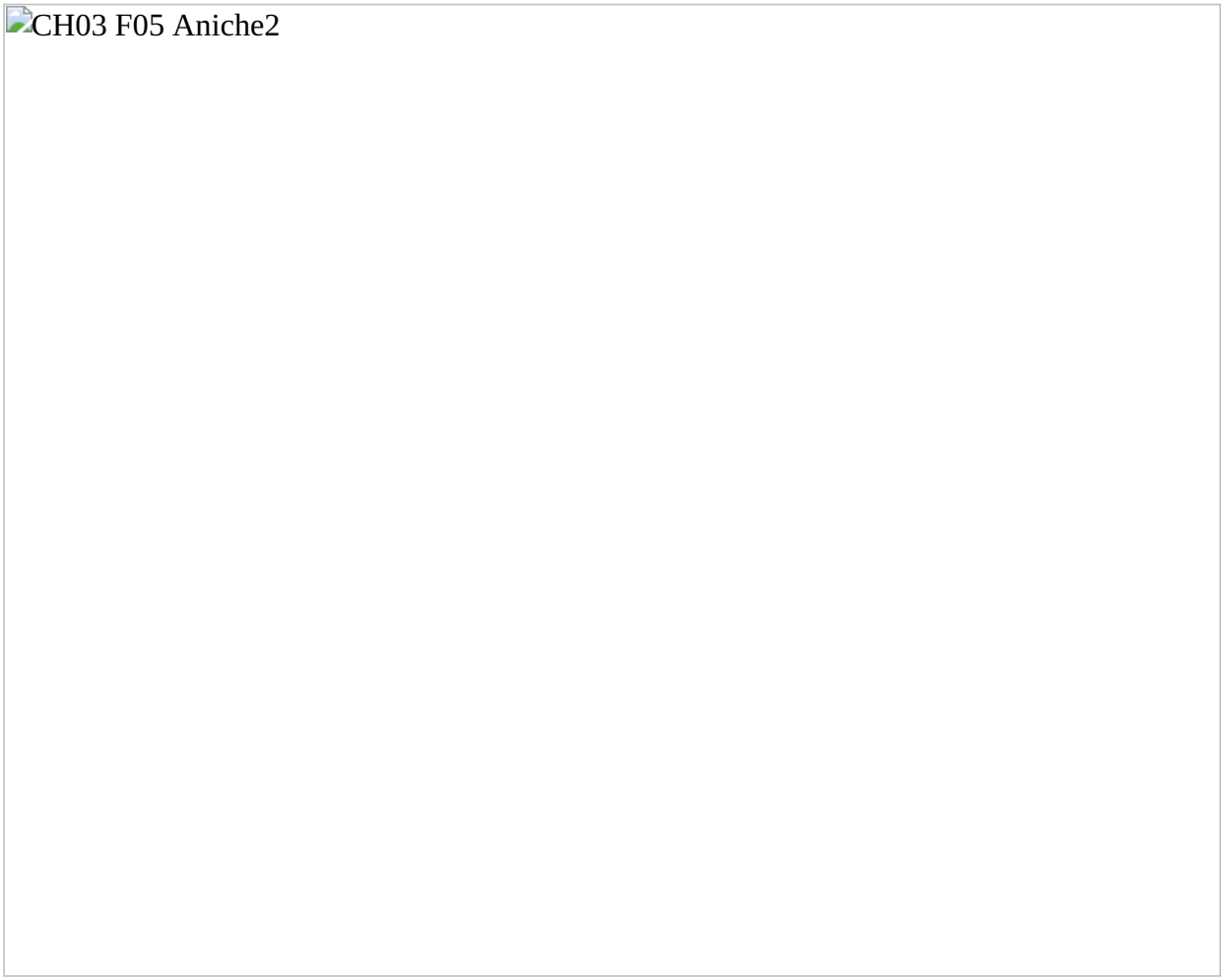
Clients often need to know the object’s state to make decisions. We’re used to encapsulating business rules, but we often forget to also encapsulate state checks.

For example, asking if the `Offering` class still has spots available. An inattentive developer may write things like `if(offering.getNumberOfAvailableSpots() == 0)` (if the number of available spots is equal to zero) or even `offering.getEmployees().size() < offering.getNumberOfAvailableSpots()` (if the number of employees enrolled is smaller than the number of total spots).

Although this may work for a while, this creates a strong coupling between the `Offering` class and all the clients. What if the class internally changes how it represents the number of available spots? Clients would have to be changed as well.

This design problem has even a name: shotgun surgery (https://refactoring.guru/smells/shotgun-surgery). Shotgun surgeries happen whenever a change in one place requires several other places to change. Figure 3.4 illustrates this. You should avoid shotgun surgeries as much as possible.

Figure 3.4 Shotgun surgery



You should encapsulate the state check so that the clients don't have to do any work. Make the `Offering` class offer a method like `boolean hasAvailableSpots()`. How the class implements it internally is not important for the clients anymore. And you can now change the internal implementation of the class as many times as you'd like.

Encapsulating state checks are crucial when they become more complex. You certainly don't want your clients to write complex if statements just so they get the information about the object's state that they need.

Though writing state checks in client code may seem convenient, encapsulating even simple checks can save time and provide flexibility. This approach prevents clients from needing to understand the inner workings of the class, allowing the class to change freely.

3.3.1 Tell, Don’t Ask

"Tell, Don’t Ask" is a principle in object-oriented programming that encourages telling objects what to do instead of asking them for data and then acting on it. You can read more about it on Fowler’s wiki (https://martinfowler.com/bliki/TellDontAsk.html).

We’ve seen that encapsulating state checks within objects is beneficial. However, if clients have to ask the object for information so that they know how to act on it, there’s room for improvement. For example, in the first version of the `Offering` class we implemented in this chapter, we had to first see if there were available spots (ask) and, if so, add the employee (tell). In the second version, clients had to tell the class to add the employee.

The latter approach is better for maintenance and evolution, as clients don’t need to check if the offering has spots before calling `addEmployee`. Moreover, if `addEmployee` requires additional checks in the future, we only need to modify it internally in the method and not in all the clients, avoiding shotgun surgery.

This shouldn’t surprise you, as we discussed similar things previously in this chapter, but now you know the name of a related principle.

3.3.2 Example: Available spots in Offering

If a client of our `Offering` class needs to know if spots are still available, the client has to get the number of available spots via `getNumberOfAvailableSpots()` and see if the number is greater than zero.

Let’s encapsulate this state check better. See listing 3.7. The `Offering` class now has a `hasAvailableSpots()` method, which abstracts away from clients how the check is done. This allows the `Offering` class to freely change its implementation if needed.

The implementation of the method is as simple as a `availableSpots > 0`. It doesn’t matter that it’s that simple. You still should encapsulate it and free the clients from knowing how to do it.

Listing 3.7 Implementing the hasAvailableSpots method

```
class Offering {

  // ...
  private int availableSpots;

  public boolean hasAvailableSpots() { #1
    return availableSpots > 0;
  }

  public int getAvailableSpots() { #2
    return this.availableSpots;
  }
}
```

The `Offering` class is getting much better!

3.4 Provide only getters and setters that matter

Offer only relevant getters and setters to clients. Getters should not modify or allow modification of the class state, while setters should be provided for descriptive properties only. This pattern promotes code clarity and maintainability by limiting the public interface of a class to what is necessary and relevant for clients.

Getters and setters enable clients to access and modify class data. These methods are essential in languages like Java but less so in Python or C#, which offer different functionalities. Regardless of your programming language, you must prevent clients from having unrestricted access to attributes.

If classes can freely modify attributes, how can we ensure consistency? Additionally, if classes can access any attribute, how can we guarantee that future class evolutions won't break clients since they are now coupled to every attribute?

However, you can't write software without offering clients ways to interact with class data. In the following two sections, we'll discuss the characteristics of good getters and setters.

3.4.1 Getters that don't change state and don't reveal too much to clients

Getters should never change the state of the class. This is an essential and unbreakable rule. Command-Query Separation (or CQS) is a principle where methods should either be a command (they perform an action that changes the state of the system) or a query (they return data to the caller), but never both.

While this rule is generally well-followed, it's crucial to think about which attributes should have getters. Some fields may be better kept inside the class or replaced with more elegant getters providing richer information.

The practical challenge with getters is that frameworks often require getters and setters for their functionality. Some developers separate classes used for mapping objects to relational tables from domain classes to avoid breaking invariants. Others prefer a more pragmatic approach, providing the required getter or setter the framework requires, and understanding that the design doesn't fully protect against bad changes. It depends on developers using classes correctly.

Unmodifiable collections

In Java, you can ensure that the returned list of employees is unmodifiable. If you have to offer a getter that returns a list, return an immutable collection.

3.4.2 Setters only to attributes that describe the object

Careless setters can lead to inconsistent objects, as they allow anyone to update the field of a class in whatever way they want. Each setter in the code should exist only after a deliberate decision.

Never offer a setter for an attribute requiring consistency checks. Instead, provide elegant methods that safely perform the operation, like the `addEmployee()` method discussed earlier. We don't want clients to do `offering.getEmployees().add(employee)`.

A safe rule of thumb for setters is when the attribute being changed mainly describes the object, such as a `description` attribute for an `Offering` class or a `name` attribute in an `Employee` class. Setters for descriptive fields usually won't cause future issues. For other fields, consider if allowing changes could lead to inconsistency.

Checks inside setters

Setters can include additional code, like checking for null values before storing them. However, if you need business checks inside setters, consider giving the method a more meaningful name (or moving the operation to a service if the consistency check is complex). This way, your code follows the convention that setters only assign values, making it clear what to expect from any setter in your codebase.

3.4.3 Example: Getters and setters in Offering

The `Offering` class wants to offer clients the possibility to see the list of employees that are on an offering. However, we don't want clients to be able to change that list without going through the proper service.

There are many ways we could implement it. For example, we could create a method that returns a different data structure with the list of employees, say, an `EnrolledEmployees` class. This way, even if clients change the data, it won't be reflected in the entity.

I often use this approach, as it strongly decouples the entities from my model that I need to keep consistent from objects that only carry data. This approach also allows me to return only what the clients need. For example, maybe clients only need the name and the e-mail of the employees. The new data structure can only contain what clients need, decoupling them even more from the entity.

Your programming language may also offer a way to return copies of lists or even immutable data structures that throw exceptions in case clients try to modify them. I'm going to use Java's way of returning an immutable collection. See listing 3.8. The `unmodifiableSet` method from the `Collections` class, part of Java's collections library, returns a list that can't be modified.

Listing 3.8 getEmployees returns an immutable list

```
class Offering {
```

```
  ...
  private List<Employee> employees;

  public List<Employee> getEmployees() { #1
    return Collections.unmodifiableList(employees);
  }
}
```

The `getEmployees` is now a safe getter! Also, note that we don't offer a `setEmployees` method, as allowing a client to pass an entire list of employees at once makes no sense.

List or Set?

The collection of employees could very much be represented as a set instead of a list. After all, we shouldn't have repeated employees enrolled in an offering, and sets offer this capability out-of-the-box.

Such business rules are often better enforced through elegant validation rules as we discussed in this chapter (see listing 3.6 again, we check whether the employee is already in an offering) and also likely ensured by a database constraint. Therefore, in practice, using a set won't bring extra benefits. Choosing a list or a set for this particular problem is then a matter of taste.

Question: should we offer a setter for the `maximumNumberOfAttendants()`? I don't think so. Modifying the maximum number of attendants may involve some logic. What if we have more employees in the offering than the new number? A simple setter isn't enough, you'd need either a proper method in the `Offering` or even a service if the logic gets more complex.

3.5 Model aggregates to ensure invariants in clusters of objects

Design aggregates to ensure consistency of entities that hold clusters of objects. This approach improves code clarity and maintainability, making it easier to reason about entities that handle complex relationships between objects.

In Domain-Driven Design, aggregate roots are clusters of objects treated as a single object. The main object, or root, ensures consistency in the entire object tree. Clients can access and call operations only on the root object and not directly on its internal objects. Clients should hold references only to the aggregate to prevent changes to internal objects without the root's knowledge.

As DDD suggests, modeling aggregate roots should be an explicit part of your design process. Identifying objects within an aggregate root and ensuring they're accessed only through it is a crucial design activity that pays off in software maintenance. If all actions go through the aggregate root, it can maintain consistency throughout the object tree.

Note that aggregate roots go beyond simply "encapsulating lists of objects." It's more than that. It's about modeling entities that should be responsible for keeping the consistency of more complex domain relationships.

Another important rule with aggregates is treating aggregate roots as the unit to be passed around. When persisting an object to the database, pass the entire aggregate root, not a specific internal object. You should therefore have one repository or data access object per aggregate root, not per database entity. This maintains consistency by passing aggregate root references.

Regarding persistence, the aggregate root repository also ensures database integrity. For example, deleting an enrollment from an offering directly in the database using `enrollmentRepository.delete(enrollment)` might break consistency in the offering, as the number of available spots field might now be incorrect. A well-designed `OfferingRepository` wouldn't offer a `deleteEnrollment` method since it could break consistency.

Note

This discussion only scratches the surface of designing aggregate roots. Consider reading books on domain-driven design, such as Eric Evans's original DDD book and Vaughn Vernon's Implementing Domain-Driven Design, for deeper exploration.

3.5.1 Don't break the rules of an aggregate root

Trust me, you will sometimes feel tempted to "skip the aggregate root" and operate directly on one of its child objects. There are different reasons for it, such as:

- Some frameworks or libraries may require it, such as persistence frameworks needing one repository per database entity.
- For performance reasons, you might prefer to access a small part of the aggregate directly rather than through the aggregate root.
- Aggregate roots with deep object clusters may need too much "boilerplate code" for simple changes.

Are you willing to sacrifice consistency and maintainability for other benefits? My advice is to be pragmatic. Use all available tools and software to your advantage, but be aware of the trade-offs.

I don't want to repeat everything that has been said about aggregates already, so I'll ask the reader to read the chapter about Aggregates in Vaughn Vernon's Implementing Domain-Driven Design book. It's a great read. Nevertheless, the personal checklist I go through whenever I feel tempted to break an aggregate root is:

- Is whatever you benefit from breaking the aggregate rules way more beneficial than keeping maintenance costs low and invariants always up-to-date?
- If you feel you want to change part of the aggregate directly without going through the aggregate root, are you sure this object should be part of the aggregate? If there's no real invariant that needs to be kept, break the aggregate into smaller parts.
- If there's an invariant to be kept, can't you still break the aggregate root, and accept that there'll be some eventual consistency between the two aggregates? You can use domain events to make sure that the aggregate is notified about the change in the other aggregate.

3.5.2 Example: the Offering aggregate

PeopleGrow! needs to improve how offerings are handled. Right now, we store the list of employees that are in one offering, and if this person gives up, we remove them from the list. We need to store more information about it. First, the business wants to know the date the person enrolled for the offering. Also, if the person cancels, the business wants to know the date of cancellation.

Storing employees as a simple list isn't enough anymore. We need another entity to store the extra required information. Let's call this entity `Enrollment`. An enrollment will contain the employee, the date of enrollment, the status of enrollment, and, if canceled, the date of cancellation.

See listing 3.9. The `Enrollment` class stores all the information of an enrollment. It also offers a `cancel` method that cancels the enrollment and sets the cancellation date.

Listing 3.9 The Enrollment entity

```
class Enrollment {
  private Employee employee;
  private Calendar dateOfEnrollment;
  private boolean status;
  private Optional<Calendar> dateOfCancellation;

  public Enrollment(Employee employee,
    Calendar dateOfEnrollment) {
    this.employee = employee;
    this.dateOfEnrollment = dateOfEnrollment;
    this.status = true;
    this.dateOfCancellation = Optional.empty();
  }

  public void cancel(Calendar dateOfCancellation) {
    this.status = false;
    this.dateOfCancellation = Optional.of(dateOfCancellation);
  }

  // relevant getters
}
```

We'll make an `Offering` to have multiple `Enrollments` instead of a direct list of employees. We'll also ensure that all changes in an `Enrollment` happen through the `Offering` that `Enrollment` belongs to. After all, we can't allow clients to change enrollments directly. Imagine if a client cancels an enrollment but forgets to update the number of available spots (there's one more now that this one was canceled).

To ensure we have no consistency issues, we make `Offering` to be an aggregate root and `Enrollment` to be one of its aggregates.

See listing 3.10. The `enroll` method (I renamed `addEmployee` to `enroll` as it makes more sense from this new business angle) creates a new instance of `Enrollment`, puts it in its list of

enrollments, and decreases the number of available spots. If someone wants to cancel their participation in a training, `Offering` offers the `cancel` method, which looks up the enrollment of that employee, cancels it, and then makes the spot available again. I won't illustrate the changes in the `AddEmployeToOfferingService` class as they are minimal (mostly call `enroll()` instead of `addEmployee()` and perhaps rename it to `EnrollAnEmployeeToOfferingService` to better reflect the new domain terms).

Listing 3.10 Offering as an aggregate root

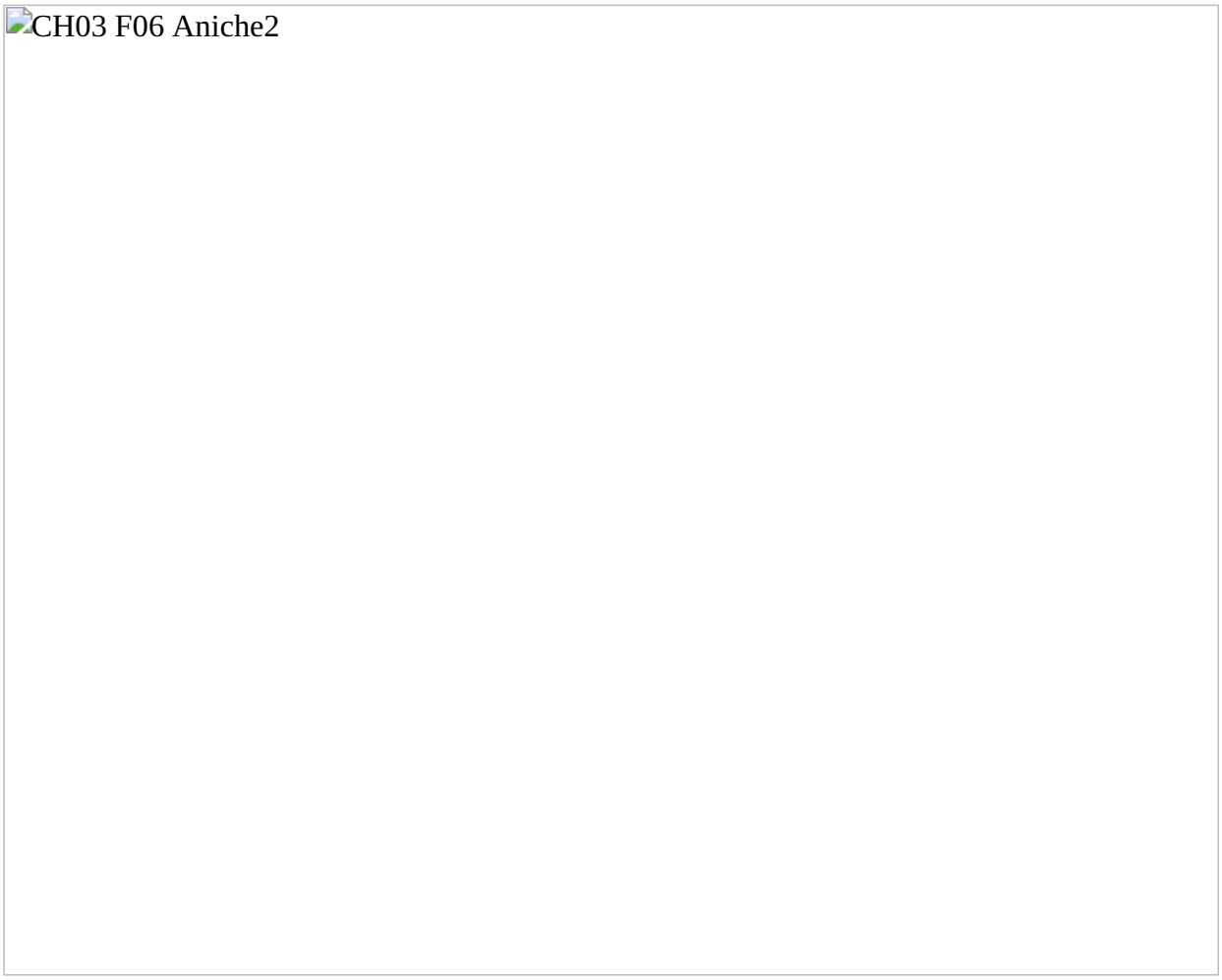
```
class Offering {

  private int id;
  private Training training;
  private Calendar date;
  private List<Enrollment> enrollments; #1
  private int maximumNumberOfAttendants;
  private int availableSpots;

  public Offering(Training training, Calendar date,  int
maximumNumberOfAttendants) {
    // basic constructor
  }

  public void enroll(Employee employee) { #2
    if(!hasAvailableSpots())
      throw new OfferingIsFullException();

    Calendar now = Calendar.getInstance();
    enrollments.add(new Enrollment(employee, now));
    availableSpots--;
  }

  public void cancel(Employee employee) { #3
    Enrollment enrollmentToCancel = findEnrollmentOf(employee);
    if(enrollmentToCancel == null)
      throw new EmployeeNotEnrolledException();

    Calendar now = Calendar.getInstance();
    enrollmentToCancel.cancel(now);

    availableSpots++;
  }

  private Enrollment findEnrollmentOf(Employee employee) { #4
    // loops through the list of enrollments and
    // finds the one for that employee
    // ...
  }
}
```

Note how the `Offering` aggregate root ensures the entire aggregation is consistent. No client should be able to manipulate enrollments directly. Again, you can play with your programming language so clients can't use the `Enrollment` entity directly.

In the future, canceling an enrollment might even become a service, as it may have more complex consistency checks and validation rules. In that case, the service would ensure that the cancellation is valid and then call the aggregate to propagate the cancellation to the `Offering`, just like what we did with adding an offering.

Figure 3.5 illustrates this aggregate root. The `Offering` is the aggregate root, while the `Enrollment` is an internal domain object. All operations in the offering or its internal domain objects must go through the aggregate root. The design should prevent any modification of internal domain objects.

Figure 3.5 An illustration of the Offering aggregate root



Although this is more of a topic for chapter 6, let me talk a bit about my decision in the `cancel()` method. The method finds the enrollment to cancel by looping through the list of `Enrollments`. In practice, we have to load up the entire list of enrollments from the database before finding the one we need. One may argue that this isn't performant enough.

The list of employees in a training is small and PeopleGrow! isn't expected to have an insanely high load. I'm therefore assuming that loading up the list of employees won't ever

cause performance issues. But you never know. We'll come up to this example in chapter 6 when I discuss how to get the best out of your infrastructure without compromising much of your design.

3.6 Exercises

Discuss the following questions with a colleague:

3.1. Have you ever faced a bug caused due to the lack of consistency of an object? What was this bug like? How did you fix it?

3.2. In your opinion, what are the real-world challenges in ensuring that things are properly consistent and encapsulated?

3.3. How have you been designing consistency checks and validation mechanisms in your software systems? How much does it differ from what's presented in this chapter?

3.4. Were you aware of the Domain-Driven Design's concept of aggregates? Do you see opportunities to apply it in your current project? Where? Why? How?

3.7 Summary

- Object consistency is crucial to prevent bugs, reduce coding effort, and ensure smooth maintenance. Consistency should be ensured first in the class, with all of its methods ensuring no invalid changes occur.
- Complex business operations may require external validation, which should be handled by services or dedicated validation classes working with the central entity to ensure consistency.
- Avoid creating getters and setters blindly, and reflect on the needs of each method.
- Design aggregate roots in complex entities to ensure the consistency of the entire object graph and prevent clients from updating the internal state of the aggregate.

4 Managing dependencies

This chapter covers

- Reducing the impact of coupling in the class design
- Depending on high-level, more-stable code
- Avoiding tightly coupled classes
- Increasing flexibility and testability with dependency injection

In any software system, classes get together to deliver more extensive behavior. For example, a service class may depend on several repositories and entities to do its job. This means that the service is coupled to these other classes.

We've discussed the problems of large classes and the advantages of smaller classes. On the one hand, having a class depend on other classes instead of doing everything alone is good. On the other hand, once a class delegates part of its task to another class, it has to "trust" that the other classes do their jobs right. If a developer introduces a bug in an entity, this bug might propagate to the service class and make it break without even touching its code.

That's why you don't want to add more dependencies to a class randomly. Dependency management, or, in simpler words, which classes depend on which classes and whether this is good or bad, is key when maintaining large software systems.

It's easy to lose control of our dependencies. For example, let's make a class depend on the details of its dependencies. Suddenly, any change in the dependency creates a ripple effect of changes throughout the codebase, increasing the system's complexity and making it harder to maintain over time. Or if we make a class depend on many other classes. Besides the code complexity that emerges from the many interactions with the other classes, too many classes may change and affect it. You don't want that.

Managing dependencies is like layering a cake. If you don't do it well, the cake will fall. In this chapter, I'll talk about patterns that'll help you get your dependencies under control.

4.1 Separate high-level and low-level code

Separate high-level behavior code from low-level implementation code to minimize the impact of changes. High-level code should mainly depend on other high-level code, reducing the potential impact of changes to low-level details. By isolating high-level code, you can create a more modular and adaptable system that is easier to maintain and update over time.

Most business functionalities can be seen from both high and low-level perspectives. The high-level perspective describes **what** the functionality should do, while the low-level perspective describes **how** it should accomplish the task.

This separation explicitly in the code is good for maintenance, especially for complex features and business rules. You want pieces of code that only describe the feature (the high-level code) and pieces that concretely implement the feature.

There are advantages when following this pattern. First, when maintaining code, start reading from the high-level code gives you a quicker understanding of the feature, as it only contains the "what" and not the "how." You only dive into the implementation of the lower-level details if you have to. Maintaining code is much easier when it doesn't require you to read hundreds of lines before understanding what it does. By hiding details, developers can focus on what's important.

Second, the separation between higher-level and lower-level code allows them to change and evolve separately. For example, you can change the internal details of the lower level without affecting the higher level. Or the other way around, you may change the higher level without having to change the lower level.

Third, higher-level code tends to be more abstract and, consequently, more stable. Therefore, when you make your code always depend on other higher-level code, you are less likely to be impacted by a change.

You may have heard of the Dependency Inversion Principle (DIP) before, which is just a different name to describe what I just said. This principle states that we should depend on abstractions, not on details. Moreover, higher-level and lower-level classes should depend only on abstractions, not other lower-level classes. While I'm less strict about depending solely on abstractions, separating high-level and low-level concerns is more critical than creating unnecessary abstractions. Some low-level components are stable enough not to require additional abstractions.

4.1.1 Design stable code

When writing higher-level code, you mostly write "stable code," which is good. Interfaces are an example of units of code that tend to be stable over time since they define, in a high-level way, what a component offers to the external world. Interfaces don't care about internal implementation details. Interfaces are a great way to decouple high-level code from low-level code.

Interfaces don't do miracles. Bad interfaces can still be designed. For example, interfaces that aren't stable or leak internal implementation details. You need to design interfaces with stability and information hiding in mind.

4.1.2 Interface discovery

Writing all the high-level code first, and then only later implementing the details, is an interesting programming style, which you get better at it the more you practice.

There are many advantages to coding this way. Not only is there a clear separation between higher-level and lower-level code, but it also prevents you from getting stuck. If the

implementation details were tackled in the order they appear, as soon as a new requirement arises, there would be a need to search for the solution, distracting from the original focus. By coding the high-level functionality first, the implementation details can be tackled later, resulting in a productivity boost.

Coding the high-level interfaces first also enables you to explore what the contract of that other class should be (some authors call it "interface discovery") and how these two classes will interact together. This is a great design tool to ensure your interfaces stay sharp and to the point and don’t contain methods or require information that’s not needed.

Note

The book by Steve Freeman and Nat Pryce, Growing Object-Oriented Systems Guided by Tests, brilliantly illustrates how interface discovery can help you in building maintainable object-oriented design. This book definitely worths the read.

4.1.3 When not to separate the higher-level from the lower-level

Not every feature needs to be separated into higher-level and lower-level code because not all features in a software system are complex.

Mixing the high-level description of what needs to happen and its implementation can be acceptable for simpler features. You can encapsulate the implementation details of the higher-level code using private methods, making navigating to the implementation details easier if needed. As always, as soon as you realize complexity is growing, refactor.

The only thing I suggest you never mix, regardless of the complexity of the feature, is infrastructure and business code. You don’t want your business logic mixed with SQL queries or HTTP calls to get information from a web service. In these cases, you should always have some higher-level interface that describes the what and let the implementation details be implemented in lower-level classes. I’ll talk more about that in chapter 6.

4.1.4 Example: The messaging job

PeopleGrow! has a background job that runs every five seconds and sends messages to users. The code gets unsent messages, retrieves the user’s internal ID from their e-mail, sends the message using the internal communicator, and marks the message as sent (see listing 4.1).

Listing 4.1 A high-level unit of code for the MessageSender

```
public class MessageSender {

  private Bot bot;
  private UserDirectory userDirectory;
  private MessageRepository repository;

  public MessageSender(Bot bot,
   UserDirectory userDirectory,
```

```
   MessageRepository repository) {
    this.bot = bot;
    this.userDirectory = userDirectory;
    this.repository = repository;
  }

  public void sendMessages() {

    List<Message> msgs = repository.getMessagesToBeSent();

    for(Message msg : msgs) { #1

      String userId = userDirectory.getAccount(msg.getEmail()); #2
      bot.sendPrivateMessage(userId, msg.getBodyInMarkdown()); #3
      msg.markAsSent(); #4

    }
  }
}

interface Bot {
  void sendPrivateMessage(String userId, String msg);
}

interface UserDirectory {
  String getAccount(String email);
}

interface MessageRepository {
  List<Message> getMessagesToBeSent();
}
```

Note how high-level this code is. It solely describes what's expected from this job but contains no low-level implementation details of any of these parts. You know that `MessageRepository` returns the list of messages to be sent, but you don't know how. You know that `UserDirectory` gets the user id based on an e-mail, but you don't know how this is done. The same thing for the `Bot`; you know what it does but not how it does.

Any developer that bumps into this code snippet will understand what this job does. Sure, the developer may not know the implementation details, but the question is: do they need to know? You rarely need to know how everything in a business flow works. It's more common that, in maintenance tasks, you need to find the small part of the flow you want to change, and you do it. Imagine how complex software development would be if you needed to understand every single detail of the software system before being able to change it.

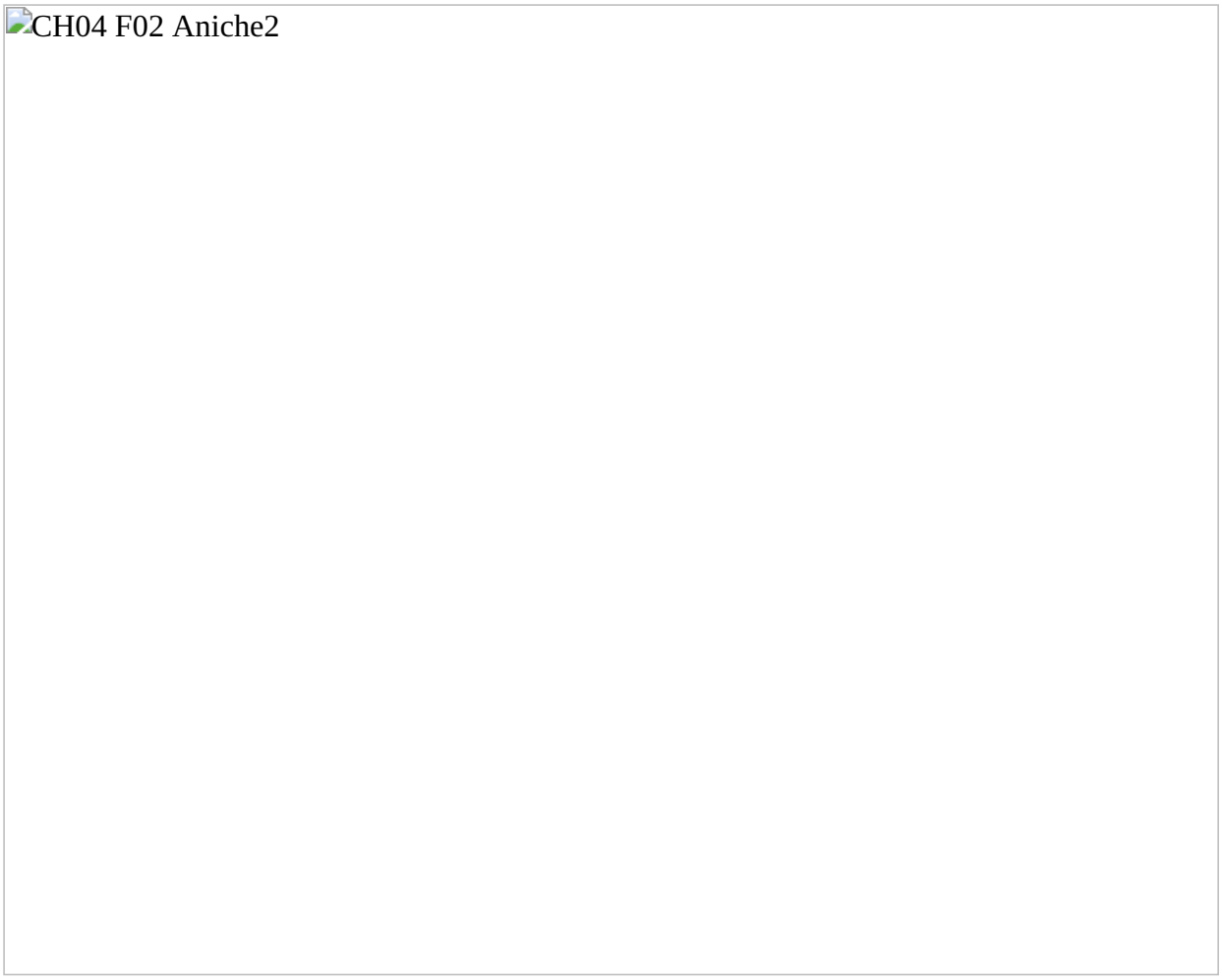
In figure 4.1, you can see that `MessageSender` depends on interfaces that are likely to be stable, such as the `MessageRepository`, `UserDirectory`, and `Bot`.

These interfaces are implemented by low-level code that gets things done. For example, the `MessageRepository` interface is implemented by the `MessageRepositoryHibernate` class, which uses Hibernate (a Java persistence framework) for database access. The `UserDirectory` is implemented by a `CachedLdapServer` class that caches the information retrieved by the company's LDAP server, and the `Bot` is implemented by an `HttpBot` class that makes an

HTTP call to the API. You only know the implementation details once you dive into the lower-level classes.

The implementation details of how the lower-level classes do their jobs are not of concern to the `MessageSender`. It only needs to know that the `Bot` interface offers a way to send a markdown message to a specific user. The `MessageSender` is decoupled from the implementation details, which is what we want.

Figure 4.1 The separation between higher-level and lower-level code.



In terms of coding, the developer that implemented this class followed the suggestion of implementing the higher-level code first (or, in other words, following a more top-down approach rather than a bottom-up). So, she:

- She started to write the entire `MessageSender` class without caring about the details.
- At some point, she needed the list of messages to be sent. The `MessageRepository` already existed, so she added a new method to the interface.
- She then needed to retrieve the user ID based on the user's e-mail. This was the first time this information was needed. She created the `UserDirectory` interface and continued

with the `MessageSender`.
- It was time to send the message to the bot. This was also the first time a bot was required, so she wrote the `Bot` interface.
- Once `MessageSender` was done, she changed her focus to the lower-level classes.
- She implemented the `getMessagesToBeSent()` in the repository.
- She investigated and learned that the user ID should come from the LDAP. She found a library to help her communicate with the LDAP and wrote the class.
- She then read the documentation of the chat tool and learned that a simple HTTP post with the message would be enough. She wrote the code.

As you can see, starting from the higher-level code allowed her to implement the entire business logic without changing focus. Then, it was just a matter of coding the lower-level classes. Well done!

4.2 Avoid coupling to details or things you don’t need

Minimize dependencies on the implementation details of other components to reduce the impact of internal changes. The less you know about how components do their job, the less likely you are to be affected by changes to their implementation.

Rule number one in good dependency management is to never depend on the details of other classes or components. The best way to achieve it is by ensuring that classes don’t expose their details in the first place.

Hiding the internal details of classes is crucial for evolving software components independently without worrying about other components. Imagine having to change hundreds of classes in your system just because you performed a refactoring in a single class. This would be time-consuming and expensive.

In computer science, this principle is also known as information hiding. The idea is to separate what’s likely to change from what’s not so that other components don’t have to be modified too much when we change these parts.

While it’s impossible to hide every detail of a class, we can explicitly design what we expose and what we conceal. Here are some guidelines to help:

- If we change the internal implementation of this class, say, we refactor it, will clients be affected?
- Will this piece of code need frequent changes? If so, can we design it so that the code is hidden behind a more stable abstraction, like an interface?
- Is this the minimum information the client needs to know? Less is better.
- Are we unnecessarily exposing implementation details? If the client doesn’t need to know, don’t reveal it.

In short, we must decide what to expose, what to conceal, and reflect on how much changes to the details we expose will impact clients.

4.2.1 Only require or return classes that you own

When designing classes or interfaces, it’s important to only require or return classes you own, not from a framework or third-party library. By a "class that you own," I mean a class that belongs to your domain model, which you have complete control and ownership over. Say, an entity, repository, or new data structure you created just for this new requirement. By returning classes that you own, you avoid coupling your code to external dependencies, such as a particular library or structure.

The importance of this pattern emerges when we start integrating our code with other modules or third-party libraries. Say, you decided to adopt an SDK of the chat tool your company uses internally. If you pass the classes from the chat SDK throughout your entire code base, you strongly couple yourself to it. What happens if the SDK changes? You’ll either be forced to never update to the newer SDK or propagate the change to the entire codebase, which is cumbersome and expensive.

A way out of this problem is to create classes that represent, from your domain’s point of view, the communication with the chat tool and let one single class in your system handle the conversion between the domain to the chat SDK.

While it may seem like a slight difference, the two have distinct consequences. Now, if the third-party library changes, all you need to do is to propagate the change to the converter class.

While we can’t avoid coupling entirely, we can control what our code is coupled to. However, be cautious not to overdo it and create excessive layers of indirection or needlessly complex code. In some cases, the third-party class is precisely what’s needed, and coupling to it is acceptable. It’s up to you to judge "how bad" the coupling is.

The cost of onboarding developers

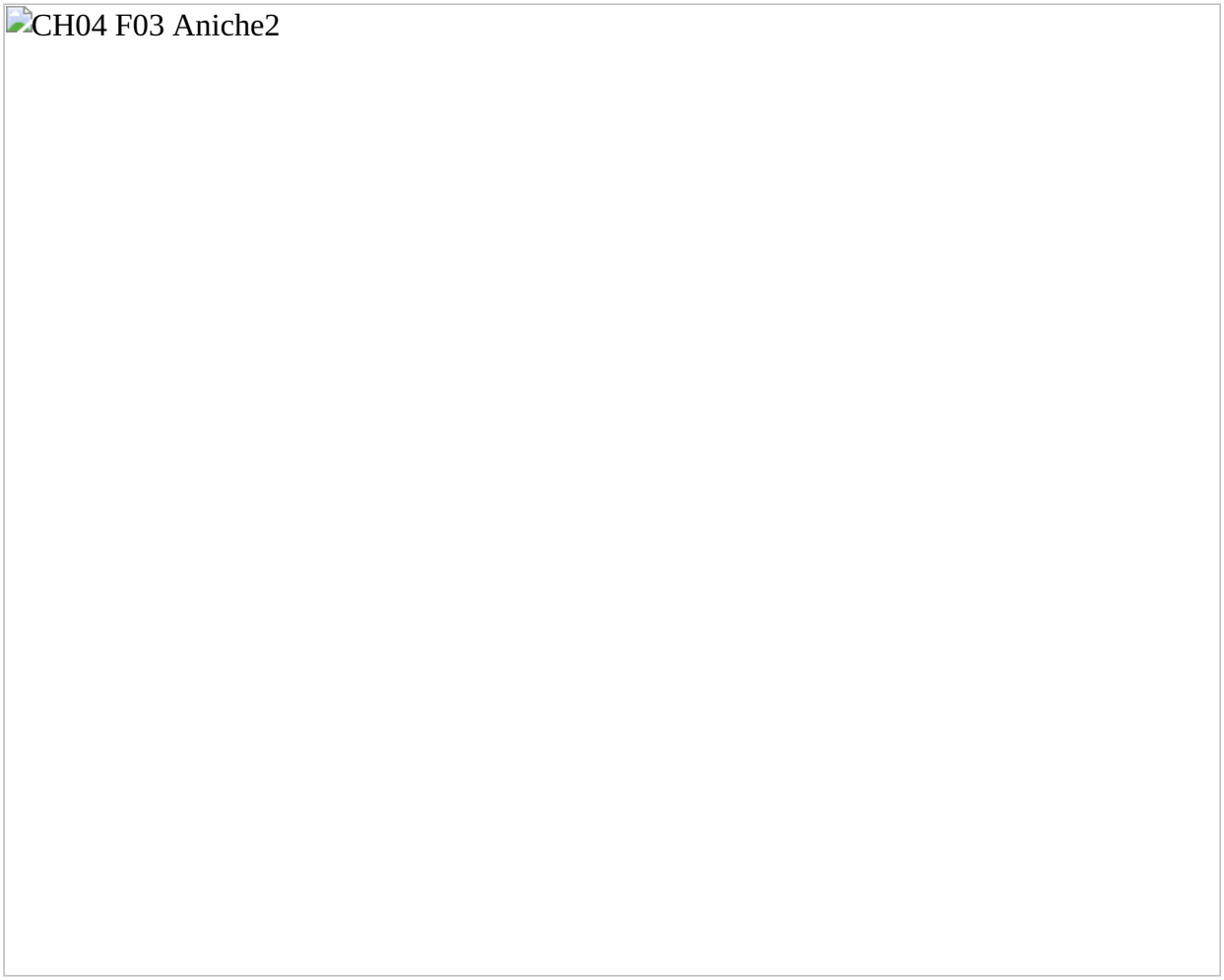
Something else to have in mind when creating wrappers or internal libraries and frameworks is that this will have a cost when onboarding a new developer. It’s likely that the developer knows how to use the wildly spread open-source library, but they won’t know how your unique layer on top of it works. So, make sure your wrapper is simple enough. This idea will appear again in chapter 6, when I’ll talk about how to connect your design with external infrastructure.

4.2.2 Don’t give clients more than what they need

In information systems, it’s common to reuse the same domain entities in different parts of the application. The same entity retrieved from a repository is used by the service and then sent back to the client that requested it (after, say, being serialized to JSON). We do this because it’s too easy to reuse an existing class, even if the client’s needs differ.

I'll focus on this particular example of the same entity being shared across the different layers of the application, as it's the most common type of reuse I see. Figure 4.2 illustrates this.

Figure 4.2 The same entity is used across different layers, creating some undesired coupling.



Giving clients the entire entity, not only what they need, has disadvantages. First, this is propagated to the client whenever you change the entity. Could this break the client? There might also be security implications in this. What if you add a field the client shouldn't see or know about? The worst part is that it's hard for you, as a developer, to do an impact analysis on the fly and identify whether the change to the entity will affect clients.

A great solution to this issue is to decouple the entity from what the clients requested. You can achieve this by creating a more specific data structure that represents the client's needs, then converting the entity to this data structure before sending it to the client. This way, you can change the entity without worrying about how it will affect clients.

4.2.3 Example: Replacing the HTTP bot with the chat SDK

The internal communication system used at PeopleGrow! now offers an SDK for the integration. This means the `HttpBot` can be replaced by a better SDK implementation. Given that the `Bot` interface clearly defines what a bot needs to implement (high-level code), all we need to do is to implement a new one (low-level code).

The SDK offers a class `ChatBotV1` with a method `writeMessage` that receives a `BotMessage` (a class that's part of the SDK) as a parameter. Given that this is a class we don't own we don't let it go outside of the low-level implementation of the new `SDKBot` class we're about to implement. See listing 4.2.

Listing 4.2 The implementation of the SDKBot

```
class SDKBot implements Bot {

  public void sendPrivateMessage(String userId, String msg) {
    var chatBot = new ChatBotV1(); #1
    var message = new BotMessage(userId, msg); #2
    chatBot.writeMessage(message); #3
  }
}
```

In no situation should we return instances of `BotMessage` to other parts of the code. This way, the rest of the codebase is fully decoupled from the library, allowing us to change the implementation in the future.

You may have noticed that I instantiated `ChatBotV1` directly in the `sendPrivateMessage`. A better way to do it would be to inject it into the `SDKBot` class. We'll talk about dependency injection later in this chapter.

4.2.4 Example: The offering list

PeopleGrow! has a page in the web system that lists all the current offerings, their number of enrollments, and total available spots. The `Offering` entity contains all this information, as you see again in listing 4.3.

However, if we return this entire entity to the front-end client, we'll return too much information it doesn't need. This page doesn't need the names of the enrolled employees, for example. In some architectures, returning this list of employees might even have a performance cost as it requires more queries to the database.

Listing 4.3 The Offering class

```
class Offering {

  private int id;
  private Training training;
  private Calendar date;
  private List<Employee> employees;
  private int maximumNumberOfAttendants;
  private int numberOfAvailableSpots;
```

```
  // relevant constructors, getters, and setters
}
```

Instead of returning the full-blown entity, we'll create a data structure that holds only the information that the client needs. See listing 4.4. We have an `OfferingSummary` class contains only the title, date, number of enrollments, and total number of spots. This class is constructed after the main entity.

Listing 4.4 The OfferingSummary class

```
class OfferingSummary {

  private int id;
  private String training;
  private Calendar date;
  private int numberOfEnrollments;
  private int maximumNumberOfAttendants;

  // relevant constructors and getters
}
```

We now only need to build the `OfferingSummary` based on the `Offering` entity. The question is where to put this code. I've seen developers putting it in different places, out of which the best two are:

- A method `OfferingSummary toSummary()` inside the `Offering` entity. This ensures that the conversion logic stays inside the entity.
- A static method `OfferingSummary convert(Offering offering)` in the `OfferingSummary` class. This way, the conversion logic becomes closer to the data structure, freeing the entity from knowing that this data structure exists.

I prefer the second option as I like my domain entities to be agnostic regarding how clients need them. But the first one is also ok and won't give you complicated maintenance challenges.

4.3 Break down classes that depend on too many other classes

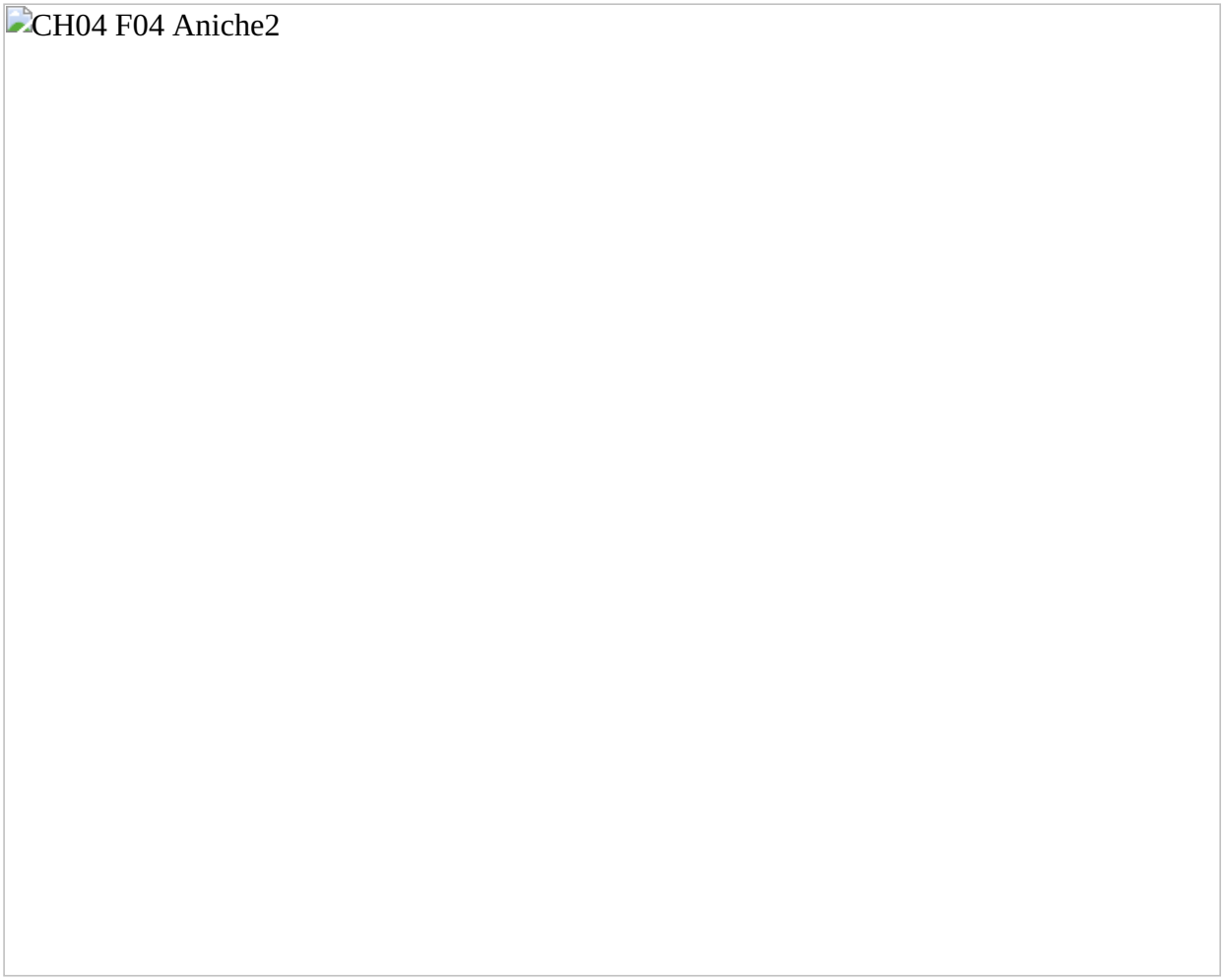
Break down classes with too many dependencies to limit the scope of potential changes. This pattern improves code maintainability and flexibility, allowing your system to better adapt to changing requirements.

Units of code should be small in all dimensions, including dependencies. If a class depends on 10 other classes, it may indicate a design problem and cause maintenance issues in the future.

As features grow more complex, dependency numbers increase. When adding functionality to an existing feature, there are two main options (figure 4.3). One is to expand the current code unit, which doesn't add dependencies but increases complexity, as mentioned in chapter 2. The

other option is to create a new class and link it to the existing one, which raises coupling without increasing the original feature's complexity.

Figure 4.3 Adding code to the same class, or creating a new class? Both decisions have advantages and disadvantages.



When a class starts to depend on many others and its dependencies have grown, consider breaking the cycle. Explore different alternatives, including those that follow.

4.3.1 Example: Breaking down the MessageSender service

Consider the `MessageSender` service in PeopleGrow!. It depends on three classes:

- `UserDirectory`: communicates with the user directory, obtaining user ids from emails
- `Bot`: sends messages to users via the company's internal chat system
- `MessageRepository`: retrieves messages to be sent at a specific time

Now, imagine a new request to deliver messages via email if the user prefers.

The developer finds the existing `EmailSender` class and adds it as the fourth dependency. A fifth dependency, `UserPreferences`, is introduced to obtain user preferences. Listing 4.5 illustrates this.

Listing 4.5 Sending the message also through e-mail

```
public class MessageSender {

  private Bot bot;
  private UserDirectory userDirectory;
  private MessageRepository repository;
  private EmailSender emailSender; #1
  private UserPreferences userPrefs; #1

  public MessageSender(Bot bot,
   UserDirectory userDirectory,
   MessageRepository repository,
   EmailSender emailSender,
   UserPreferences userPrefs) {
    this.bot = bot;
    this.userDirectory = userDirectory;
    this.repository = repository;
    this.emailSender = emailSender;
    this.userPrefs = userPrefs;
  }

  public void sendMessages() {

    List<Message> msgs = repository.getMessagesToBeSent();

    for(Message msg : msgs) { #1

      String userId = userDirectory.getAccount(msg.getEmail());

      bot.sendPrivateMessage(userId, msg.getBodyInMarkdown());

      if(userPrefs.sendViaEmail(msg.getEmail())) { #2
        emailSender.sendMessage(msg);
      }

      // mark the message as sent
      msg.markAsSent();
    }
  }
}
```

A class that was previously dependent on three classes now depends on five. This is not good. We should move code to another class and its required dependencies to regain control.

For instance, the `Bot` interface could be changed from `sendPrivateMessage(user id, markdown message)` to `sendPrivateMessage(Message)`, since the Message object includes the user's email and the message. This change may necessitate updates to the code but could be manageable if the bot isn't widely used.

If modifying an existing interface is costly, try creating a wrapper class that groups dependencies, especially if it can be given a meaningful domain name. For example, a `MessageBot` class could combine the responsibilities of obtaining the user id and sending the message. This class would have a single `send()` method that takes a `Message` object and calls `UserDirectory` and `Bot`. See listing 4.6 for implementation.

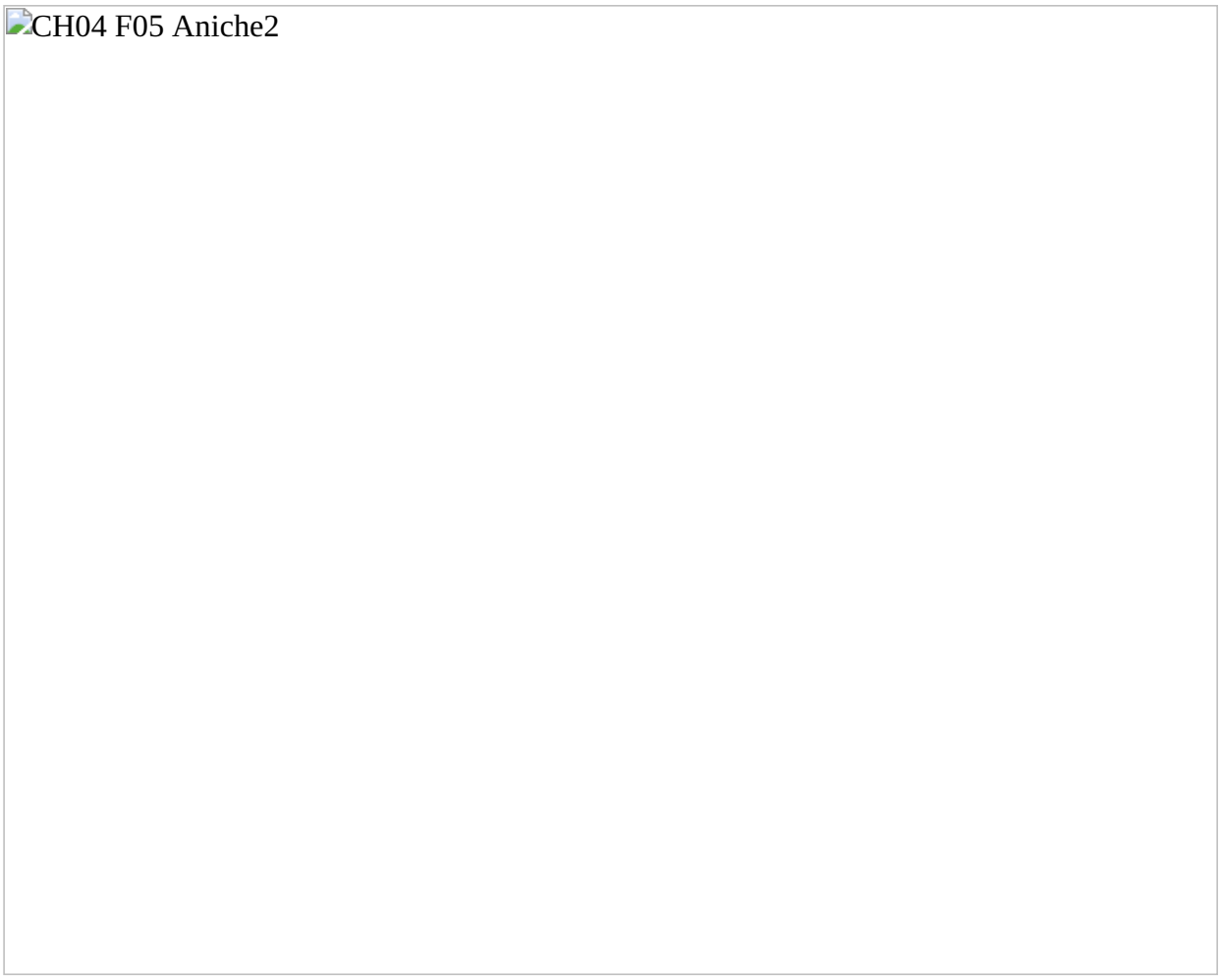
Listing 4.6 The MessageBot class

```
public class MessageBot {

  private Bot bot;
  private UserDirectory userDirectory;

  public MessageBot(Bot bot,
   UserDirectory userDirectory) { #1
    this.bot = bot;
    this.userDirectory = userDirectory;
  }

  public void send(Message msg) { #2
    String userId = userDirectory.getAccount(msg.getEmail());
    bot.sendPrivateMessage(userId, msg.getBodyInMarkdown());
    }
  }
}
```

Introducing this new class has benefits: it reduces client coupling by replacing `Bot` and `UserDirectory` in `MessageSender` with `MessageBot`, simplifying the `Bot` interface. The drawback is managing an additional class in the code.

The "grouping dependencies" tactic could also apply to `UserPreferences` and `EmailSender`. The `EmailSender` could depend on `UserPreferences` to verify user email preferences, reducing the dependency count by one more. Observe the fewer dependencies in `MessageSender` in figure 4.4.

Figure 4.4 MessageSender before and after grouping dependencies



Indirect coupling

In the example, MessageSender depends on `MessageBot`, which relies on `UserDirectory` and `Bot`. Although `MessageSender` doesn't directly depend on `UserDirectory` and `Bot`, it has an indirect dependency.

Although also critical, indirect coupling is less concerning than direct coupling. If the `Bot` class changes, the modification won't affect `MessageSender` since it doesn't directly use the bot. As long as `MessageBot` effectively encapsulates the `Bot` usage, changes are unlikely to propagate beyond `MessageBot`.

4.4 Inject dependencies, AKA dependency injection

Enable dependency injection in components to increase flexibility and testability. By allowing dependencies to be injected, components become more modular and can be easily tested in isolation.

Passing different concrete implementations to a class during runtime gives your design flexibility. We can create as many different implementations of an interface as we want, and the main class will work just fine. We'll explore extensible abstractions in the next chapter, but for now, understand that dependency injection enables compatibility with various bots and user directories.

Another benefit of injecting dependencies is enhanced testability. Developers can easily inject mocked dependencies, which is particularly useful when dependencies have computational costs or extend beyond the application. I won't discuss design for testability, as I did it in my other book, "Effective Software Testing: A Developer's Guide," (https://www.effective-software-testing.com/), also published by Manning, but you get this for free if your classes allow its dependencies to be injected.

The best part of it all is that the implementation is quite simple. All you need to do is create constructors that receive the class's dependencies instead of instantiating them directly.

There are few benefits in hard-coding dependencies and not allowing them to be injected. There used to be a time when teams working on highly performant applications would avoid dependency injection due to its computational costs. These concerns are largely outdated. Dependency injection frameworks are optimized for performance, and virtual machines have improved, handling numerous short-lived object allocations. Unless working at the scale of Google or Facebook, dependency injection performance costs are unlikely to be a problem.

Another common argument was that using static methods was a way to simplify the dependency graph, but this is misleading. Coupling still exists, and now you have less control over it. When dependencies are injected through constructors, developers can easily see a class's dependencies. Identifying dependencies becomes difficult with static methods, as they're dispersed throughout the class's source code.

Interestingly, dependency injection is a typical pattern nowadays. It's due to most frameworks, such as Spring Boot or Asp.Net Core, being quite opinionated.

4.4.1 Avoid static methods for operations that change the state

Static methods can't be replaced at runtime, making the design inflexible and hindering testing. Using static methods as a design pattern can lead to a chaotic, hard-to-maintain system.

The problem with using "static methods as a design pattern" is that the design quickly becomes a big ball of mud. I've seen systems that would perform database access inside static methods. For a static method to call the database, it needs an active connection. Since static methods can call only static methods, you need a static method that returns the active connection. To create a connection, you need all the database information. There you go, another static method to return the database configuration. Before you know it, you are stuck with a set of methods that can't be injected.

Another way to reason about what can be static and what can’t is to look at "how pure" the operation is. A simple utility method that receives a string and returns the number of commas in that string is an example of a pure operation. It doesn’t depend on many other classes, and, more importantly, it doesn’t depend on any external resources. It’s often ok for pure functions to be static methods, as you rarely have to replace them during production or testing.

Conversely, the methods in our previous Service class examples are impure functions. They change the system state (for example, by persisting new information into the database) and may yield different results even with the same input. Such classes and operations shouldn’t be static.

4.4.2 Always inject collaborators; everything else is optional

Classes in information systems often implement different behaviors and collaborate with other classes to deliver complex features. For example, the `MessageSender` class we discussed earlier in this chapter relies on `Bot`, `UserDirectory`, and `MessageRepository` as collaborators, each responsible for a different aspect of the "sending messages to users" feature.

Collaborators should always be possible to inject. After all, collaborators are the dependencies you may want to change in the future (say, change to a new bot) or mock during testing (like the `UserDirectory` because you don’t want to require an entire LDAP server available during testing).

However, not all dependencies are collaborators. Your class may use entities or other data structures representing "information." These aren’t typically injected. It’s more common that classes or services use repositories or factories to instantiate them rather than receiving them from clients. If clients have the entity in their hands, it’s expected that they pass it to other classes via the parameters of a method.

4.4.3 Strategies to instantiate the class together with its dependencies

A pragmatic question around dependency injection is how to instantiate such a deep graph of dependencies. Dependency injection frameworks remove all the cumbersome work of instantiating complex dependencies graphs for you. My favorite option has always been relying on a dependency injection framework such as Spring, Guice, or whatever is available in your programming language. In larger applications, you are likely using a framework to support you in this endeavor, such as Spring or Asp.Net MVC, and a dependency injection framework comes out of the box.

Some developers think that hiding this work from you makes you create even more complex dependency graphs. After all, if you were the one instantiating things manually, you’d see all the work and re-design your code to simplify the graph. While I agree with that, there are other ways to get the same type of information, one that doesn’t require you to do manual work all the time.

That being said, I don’t use dependency injection frameworks for more straightforward applications. I have built a lot of small command-line tools over the past years. In these cases, I prefer factory classes that instantiate my dependency graph rather than configuring a dependency injection framework.

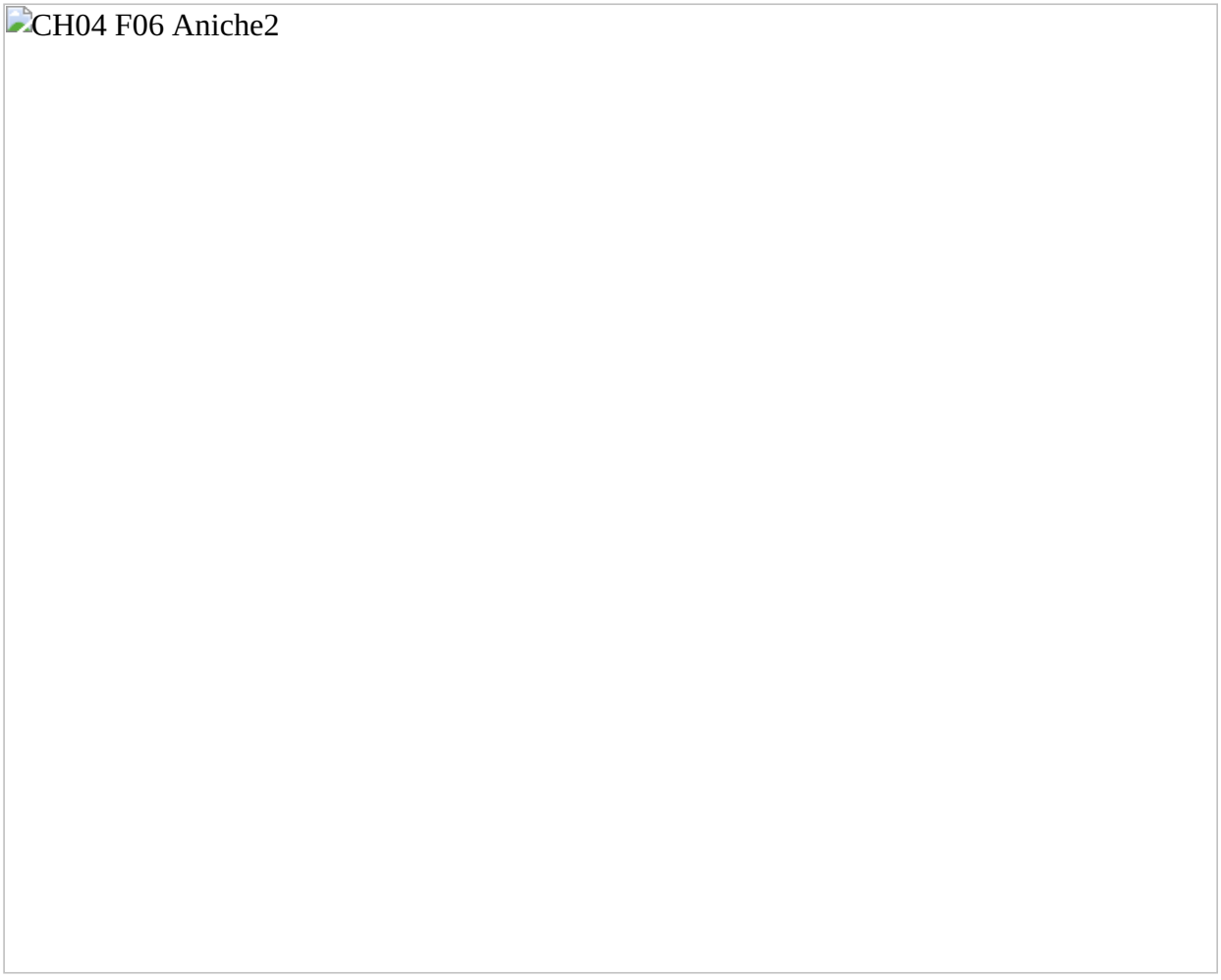
Using a dependency injection framework is, in the end, a decision full of trade-offs. Pick one, get all the benefits of the chosen option, and ensure the downsides are controlled.

4.4.4 Example: Dependency injection in MessageSender and collaborators

The `MessageSender` class in PeopleGrow! was already born with the idea of dependency injection in mind, and so that’s why all its dependencies are injected through the constructor. If we take the idea of dependency injection further down the road, the dependencies of `MessageSender` also require their own dependencies to be injected via the constructor.

Figure 4.5 illustrates the dependency graph when we instantiate `MessageSender`.

Figure 4.5 The dependency graph of the MessageSender class



Luckily, PeopleGrow! makes use of Spring, so the dependencies are automatically injected whenever we need to use `MessageSender`.

4.5 Exercises

Discuss the following questions with a colleague:

4.1. Have you ever suffered from a software system that didn't manage its dependencies properly? What were the consequences? What did you do it, if anything?

4.2. How often do you separate high-level code from low-level code, as explained in this chapter? Do you (now) see advantages in doing it?

4.3. Some developers don't like using dependency injection frameworks. What's your opinion about it?

4.6 Summary

- Minimizing how much a class knows of its dependencies is essential to reduce the impact of changes in the dependencies.
- Separate high-level and low-level code and depend on more stable abstractions. Avoid excessive dependencies and identify opportunities for better abstractions.
- Use dependency injection to increase flexibility and simplify testing. Classes should allow collaborators to be injected.

5 Designing good abstractions

This chapter covers

- Understanding abstractions
- Adding abstractions in code
- Keeping abstractions simple

Good abstractions allow us to add new functionality to a system without changing existing code constantly. For example, think of a bookstore with a range of discounts like "buy three books, get one free," "45% off during Christmas", or "buy five e-books, get one printed copy free." The marketing team proposes new discounts regularly, so the development team needs an easy way to add them to the code. A well-designed software system would have abstractions in place so that developers can add new discounts with minimal effort.

It’s difficult to define what abstractions mean in one sentence, so I’ll use a few of them:

- They describe a concept, functionality, or process in a way that clients can understand without knowing the underlying mechanisms.
- They focus on essential characteristics and ignore non-essential ones.
- Abstractions don’t care (and don’t know) about their concrete implementations.

Abstractions work well with extension points. An extension point enables developers to extend or modify the system’s functionality. In the bookstore, an extension point would allow developers to plug or unplug whatever discounts should be applied for a given basket.

Dijkstra, a remarkable computer scientist, once said, "Being abstract is something profoundly different from being vague. The purpose of abstraction is not to be vague but to create a new semantic level in which one can be absolutely precise."

Designing abstractions is the most fun part of working on object-oriented systems. It involves identifying common characteristics among existing and future business rules or functionalities and expressing them in abstract terms.

Abstractions and extension points are a way to achieve modularity and flexibility in software design, as they allow different parts of a system to evolve independently of each other. However, creating effective abstractions can be challenging. A wrong abstraction can be worse than not having one. To truly facilitate system evolution, we must go beyond "simply creating interfaces for things." Abstractions must be carefully planned and thought through.

Creating good abstractions is like designing the perfect puzzle where pieces nicely fit altogether. In this chapter, I’ll discuss patterns that help you understand when it’s time for an abstraction, how to design good and simple abstractions, and when not to design them.

5.1 Design abstractions and extension points

Create abstractions and extension points in your system to accommodate variability and simplify the addition of new functionality. They improve maintainability and flexibility while minimizing the need to rewrite existing code.

Abstractions and extension points are what enable us to plug new features or variations of existing features into a software system in an easy way. A software system without them forces developers to make existing code more complex whenever a new feature comes in.

It's not uncommon to see classes full of `if` statements, each block handling a different variation of that feature. Or classes with several blocks of code, each handling one piece of the functionality. After all, without proper abstractions, the only way to extend a feature is by writing more code into existing classes or methods, making them even more complex.

As we discussed before, having a class with a couple `if`s or blocks of code isn't problematic. The problem appears once this number explodes. I have once seen a class that had around 40 `if` blocks, one for each variation of the feature, each block containing around 20 lines of code, and many similarities. Regardless to say that the class had no tests and no engineers wanted to maintain this class. A brave engineer at some point refactored this class. His solution was to create an interface that each of the variants would implement. He then implemented something similar to the Template Method design pattern, reducing most duplicated code across variants. Life was much better afterward.

I won't dive into how to implement abstractions or teach design patterns, as the existing literature is extensive already. Rather, I'll talk about when to add an abstraction and pitfalls to pay attention to.

5.1.1 Identify the need for an abstraction

You shouldn't introduce a new abstraction just because "it looks cool." After all, abstractions add flexibility but also complexity to the code. There must be a reason for you to create one.

When is it a good idea to introduce an abstraction? Some of my rules of thumb are:

- Features that require lots of variations may benefit from an abstraction. This way, whenever a new variation appears, all we need to do is to create yet another implementation of the abstraction.
- Features that require flexibility in terms of composability. If a feature has dozens of variations and you may assemble a different combination for each client, then having an abstraction helps you combine different variations without much need for extra code.
- Places where you expect changes in the future. If you know that parts of the feature will likely change, you may do a favor to yourself and facilitate this change through good design.
- Decisions or code you want to hide from the rest of the system. You may use abstractions to prevent details from leaking to other parts of the code. For example, you should hide

database access logic from your domain model. You can do that by introducing an interface that's ignorant of the details.

You may have other reasons to introduce an abstraction. This list is certainly non-exhaustive. Just analyze the trade-offs and see if the extra cost that abstractions bring to the code is worth it.

5.1.2 Design an extension point

In some cases, having clients directly use the newly created abstraction is enough. With the clients depending on the abstraction rather than the concrete implementation, they are decoupled from details that may change. That helps you easily replace one concrete implementation for another without changing a single line in the client's code. For example, the `Bot` interface we created in chapter 4 and that `MessageSender` used. Depending on the interface rather than the direct `HttpBot` allowed us to change the bot to the `SDKBot` without any changes in the `MessageSender` itself.

However, in some other cases, we design abstractions to provide flexibility and variability to a feature. Consider the bookstore from the beginning of this chapter. The abstraction `Discount` was created so that we could apply many of them in the customer's basket. The abstraction isn't enough in this case. We need a mechanism to plug as many varied discount rules as we need to the customer's basket and have them calculate the final price.

Designing an extension point (or not) involves identifying how the abstraction should be used in the wild. You should not only design the abstraction but also put yourself in the shoes of those that'll use it.

Extension points are prevalent in open-source libraries. Think of any framework you use. Say, Spring or Asp.Net MVC. These frameworks provide you with lots of extension points so that you can customize them. Spring allows you to define different security filters to define your own security rules. These are nothing but extension points.

In business applications, although less common than in frameworks, you see them being used in features requiring great flexibility, for example, calculating the final price of the customer's basket after going through complex rules or calculating the salary of an employee based on the many different salary components or rules based on their location, or calculating how much taxes to pay for a given invoice based on the types of products that were sold.

5.1.3 Attributes of good abstractions

Designing good abstractions is an art. What does a good abstraction look like?

A good abstraction separates the "what" from the "how." In other words, good abstractions focus on what they should do but know nothing about how it's done. Good abstractions make you, the developer, not care at all about the concrete implementations. You read the code that uses the abstraction, and that's enough for you to know what's going on.

Good abstractions define a clear contract that clients can rely on. They make it clear to clients what their pre-conditions are and what they promise to deliver. The contracts of a good abstraction are also as lean as possible. They don’t expect from clients anything more than the precise data they require to get the job done, and they don’t return more than what’s needed by the clients.

Once a suitable abstraction is there, writing the different concrete implementations should be as easy as possible. You shouldn’t need to read lots of documentations or spend hours trying to figure out what’s right and what’s wrong. Good abstractions lead you the correct way from the very beginning. That being said, good abstractions are well-documented, and developers can learn everything about it without bothering other team developers.

A good abstraction is easy to be used by its clients. With a few lines of code, you should get the maximum out of the abstraction. You shouldn’t need hundreds of lines of code or meet complex requirements to use it. Good abstractions are straightforward.

A good abstraction doesn’t force extension points to change whenever they change or evolve. A good abstraction is, in fact, stable and doesn’t change much. That’s why coupling to them is less of a problem than coupling to a more unstable concrete implementation.

Finally, a good abstraction allows us to plug and unplug the concrete implementations without requiring any changes in the code. Good abstractions even allow developers to mix and match different concrete implementations to compose more complex behavior in very complex features.

5.1.4 Learn from your abstractions

Designing abstractions and extension points is not an exact science, and even established frameworks sometimes need to evolve their APIs over time. Therefore, it’s natural to expect changes in your code as well.

You should continuously refine abstractions and extension points based on real-world usage. Though valuable for accommodating variability and simplifying code changes, abstractions require ongoing improvement to meet evolving needs. Observe how they are used throughout the software system. Learn from the clients. Listen to the developers. And then improve it.

The challenge is that you can’t keep changing your abstractions and interfaces, as this can cause breaking changes in everyone who depends on it. Maybe the abstraction you are changing isn’t so much used, and you can still afford to change code everywhere. In some other cases, you may have to consider keeping backward compatibility. Changing abstractions safely is a complex software engineering task, maybe even considered still an open problem, and I won’t dive much into it.

5.1.5 Learn about abstractions

One valuable resource for learning more about abstractions and how to design extension points is the Gang of Four’s Design Patterns. Many of these patterns involve creating

extension points for different parts of the system. These patterns can provide a range of ideas for designing extension points. I strongly recommend studying the following design patterns, especially if you're working on enterprise systems: Strategy, State, Chain of Responsibility, Decorator, Template Method, and Command. Although some may not apply directly to your programming language or framework of choice, you can still learn a lot from them.

Another excellent resource is to examine the design of open-source frameworks. Look closely at your favorite framework to see how it designs extension points. Dive deep into their source code. Exploring the source code of frameworks like Spring can teach you a lot about creating extensible software systems.

5.1.6 Abstractions and coupling

In chapter 4, we discussed dependency management and the challenges that high coupling brings to our design. There, I also said that designing stable code is a good practice.

Dependency management and designing good abstractions walk together. A good abstraction tends to be stable. This means that being coupled to an abstraction is less of a problem, as it's less likely to change and as a consequence force other classes to change as well.

5.1.7 Example: Giving badges to employees

PeopleGrow! gives badges to employees that follow the trainings. As with any gamified system, PeopleGrow! has many badges, and the rules to get one range from simple to complex.

The initial implementation of the badge system was as you see in listing 5.1. The `BadgeGiver` offers a `give()` method that receives the `Employee`. The implementation goes rule by rule; if the employee satisfies that rule, they win the badge. `Badge` is a simple enum that lists the available badges.

As you can see, all the badge-assigning rules are implemented in a single class. You can see an attempt to organize the code, as rules are grouped in different private methods, and code comments separate different rules.

Listing 5.1 The first implementation of the badge giver

```
class BadgeGiver {

  public void give(Employee employee) { #1
    perTraining(employee);
    perQuantity(employee);
  }

  private boolean perTraining(Employee employee) {
    TrainingsTaken trainingsTaken = employee.getTrainingsTaken();

    // you get a badge if you did quality-related trainings #2
    if(trainingsTaken.has("TESTING") &&  trainingsTaken.has("CODE QUALITY")) {
```

```
      assign(employee, Badge.QUALITY_HERO);
    }

    // you get a badge if you took all the security trainings
    if(trainingsTaken.has("SECURITY 101") &&  trainingsTaken.has("SECURITY FOR
MOBILE DEVS")) {
      assign(employee, Badge.SECURITY_COP);
    }

    // ... and many more ...
  }

  private void perQuantity(Employee employee) { #3
    TrainingsTaken trainingsTaken = employee.getTrainingsTaken();

    if(trainingsTaken.totalTrainings() >= 5) {
      assign(employee, Badge.FIVE_TRAININGS);
    }

    if(trainingsTaken.totalTrainings() >= 10) {
      assign(employee, Badge.TEN_TRAININGS);
    }

    if(trainingsTaken.trainingsInPast3Months() >= 3) {
      assign(employee, Badge.TRAININGS_ON_FIRE);
    }
  }

  private void assign(Employee employee, Badge badge) {
    employee.winBadge(badge);
  }
}
```

Let’s try a simpler design before going for more complex solutions. We can try a pattern we’ve seen before, breaking a complex class into a few smaller ones.

Let’s see what happens if we move the different groups of badge rules to different classes. As you see in listing 5.2, the new `BadgesForTrainings` and `BadgesForQuantity` classes each contain just the rules related to their group of badges. If a new group of badges emerges, we create a new class. The `BadgeGiver` now coordinates the work.

Listing 5.2 Groups of badges in different classes

```
class BadgeGiver {

  public void give(Employee employee) { #1
    new BadgesForTrainings().give(employee);
    new BadgesForQuantity().give(employee);
  }
}

class BadgesForTrainings { #2
  public void give(Employee employee) {
    // same code for badges for trainings as before
  }
}
```

```
class BadgesForQuantity { #3
  public void give(Employee employee) {
    // same code for badges for quantity as before
  }
}
```

The new implementation is more straightforward due to smaller classes, but it's not a suitable final implementation for two reasons. First, the internal implementation of the new classes, `BadgesForTrainings` and `BadgesForQuantity`, is repetitive and will continue to grow whenever a new badge appears. Second, creating a new group of badges requires modifying the `BadgeGiver`. Although this is a minor issue, improving it can aid in future maintenance.

The first step in improving it is to identify the common behavior we want to abstract, deciding whether a badge must be given to an employee. We can create a `BadgeRule` interface representing any rule to determine the badge assignment.

See listing 5.3. The `BadgeRule` requires two methods from its concrete implementations, `give()` and `badgeToGive()`. The first determines whether or not the employee deserves that badge, and the second returns the badge that should be given.

Listing 5.3 The BadgeRule interface

```
interface BadgeRule { #1
  boolean give(Employee employee);
  Badge badgeToGive();
}
```

We can now implement the different rules, each of them in their own class. See listing 5.4 with the implementation of three of these rules. All classes implement the `BadgeRule` abstraction.

Listing 5.4 A few implementations of BadgeRule

```
class QualityHero implements BadgeRule { #1
  public boolean give(Employee employee) {
    TrainingsTaken trainingsTaken = employee.getTrainingsTaken();

    return trainingsTaken.has("TESTING") &&  trainingsTaken.has("CODE
QUALITY"));
  }

  public Badge badgeToGive() {
    return Badge.QUALITY_HERO;
  }
}

class SecurityCop implements BadgeRule { #2
  public boolean give(Employee employee) {
    TrainingsTaken trainingsTaken = employee.getTrainingsTaken();

    return trainingsTaken.has("SECURITY 101") &&  trainingsTaken.has("SECURITY
FOR MOBILE DEVS"))
  }
```

```
  public Badge badgeToGive() {
    return Badge.SECURITY_COP;
  }
}

class FiveTrainings implements BadgeRule { #3
  public boolean give(Employee employee) {
    TrainingsTaken trainingsTaken = employee.getTrainingsTaken();

    return trainingsTaken.totalTrainings() >= 5;
  }

  public Badge badgeToGive() {
    return Badge.FIVE_TRAININGS;
  }
}
```

In the `BadgeGiver`, now that we have a proper abstraction to represent the badge rules, we make this class depend on the abstraction and not to the concrete implementations. We can do this by receiving a list of `BadgeRule's, say via the constructor. The `give()` method loops through all the rules, and checks whether the badge should be given to the employee, and if so, gives the badge.

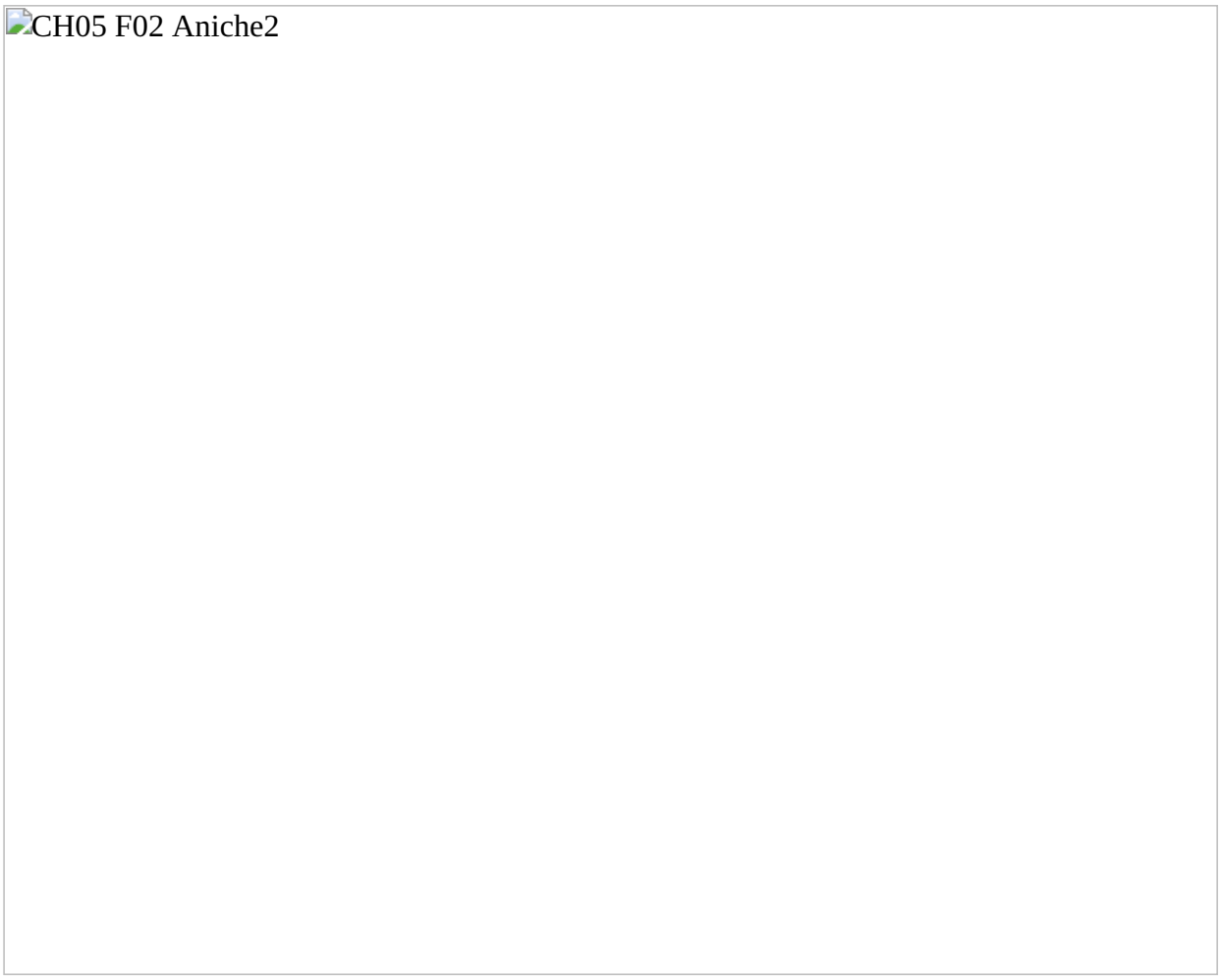
See how the `BadgeGiver` can work with any new badge we create, as long as the rule implements the `BadgeRule` interface.

Listing 5.5 BadgeGiver depends on BadgeRules

```
class BadgeGiver {

  private final List<BadgeRule> rules;

  public BadgeGiver(List<BadgeRule> rules) { #1
    this.rules = rules;
  }

  public void give(Employee employee) {
    for(BadgeRule rule : rules) { #2
      if(rule.give(employee)) {
        employee.winBadge(rule.badgeToWin());
      }
    }
  }
}
```

Figure 5.1 illustrates what our design looks like now. The `BadgeRule` is an abstraction, and badges implement it. Badges are then plugged in to `BadgeGiver` which processes them all.

Figure 5.1 The BadgeRule abstraction and the BadgeGiver.



With this new design, we reached our first goal: we don't have to change the `BadgeGiver` if a new badge emerges. We need to implement a new `BadgeRule`.

The next problem to solve is to avoid the explosion of classes. With this design, we need a new class for a badge, even if the badge is similar to an existing one. For example, the badges we give if the employee follows the quality and the security-related trainings are very similar, with the only difference being the trainings we look for—the same for the badges we give based on quantity. There's a lot of code duplication happening. We'll improve this after we discuss the following pattern in this chapter: generalizing business rules.

Although unrelated to abstractions, another exciting decision we took in this code that I want to highlight is the `TrainingsTaken` class. Note how the method `employee.getTrainingsTaken()` doesn't return a simple list of trainings, but a `TrainingsTaken` class. This class is just a wrapper on top of the list of trainings that encapsulates complex query logic that clients may ask, for example, was this training taken, or how many trainings did the person take in the past 3 months. Listing 5.6 illustrates its implementation.

Listing 5.6 The TrainingsTaken class

```
class TrainingsTaken {

  private final List<Training> trainings;

  public TrainingsTaken(List<Training> trainings) { #1
    this.trainings = trainings;
  }

  public boolean has(String trainingName) { #2
    ... find whether the training is in the list of trainings ...
  }

  public int totalTrainings() {
    ... return the number of completed trainings ...
  }

  public int trainingsInPast3Months() {
    ... return the number of completed
    ... trainings in the past 3 months
  }
}
```

Creating classes on top of lists of objects is something I do, especially if you expect clients to query this list in multiple different ways.

5.2 Generalize important business rules

Generalize business logic to create multiple variants of the same rule without introducing unnecessary complexity or duplication. This approach facilitates the creation of flexible and scalable code that can adapt to changing requirements and improves code reuse by eliminating redundant code.

Sometimes you must apply the same business rule in different contexts with varying concrete values. Without a proper abstraction, developers may duplicate the original code to account for these slight differences.

If we go back to our bookstore example, suppose a specific discount that's based on the authors of the books the customer is buying. For example, if the customer buys books from Kent Beck and Martin Fowler, renowned software engineering authors, she gets a discount. She also gets a discount if she's buying books from Tolkien and Rowling, two of my favorite fiction authors. Now, suppose that the bookstore has 50 or 60 discounts based on different combinations of authors.

A very naive approach to implement this is by writing a series of `if` statements, each checking whether the authors match the customer's books and, if so, applying the discount. Note how these `if` statements pertain to the same business rule but with slight variations. Whenever a new discount is added, the developer copies and modifies the business rule, which is not maintainable.

You should identify the general business rule and create an abstraction to solve this issue. Each concrete implementation of this discount is a different instance of the generic business

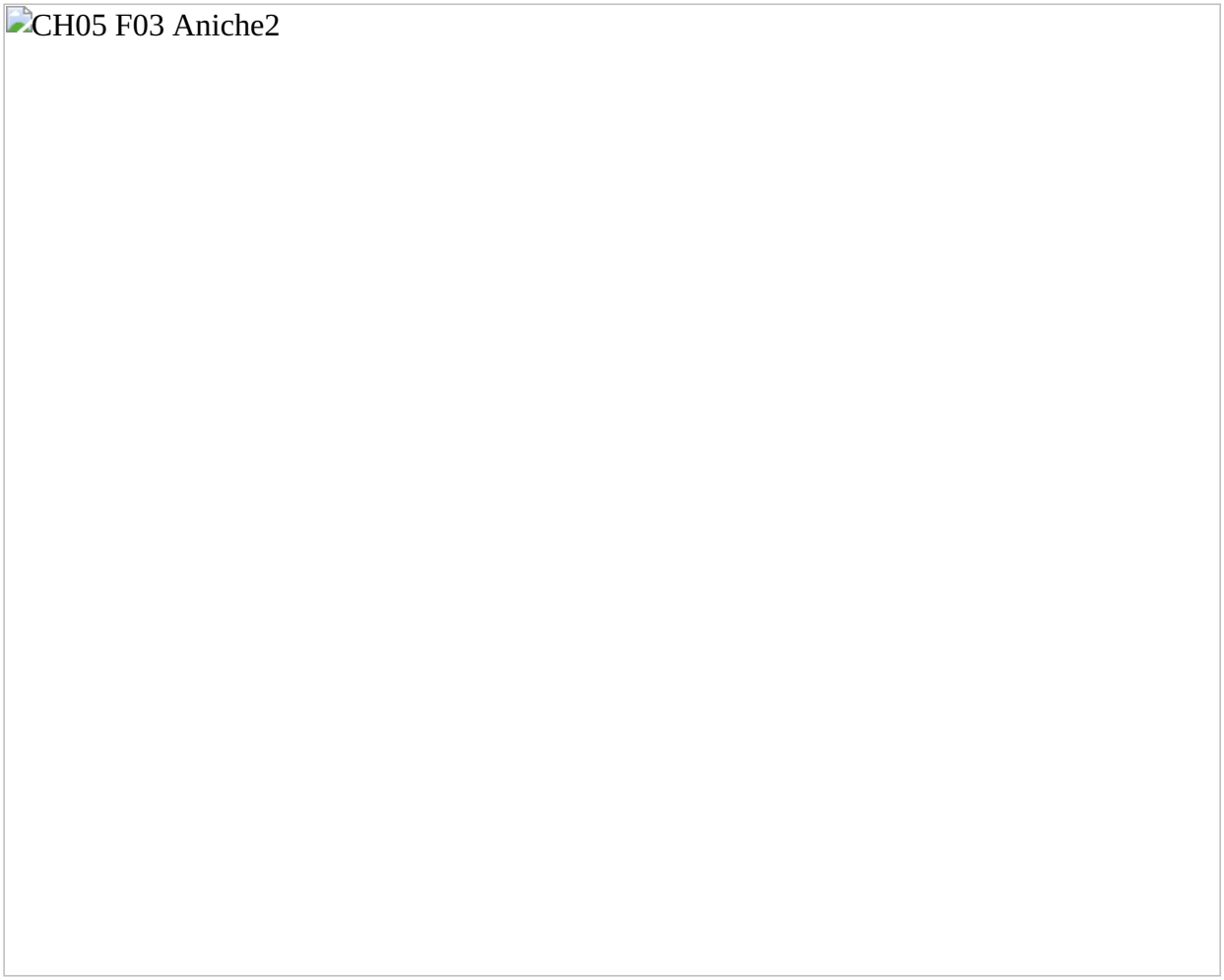
rule but with concrete values.

In essence, this pattern isn’t that different from the previous one. Both are about identifying the right abstractions to enable us to evolve the software system with minimal effort. In my experience, it’s easier for developers to create extension points or abstractions that resemble "different strategies," such as different ways of calculating discounts. Generalizing business rules is harder, so I have a dedicated pattern for it.

5.2.1 Separate the concrete data from the generalized business rule

The main challenge around generalizing business rules is related to data. In most information systems, business-related data always comes from a database. Without a proper design, mixing business rule and data retrieval concerns is too easy, making the code too complex and hard to follow. Figure 5.2 illustrates that.

Figure 5.2 Code that mixes business logic and data retrieval quickly becomes messy, hard to maintain, and hard to test.



Modifying and adding new features becomes challenging when we mix data retrieval and business logic in our code. The code becomes too specialized for one rule, making it challenging to add new variations or enhance its flexibility. In these situations, it’s better to segregate the code that fetches data from the code that executes the business logic, just like we did before.

When generalizing business logic, you should avoid tying them to specific data. Instead, you should ensure that the abstraction doesn’t depend on specific values but on generic ones. For example, instead of having "JK Rowling" and "Tolkien" hard-coded in the code, the generalized business logic should deal with a list of authors' names, whoever these authors are.

There are many different ways to implement this. Some of them are more manual, some fancier. For example, having a class that implements the generalized business rule and receives the data through its constructor and another class responsible for retrieving the data (via some repository) and instantiating the generic business rule with the concrete data is often a good enough solution.

The advantage of a simplistic "glue code" is that it’s easy to understand and evolve. However, the disadvantage is that it can become complex quickly when new discount types emerge.

Deciding how far you should go when designing abstractions is always a challenge. As always, there are no rights and wrongs, just different design decisions with different advantages and trade-offs. Should you go for the most flexible solution immediately or start simple? We’ll talk more about this soon.

5.2.2 Example: Generalizing the badge rules

We have too much duplication across the different badges. For example, `QualityHero` and `SecurityCop` are basically the same in terms of code and structure. We should generalize them. In both cases, the badge is assigned if the participant has taken a selected list of trainings. Therefore, our generalization needs a list of trainings and the badge that should be assigned.

This is what we do in listing 5.7. See the `BadgeForTrainings` class that does what we just described. It receives the list of trainings and the badge to give in its constructor. The `give()` method loops through the list of trainings and checks whether they were all taken. The `badgeToGive()` returns the badge that was provided through the constructor. To instantiate the concrete quality hero badge rule, we need to instantiate `BadgeForTrainings` and provide it with the list of trainings that this badge requires, in this case, testing and code quality.

Listing 5.7 The BadgeForTrainings implementation

```
class BadgeForTrainings implements BadgeRule {

  private final List<String> trainings;
  private final Badge badgeToGive;

  public BadgeForTrainings(List<String> trainings,  Badge badgeToGive) { #1
```

```
    this.trainings = trainings;
    this.badgeToGive = badgeToGive;
  }

  public boolean give(Employee employee) {
    TrainingsTaken trainingsTaken = employee.getTrainingsTaken();

    return trainings.stream()
      .allMatch(training -> trainingsTaken.has(training)); #2
  }

  public Badge badgeToGive() {
    return badgeToGive; #3
  }
}

var qualityHero = new BadgeForTrainings( Arrays.asList("TESTING", "CODE
QUALITY"), Badge.QUALITY_HERO); #4

var securityCop = new BadgeForTrainings( Arrays.asList("SECURITY 101",
"SECURITY FOR MOBILE DEVS"), Badge.SECURITY_COP); #5
```

We can apply the same strategy for the badges based on quantity and training. See the implementation of `BadgeForQuantity` in listing 5.8. Similarly to `BadgeForTrainings`, the constructor receives the number of trainings the employee must have completed to get the badge and the badge. The `give()` checks whether this has happened.

Listing 5.8 The BadgeForQuantity implementation

```
class BadgeForQuantity implements BadgeRule {

  private final int quantity;
  private final Badge badgeToGive;

  public BadgeForQuantity(int quantity,  Badge badgeToGive) { #1
    this.quantity = quantity;
    this.badgeToGive = badgeToGive;
  }

  public boolean give(Employee employee) {
    TrainingsTaken trainingsTaken = employee.getTrainingsTaken();

    return trainingsTaken.totalTrainings() >= quantity; #2
  }

  public Badge badgeToGive() {
    return badgeToGive;
  }
}

var fiveTrainings = new BadgeForQuantity(5, Badge.FIVE_TRAININGS); #3
```

With the generalized badge rules, the next step is to assemble the final `BadgeGiver` with all the concrete rules. Before, when we had one class per rule and lots of duplication, instantiating them wasn't that hard. If the system were using a dependency injection framework, all we needed to do was to ask for all implementations of `BadgeRule`, and the framework would find

them. However, now, with the generalized versions, we need smarter logic to instantiate the concrete rules and get the data from the database.

There are different ways to achieve that, ranging from simple, less flexible, and with some minimal human intervention up to more complex but more flexible and automated. One approach would be to create a Factory method (https://refactoring.guru/design-patterns/factory-method), where we create factories for each type of badge rule, and these factories are responsible for instantiating them.

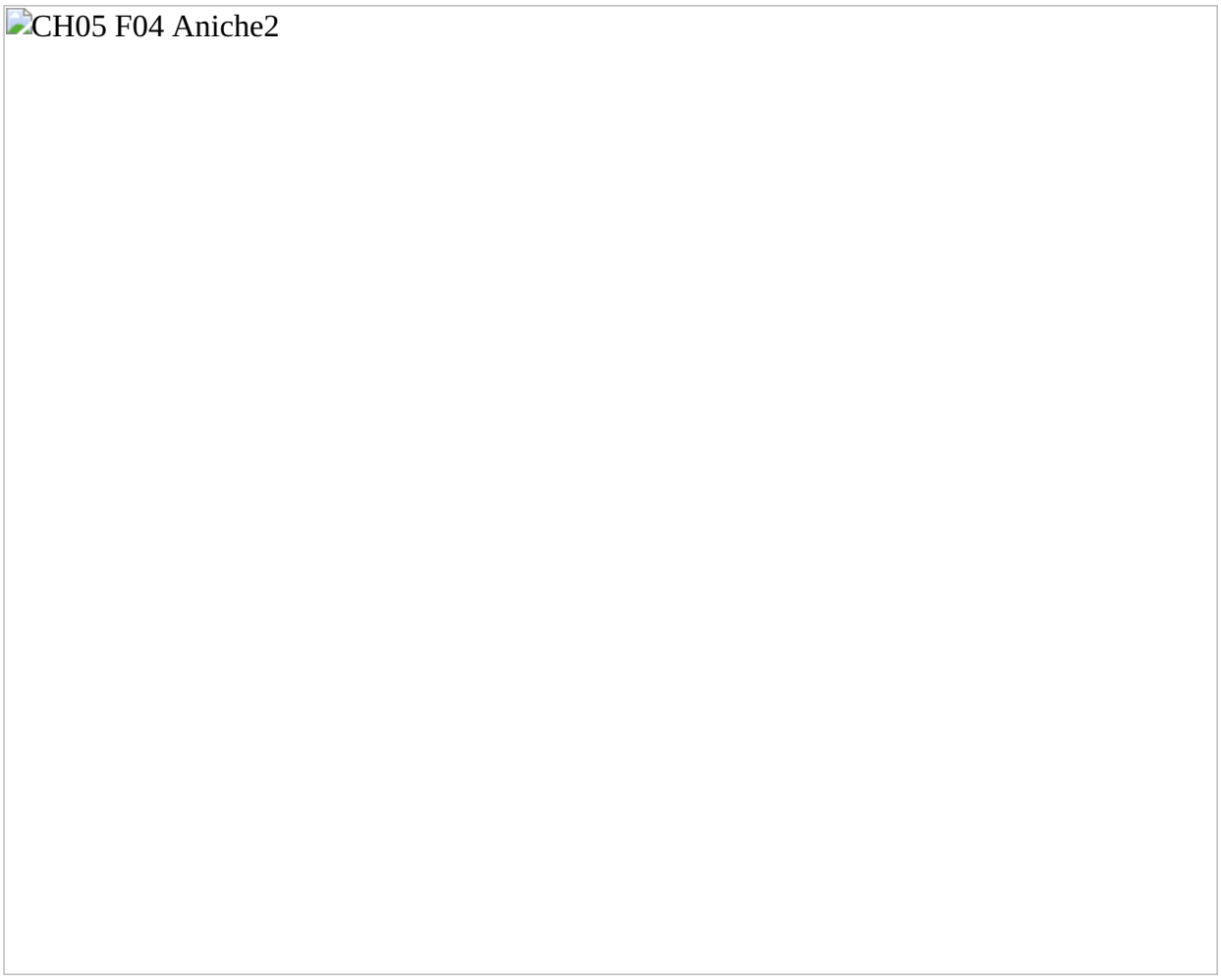
Listing 5.9 shows the factories for the `BadgeForTrainings` and `BadgeForQuantity`. See the `BadgeRuleFactory` that describes what a factory should look like. Then, each concrete implementation, `BadgeForQuantityFactory` and `BadgeForTrainingsFactory`, gets their data from the database and returns a list with all the concrete rules.

Listing 5.9 BadgeRule factory

```
interface BadgeRuleFactory { #1
  List<BadgeRule> createRules();
}

class BadgeForQuantityFactory implements BadgeRuleFactory {

  public List<BadgeRule> createRules() { #2
    // access the DB, gets the data
    // and for each of them, instantiates a BadgeForQuantity class
    // ...

}

class BadgeForTrainingsFactory implements BadgeRuleFactory {

  public List<BadgeRule> createRules() { #3
    // access the DB, gets the data
    // and for each of them, instantiates a BadgeForTrainings class
    // ...
  }
}
```

Figure 5.3 illustrates the final class design of the badge rules. Observe how easy it is to extend the system with more types of rules. All you have to do is to create a new implementation of `BadgeRule` describing the high-level business rule of that badge, and a `BadgeRuleFactory` which gets the current concrete rules for that badge.

Figure 5.3 The final class design of BadgeRule and BadgeRuleFactory



With a little creativity, you can even make your design to force developers to create a factory whenever a new badge rule is created. This is one example of how to do it, but as I said before, it’s ok to start with simple glue code and make it more sophisticated later if needed. We also haven’t talked about infrastructure so far (this is the topic of chapter 6), but if your design doesn’t allow for factories to access the database, then you can add one extra layer between the factory and the database access.

Good abstractions equal easier maintenance!

5.3 Prefer simple abstractions

Abstractions should be simple and require their implementations to do as little work as possible. Simple abstractions simplify the creation of new concrete implementations and reduce the overall complexity of the code that may accumulate over time when the number of implementations grows.

Abstractions add complexity to code. It’s harder for developers to follow a flow full of interfaces and polymorphic calls. But this is the trade-off you make in exchange for more

flexibility. You should strive to simplify the abstraction as much as possible by letting it represent only the bare minimum behavior and delegating the rest to concrete implementations.

Keeping your code simple is always the best option. If there's no clear reason to abstract the code, don't do it. If you aren't sure how much flexibility you need, starting with a bunch of `if` statements might be good enough.

It is, however, hard to define what "simple" means. Unfortunately, it's common to see code lacking abstractions under the excuse of simplicity. For example, code full of `if` statements or long classes to avoid the creation of an interface. Sandi Metz, a famous Ruby developer who also authored books on object-oriented design, once said that duplication is cheaper than wrong abstraction (https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction). While this is true, remember that duplication is also more expensive than a good abstraction.

You shouldn't fall into this trap. Simple and small code isn't enough in places an abstraction is needed.

5.3.1 Rules of thumb

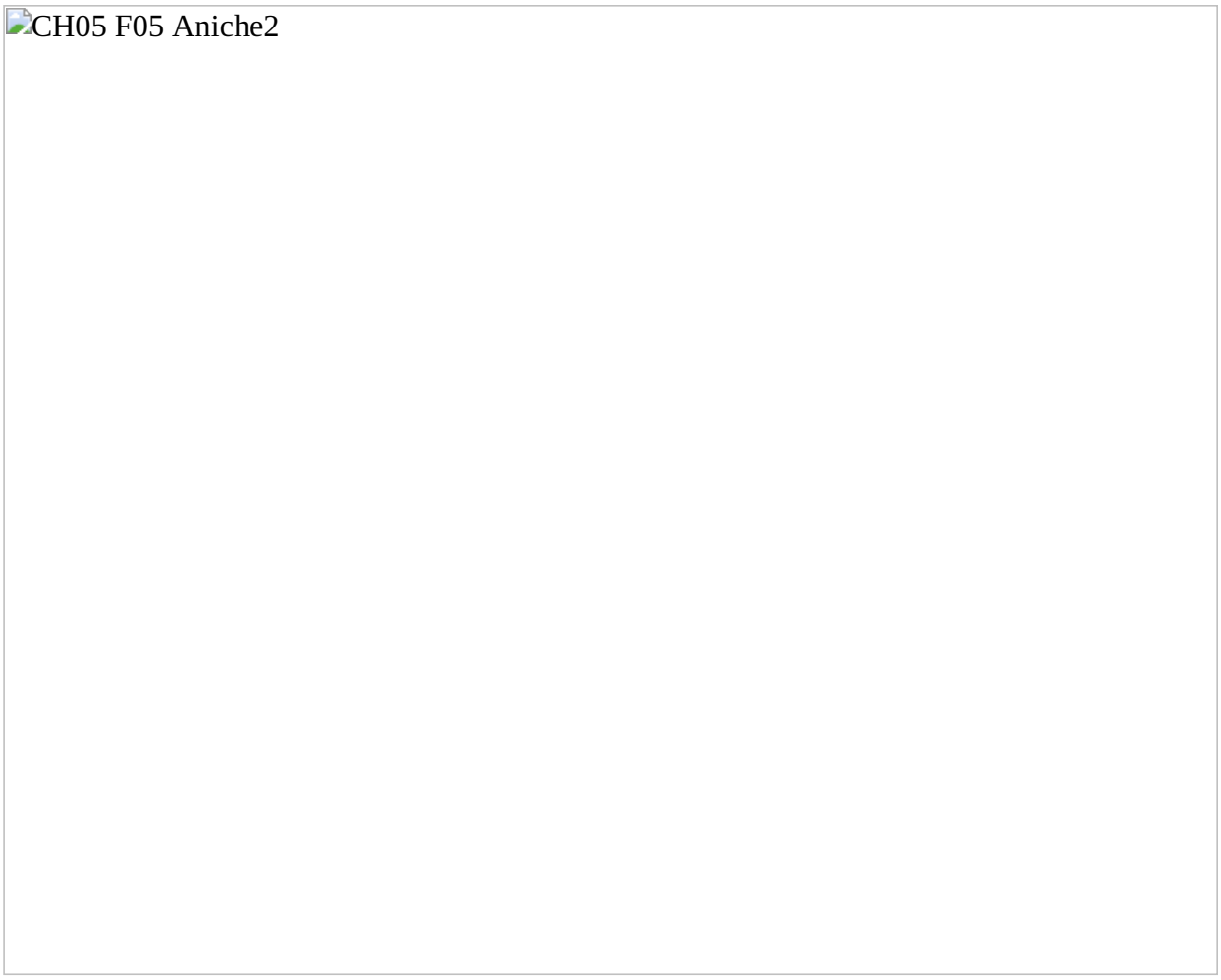
Choosing the best design can be tricky as it often involves trade-offs. Ultimately, it's up to you to decide which trade-offs are most suitable for your specific problem.

Although extension points and abstractions can make adding new functionality to a system easy, they can also add complexity to the code. It's essential to weigh the benefits against the costs to ensure that an abstraction is worthwhile.

It's not always easy to anticipate how much flexibility a system will need. Sometimes we implement a simplistic design that could use more flexibility, or overcomplicate a design that doesn't need it.

To help you decide when to create extension points and abstractions, I suggest following three rules of thumb, illustrated in figure 5.3.

Figure 5.4 The three rules to help you decide whether or not to create an abstraction.



5.3.2 Enough is enough

If you are in doubt whether or not an abstraction is needed, you can "wait for empirical evidence." For example, when implementing the first discount rule for the e-book store, you don’t need more than simple code. You are then asked to implement the second discount rule, ok, maybe two’s all you ever need. When discount rule number 3, 4, 5 comes in, and you start to write many `if` statement in a row, it may be time to re-think and go for an abstraction.

Other things that I pay attention to decide if it’s time for an abstraction:

- Do I keep changing to the same class over and over?
- Do classes keep getting bigger and bigger?
- Am I constantly making use of `if` statements to implement variability?
- Do I keep finding clunky ways to glue existing business rules to other parts of the system?

This is far from being an exhaustive list, but hopefully, it gives you a good idea of things to look out for.

Deciding when enough is enough is relative and a decision full of trade-offs. If you make an early decision, there's a chance you are over-engineering. If you take too long, it can require more effort for you to refactor the existing code.

5.3.3 Don't be afraid of designing abstractions from day one

Sometimes you are confident from the very beginning that you'll need flexibility. In these cases, don't be afraid of proposing an abstraction or extension point from day one, even if you have only one or two implementations at the beginning.

If you know this will happen, and you still opt for starting with simple code, you might be only delaying the inevitable. Such a decision can cost you more than a flexible solution immediately.

5.3.4 Example: Revisiting the badge example

The design we created around badges in PeopleGrow! followed the ideas of this pattern:

- The `BadgeRule` interface is straightforward. It's easy to create any new implementations. The interface requires minimum information from the clients (just the employee under evaluation) and returns only what the clients of the abstraction need (whether to give the badge and the badge).
- The generalization of specific badge rules, such as badge for trainings (implemented in the `BadgeForTrainings` class) and badge for quantity (in `BadgeForQuantity`) are also simple in terms of implementation. They are decoupled from each other and won't ever change in case a new badge rule emerges or because of a change in another badge rule.
- The `BadgeGiver` can work with any badge rule and doesn't have to change if new badges are plugged or unplugged.

The fun part of designing abstractions is that there are many different ways to do it for the same problem. If you were to do this yourself, your final design would likely be different from mine. Which one is better? Yours or mine? Hard to say, and that's why we should always keep learning from our abstractions in the wild.

5.4 Exercises

Discuss the following questions with a colleague:

5.1. How often do you encounter well-designed extension points in your current software system? What do they look like?

5.2. Are there parts of your current project that would deserve a better abstraction or extension point? What part? Why? How would you design it?

5.3. Have you ever suffered from a badly designed abstraction? What did it look like? Why was it bad?

5.5 Summary

- Abstractions and extension points simplify software evolution and prevent code complexity over time by allowing new functionality to be added without changing existing code.
- These design elements can be applied in various aspects of the system for flexibility, such as adding discount rules or generalizing specific rules for multiple instances.
- Simplicity should be a priority when designing abstractions and extension points to avoid adding unnecessary complexity to the code.
- Introducing an abstraction early on can be beneficial, as not having one when needed can be costly.

6 Handling infrastructure

This chapter covers

- Decoupling infrastructure code from domain
- Understanding how far we should go when decoupling infrastructure
- Creating wrappers on top of infrastructure libraries and data structures

Software systems rarely exist in isolation; they often interact with databases for data persistence or web services from external companies or internal teams. A significant challenge in software design is to prevent our code from being contaminated by these "outside details."

You may wonder why this is a problem. There are a few reasons why you should protect your domain from external influences. First, it can impede your ability to replace a component with something simpler, which would facilitate testing. For instance, if you lack a layer between the code that accesses an external web service and your domain logic, it becomes difficult to test the domain logic in isolation.

Second, your code becomes tightly coupled to the data structures and abstractions of third-party APIs. Many libraries provided by companies are not particularly stable and undergo frequent changes. You don’t want minor updates to those libraries affecting numerous places in your code.

Third, handling infrastructure involves low-level code, and without proper encapsulation, making changes becomes cumbersome. Consider a caching layer: you may start with a straightforward implementation using an in-memory hash map for quick access. However, as your application grows and requires more advanced caching strategies, you’d prefer to implement a new approach without needing to modify every instance where the old caching is used.

You might now think that all you need to do is add an interface to abstract the infrastructure, and you’re done. While this is a good starting point and often sufficient, I’d like to delve deeper into the topic.

A well-designed abstraction should hide the details of the infrastructure, allowing the rest of the system to remain largely unaware of it. However, there’s an interesting trade-off between proposing a simple interface that completely abstracts away the infrastructure (enabling easy evolution or even full replacement) and losing the ability to leverage specific features of the underlying infrastructure.

For example, you may want to take advantage of the performance of your Oracle database or the scalability of Amazon SQS queues. However, doing so might require a specific implementation that wouldn’t apply to any other database or queue. The challenge lies in

creating abstractions that allow you to harness the benefits of the underlying infrastructure while avoiding contamination of your code with these specific details.

Furthermore, although I've used external systems (databases, caches, web services) as examples thus far, similar challenges can arise within your software system. Perhaps you're using an opinionated framework that imposes a coding style that isn't ideal for your domain code. Your design should shield you from such limitations as well.

Designing a solid infrastructure abstraction is akin to organizing the plumbing in your house. It involves ensuring organization, ease of maintenance, and replaceability. In this chapter, we'll explore how to create abstractions that'll help you make your code independent of specific external systems so that you end up with designs that don't break when major changes happen in them.

6.1 Separate infrastructure from the domain code

Code that handles infrastructure should be separated from the domain code. These classes should be as thin as possible and contain no business logic. This separation keeps the design clean, easier to evolve, and more testable.

Don't write any infrastructure-handling code inside any class with business logic. This is rule number one when it comes to handling infrastructure code — and a no-brainer because it's easy to follow and has high gains. Instead, write the handling code in a class whose sole purpose is to represent the communication between your application and the external system.

Even before diving into the lengthy discussion of whether we need an interface or if the concrete class representing the external system is enough, you already improved your design by just putting domain code and infrastructure code in different classes by one order of magnitude. Your code is now more testable because it's easier to replace the concrete class with a fake during testing, and any changes in the infrastructure code happen in a single place.

Separating infrastructure code from domain logic is common in more modern codebases. It's been years since I've seen a system that contains SQL queries and database logic code mixed with business-rule code. Patterns like Data Access Objects (DAO) and Repositories (from the Domain-Driven Design book) are prevalent in most code bases.

However, the devil lies in the details. Let's say you need to retrieve a simple configuration value from a file to process business rules. It's common to see file access and business logic mixed together. While this may not appear to be a major issue, you've inadvertently coupled your code to the file's structure. This can complicate your life if the file needs to evolve because you'll have to locate all the places that read the file and make corresponding changes. Moreover, it becomes more challenging to test your code as you can't easily replace the configuration value to facilitate testing. To avoid these problems, you could introduce a `Configuration` class that encapsulates the logic of reading the configuration value from the file and returns it cleanly to the client.

Now, let’s consider a scenario where you need to retrieve data using a third-party library provided by an external company. An inattentive developer might make direct method calls to this library within the business logic. Similar to the previous example, if the library undergoes changes, you’ll have to update all the classes in your system accordingly. However, this issue could have been prevented if a single class handled all the calls to the library.

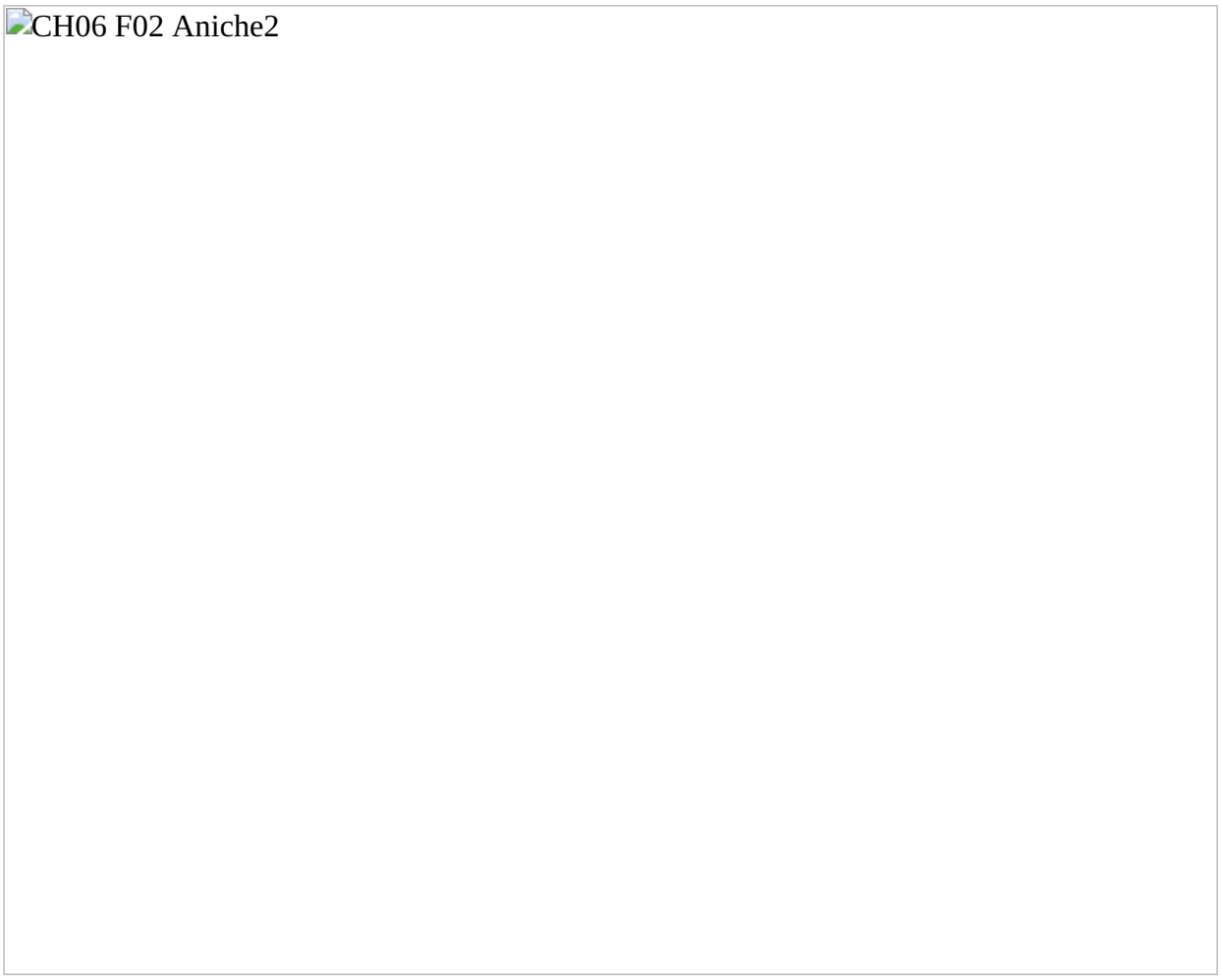
In a nutshell, you aim to minimize the impact of these external systems on your code. By doing so, if these systems ever need to change, the resulting impact will be as minimal as possible.

6.1.1 Do we need an interface or not?

More often than not, it’s a good idea to have an interface that explicitly separates the infrastructure from the domain code. While many argue that interfaces support us in changing the infrastructure’s implementation later on, the primary reason I utilize interfaces is that they prevent me from writing any code that relies on the infrastructure directly. Within an interface, I can only define a set of methods that I anticipate from the infrastructure and nothing more.

Let’s consider a scenario where you’re tasked with implementing a feature that requires obtaining a list of employees. You’re aware that this list is stored in a database, so your code needs to interact with this external system. One approach to implementing this (also depicted in figure 6.1) would be to define an `EmployeeRepository` interface with a method `List<Employee> allEmployees()`. Additionally, you would have a concrete implementation, let’s call it `HibernateEmployeeRepository`, which implements this interface and internally utilizes Hibernate to communicate with the database in the background. By using this interface, you ensure that no details related to the database leak into the domain code.

Figure 6.1 The EmployeeRepository interface prevent the implementation details from leaking to the domain.



Different architectures, such as the Ports and Adapters (also known as the *Hexagonal Architecture*) and the Clean Architecture, suggest the same idea: an interface that speaks the domain language and prevents implementation details from leaking.

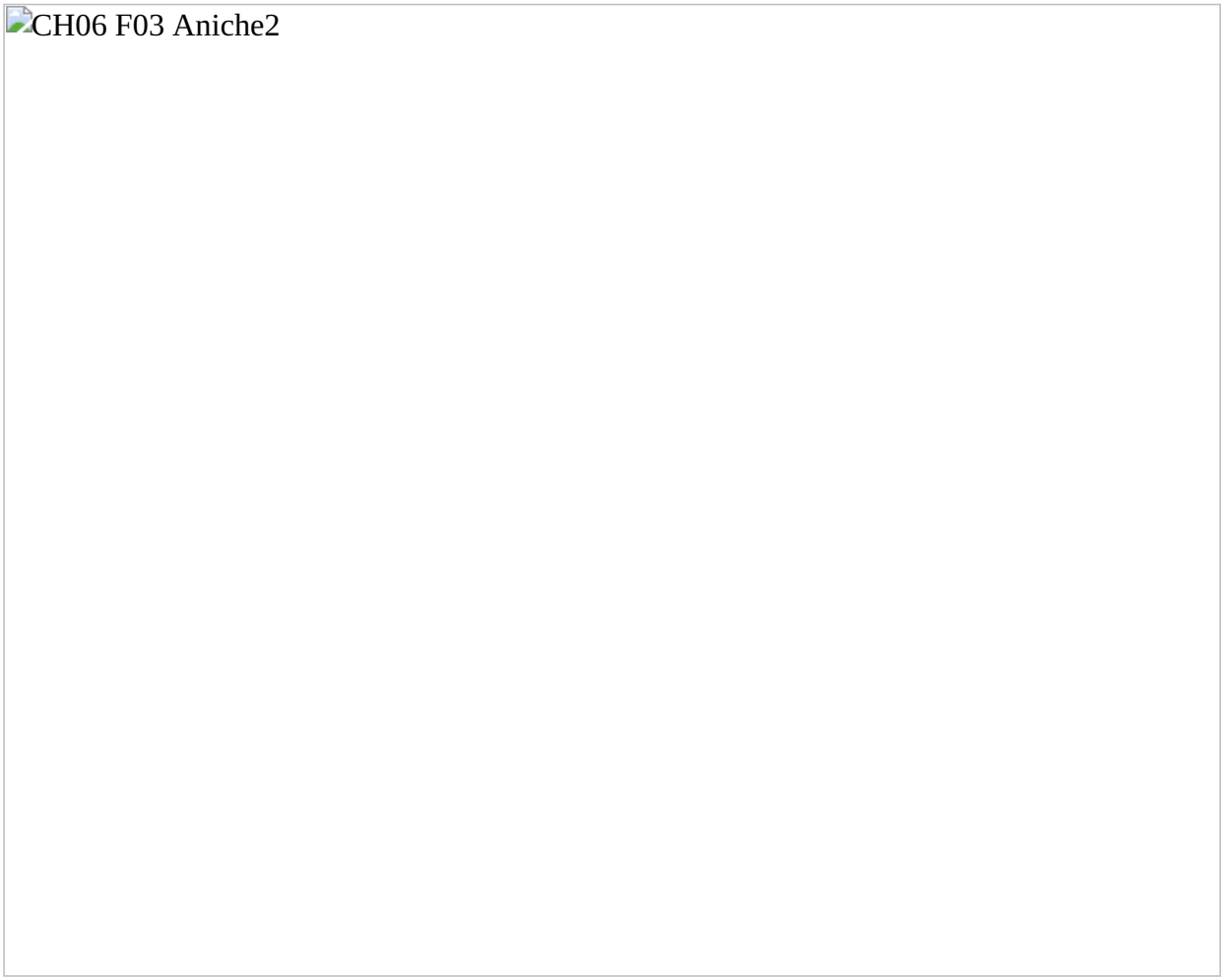
That being said, always having an interface is not a hard rule you can't break. Sometimes a class is good enough. As long as this class speaks domain language and doesn't allow implementation details to leak, you are in a good place.

My rules of thumb (also in figure 6.2) for when to create an interface are the following:

- If I anticipate multiple implementations of the same infrastructure, I immediately implement an interface. Doing so ensures a smoother transition when additional implementations arise and also enables me to create a fake implementation of that infrastructure for testing purposes.
- If I have limited knowledge about the specific details of the infrastructure, the interface provides a solid starting point. It allows me to proceed with coding the rest of the feature without being hindered by my lack of knowledge.
- When I know that I'll utilize this infrastructure in multiple places, relying on an interface is more lightweight than relying on a heavy class that brings along numerous

dependencies.
- If the underlying infrastructure is complex, it becomes more challenging to prevent details from leaking. By employing an interface, I am compelled to carefully consider the design from day one.

Figure 6.2 Deciding if there's a need for an interface.



With or without the interface, you must ensure that the code is well-encapsulated and the implementation details are hidden from the code, but not from the developers.

6.1.2 Hide from the code, not from the developers

Your design should conceal the internal details of the infrastructure from the rest of the code to minimize the impact of implementation changes. However, it's important not to hide too much from developers because understanding what's happening behind the scenes enables them to write optimal and efficient code.

A common misconception in infrastructure-related code design is the belief that everything must be hidden from everyone. The primary objective of properly encapsulating infrastructure

is to ensure that changes have a limited impact on the overall system. Your entire system shouldn't require extensive modifications when upgrading the version of your persistence framework, introducing read replicas to the database, or transitioning from SOAP to REST for a consumed web service.

However, simultaneously, you don't want to hide everything from developers. Such an approach can be impractical and unnecessary for most projects. For instance, if developers know that a relational database is used instead of a document-oriented database, they may optimize their designs accordingly. Creating a design that supports seamless switching between different database types is possible but expensive, requiring a compelling business reason to implement it.

Consider another example. If you are aware that retrieving the list of employees involves making a remote call to the Employee service, you may exert additional effort to ensure your code handles common failures in distributed architectures. While it's true that you can create abstractions and components that make a remote call appear like a local one to the rest of the system (though it may not always be achievable), doing so correctly demands significant effort, which may not be necessary for most systems. Moreover, in most systems, if developers are unaware that a remote call is involved in retrieving the employee list, they may inadvertently write code that doesn't function correctly from the start.

The extent to which you abstract your infrastructure is a topic of heated debates within the community. On one side, some advocate for treating infrastructure as a mere detail, not warranting excessive concern during the modeling of the rest of the system. On the other side, there are proponents who assert that infrastructure is not a mere detail but an integral part of an information system that should be accepted as such.

My viewpoint lies somewhere in between these perspectives. I agree that infrastructure should not be treated as a mere detail. For example, databases are core components in information systems. However, this doesn't imply that we should scatter their implementation details throughout the entire codebase. Instead, we should isolate infrastructure-specific code as much as possible to minimize the impact of changes, similar to what we discussed in the previous section.

6.1.3 Changing the infrastructure one day: myth or reality?

The strong encapsulation of the infrastructure code proves beneficial when changes become necessary. In such cases, the goal is to modify only the classes related to the infrastructure while ensuring the rest of the system continues to function seamlessly.

However, complete infrastructure overhauls are rare, so why optimize for such scenarios? This is a valid point. Most software systems seldom undergo complex migrations, such as transitioning from one type of database to another or switching web frameworks. Nevertheless, the infrastructure of most software systems undergoes constant evolution over time.

From a testing perspective, your infrastructure is constantly changing. For example, you may not want your unit tests to depend on a Kafka instance.

I can think of many evolutions in infrastructure you want to implement without breaking the rest of the system, including the following examples:

- The system requires caching, and you wish to cache specific queries without altering the entire codebase.
- Scaling demands necessitate the introduction of a read replica. You don't want to change every place in the code that calls the database and redirects it to the replica. Instead, you want your infrastructure abstraction to handle it.
- The authentication mechanism used internally by your company to validate calls from internal web services has changed. It shouldn't be necessary to modify every occurrence of a call due to this change.
- The system now needs to handle large file uploads and downloads, prompting a decision to transition to Amazon's S3 instead of storing files on on-premise disks. This migration becomes much easier if the infrastructure code is encapsulated behind a well-defined interface.

Therefore, even if you are sure your infrastructure won't drastically change, remember that it'll evolve, and you want to keep the evolution cost to a minimum.

6.1.4 Example: Database access and the message bot

From the start, we kept infrastructure code separate in all the code examples in PeopleGrow! We didn't want those implementation details to clutter the entire code base. This applies not just to infrastructure code but to everything we've talked about in this chapter. So, let's go over some of it again.

Look at the repository interfaces. From a domain point of view, these repositories represent all the required data-access behavior. For example, the `EmployeeRepository` interface in listing 6.1 has methods such as `findById`, which returns the employee with that ID, `findByLastName`, which returns all employees with a specific last name, and `save`, which persists the new employee to the database.

Note how this interface doesn't give any hints on how these methods are implemented. Which database is behind it? Which library is used to communicate with it? This means that if we ever need to change any implementation detail related to accessing employee information in the database, the rest of the system won't be affected by the changes.

Listing 6.1 The EmployeeRepository interface

```
interface EmployeeRepository { #1
        Employee findById(int id);
        List<Employee> findByLastName(String lastName);
        void save(Employee employee);
        // ...
}
```

PeopleGrow! is definitely in a good position in case a change needs to happen in how we handle database access.

Behind the scenes, PeopleGrow! uses Hibernate as a persistence framework and Postgres as its database of choice. The `HibernateEmployeeRepository` contains all the handling code.

Suppose we now want to improve the performance of the `findByLastName` method, and we decided to add some caching. All we need to do is to change the implementation of this method, hitting a cache first, and if not there, then hitting the database. We plug our caching library into this class, illustrated as the `Cache` class in listing 6.2.

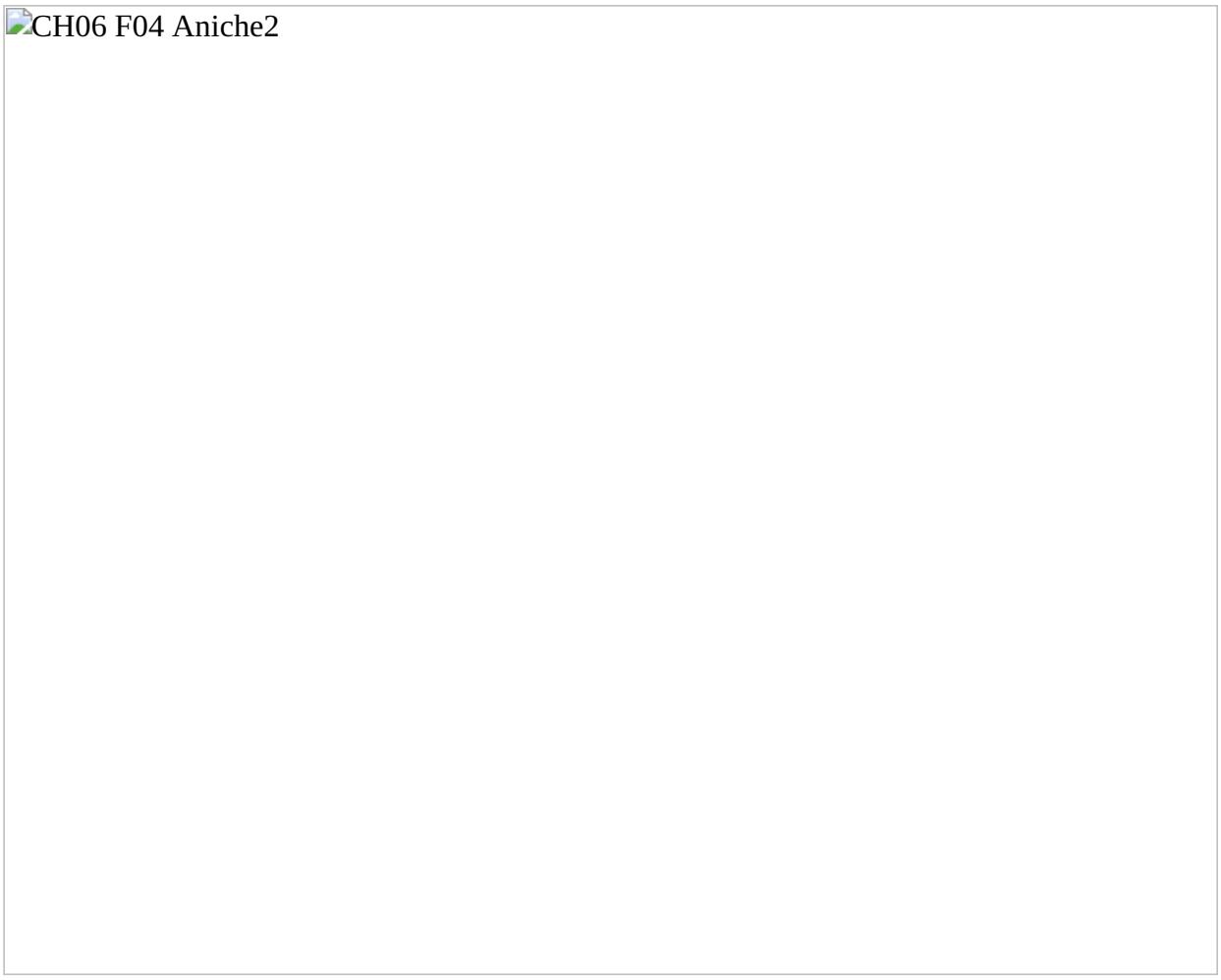
Listing 6.2 Changing some infrastructure details

```
class HibernateEmployeeRepository implements EmployeeRepository {

        private Cache cache; #1

        // ...

        public List<Employee> findByLastName(String lastName) { #2
          if (!cache.contains(lastName)) {
            cache.add(lastName, session
              .createQuery("from Employee e where e.lastName = ...")
              .setParameter(...)
              .list());
          }

          return cache.get(lastName);

        }
}
```

Thanks to this separation between infrastructure and domain code, we only needed to modify the `HibernateEmployeeRepository`, and the rest of the codebase remained untouched.

I won't get into the details of how to implement caching or what challenges it brings to the implementation because this isn't the focus of this book. Introducing caching in a real system may be more complex than in the example above. Nevertheless, changing code in a single place rather than everywhere is already a great advantage, illustrated in figure 6.3.

Figure 6.3 Changes only have to happen in a single place, not in the entire system.



A challenge that emerges once you are modeling an interface to hide away the details is that you may design one that doesn’t use the infrastructure to its fullest. And you don’t want that.

6.2 Use your infrastructure to the best

Get to know your underlying infrastructure and use it to the best advantage. Design your classes in a way that optimizes what they have to offer. This helps you to write the best system you can with the least effort.

Most software systems rely on existing components to fulfill their tasks. For instance, they depend on databases for data persistence and web frameworks for efficiently building APIs. These components have undergone significant advancements in the past decades, providing developers with a wealth of features that would be a shame to disregard simply because they don’t fit our class design.

Let’s consider a scenario where you need to perform a data operation, and your database offers a remarkable feature that can assist in accomplishing it correctly and efficiently. The alternative approach would be to load the data into the system and perform the operation

purely through code, avoiding the "pollution" of your design with a feature specific to that particular database. While this may make your design appear clean and technology-independent, it ultimately complicates your work and degrades the quality of your system. The second option is significantly slower and more susceptible to bugs compared to the first. Unless there's a valid reason to do so, you shouldn't overlook the advantages offered by the infrastructure.

The challenge lies in utilizing your infrastructure to its fullest potential without compromising the integrity of your design.

6.2.1 Do your best not to break your design

In the world of software design, nothing comes without trade-offs. If you wish to utilize a remarkable feature from your infrastructure, it may require creating additional abstractions. This ensures that your design remains protected in case there are any changes in how this operation is handled in the future, which is not uncommon.

Technically speaking, as we've just seen, an interface that provides a domain-focused operation implemented by a class encapsulating the handling code is often sufficient.

For instance, let's consider the scenario where you need to generate a report that aggregates information from multiple tables, and your database offers a perfect solution for this. Similar to the approach taken with the employee list, one design idea would be to create an interface called `ReportGenerator` with a domain-focused method like `Report generateReport()`. A concrete class would then implement this interface and utilize the magical query capabilities of your database. If the need arises to change this in the future, such as migrating to a database with fewer capabilities, you would simply create a new implementation of the interface to achieve the same outcome.

Notice how similar this is to the previous example of `EmployeeRepository`. By creating domain-focused interfaces that emphasize the expected business outcomes and abstract away implementation details, many challenges related to utilizing specific infrastructure features can be mitigated.

In information systems, a common challenge arises when you consider bypassing the aggregate and performing an operation directly on one of its aggregated objects. For instance, a direct database update command might be more performant than making the change through the aggregate root. However, I recommend revisiting your design before proceeding with such an approach.

Ask yourself if the object needs to be part of the aggregate in the first place if you feel the change should be made directly within the object. This is often the appropriate response to this design challenge. Alternatively, you can consider eliminating the invariant. Not all invariants we initially consider are truly necessary. If the object must be part of the aggregate due to essential invariants, can you accept some eventual inconsistency and re-model it using domain events? Breaking your design should be a last resort, as it is something you want to avoid at all costs.

6.2.2 Example: Cancelling an enrollment

As you might remember, PeopleGrow! lets employees cancel their enrollment in an offering of a training. In chapter 3, we implemented the `cancel()` method in the `Offering` aggregate root that would receive the employee that wants to skip the training and then would find them in the list of enrollments, remove them from the list, and add one extra available spot to the offering again. This code is reproduced in listing 6.3.

Listing 6.3 The cancel method in the Offering entity

```
class Offering {

  private List<Enrollment> enrollments;
  private int availableSpots;
  // ...

  public void cancel(Employee employee) { #1
    Enrollment enrollmentToCancel = findEnrollmentOf(employee);
    if(enrollmentToCancel == null)
      throw new EmployeeNotEnrolledException();

    Calendar now = Calendar.getInstance();
    enrollmentToCancel.cancel(now);

    availableSpots++;
  }

  private Enrollment findEnrollmentOf(Employee employee) { #2
    // loops through the list of enrollments and
    // finds the one for that employee
    // ...
  }
}
```

We discussed back then that this `cancel()` method isn't the most performant. After all, it needs to loop through the entire list of enrollments. As we know, there's a relational database behind the scenes, which means we may need to have an extra query to bring this list to memory. We implemented this because we considered this performance hit not to be a problem for such a small system. But PeopleGrow! is growing, and we can't afford that anymore.

The first idea that comes to the developers' minds is to move the entire canceling logic to a Service class. The Service would coordinate canceling the enrollment and then bump the number of available spots-—something like what you see in listing 6.4.

Listing 6.4 The cancel operation as a service

```
class CancelEnrollmentService {

  private OfferingRepository offerings;

  public void cancel(int offeringId, int employeeId) { #1

    if(offeringId==null || employeeId==null)
```

```
      throw new InvalidArgumentException();

    Offering offering = offerings.getById(offeringId);

    Enrollment enrollment =
      offerings.getEnrollment(offeringId, employeeId); #2

    if(enrollment == null)
      throw new EnrollmentDoesntExistException();

    enrollment.cancel(now());
    offering.increaseAvailableSpots(); #3
  }
}

class Offering {

  // ...

  public void increaseAvailableSpots() {
    availableSpots++;
  }
}
```

Although this implementation no longer requires loading the complete list of enrollments, it does have some drawbacks. The most significant drawback is that it reduces the control of invariants in the aggregate root. With this implementation, any client can now manipulate the available spots by simply requesting an increase. Additionally, this approach may introduce inconsistencies. For instance, what happens if, in another part of the business logic, we load the list of enrollments and then the cancellation service is invoked? The list of enrollments within the offering may become outdated because we don't reload it when removing an offering from the service. Lastly, the implementation of `getEnrollment` had to be included within the `OfferingRepository` because we typically don't have repositories for internal parts of the aggregate. We usually handle them through the aggregate root, which isn't ideal.

Transferring the control of invariants out of the aggregate root may seem like the easiest solution, but it can quickly result in inconsistent objects. As we discussed in Chapter 3, it's crucial to do our best to avoid this. However, I understand why it happens. Refactoring a settled design can be challenging. Nonetheless, investing in this refactor and eliminating any possibilities of object inconsistencies will ultimately pay off, even in the short term. The payoff will depend on how important or frequently used the aggregate is within the entire system.

Let's try and re-design this. The problem seems to originate from the need to keep the list of free spots up-to-date in the `Offering` entity. Do we really need this information in the entity? Because if we take `availableSpots` out of the entity, the problem suddenly becomes simpler. We could then promote enrollments to an aggregate of itself. Whenever a request for canceling an enrollment comes in, an application service grabs the offering and the enrollment and cancels it. Listing 6.5 is the implementation of the `CancelEnrollmentService`.

Listing 6.5 The CancelEnrollmentService application service

```
class CancelEnrollmentService {

  private OfferingRepository offerings;
  private EnrollmentRepository enrollments;

  public void cancel(int offeringId, int employeeId) { #1

    if(offeringId==null || employeeId==null)
      throw new InvalidArgumentException();

    Offering offering = offerings.getById(offeringId);
    if(offering == null)
        throw new OfferingDoesntExistException();

    Enrollment enrollment =
      offerings.getEnrollment(offeringId, employeeId);
    if(enrollment == null)
        throw new EnrollmentDoesntExistException();

    Calendar now = Calendar.getInstance();
    enrollment.cancel(now); #2
  }
}

class Offering { #3

  // private int availableSpots;

  // public void cancel(Enrollment enrollmentToCancel) { ... }
}
```

The number of available spots could easily be calculated by a SQL query in the Postgres database we have behind the scenes. We then create a method `availableSpots(Offering)` in the `OfferingRepository`. In the concrete `HibernateOfferingRepository` implementation of that repository, we'll use Hibernate's query language to get that information. Internally, the class even caches the results to speed up the response. See the implementation in listing 6.6.

Listing 6.6 Calculating the available spots via SQL query

```
class HibernateOfferingRepository implements OfferingRepository {

  private Session session;

  public int availableSpots(Offering offering) { #1

        if(!cache.contains(offering)) { #2
          int spots = (int) session
            .createQuery("select maximumNumberOfAttendants - " +
              "count(...) from Offering o where ...")
            .setParameter(...)
            .getSingleResult();

          cache.put(offering, spots);
        }

        return cache.get(offering);
```

```
  }
}
```

There are other ways you could think of modeling it. For example, you could keep the `availableSpots` attribute in the `Offering` entity and use domain events to update both aggregates. Once a cancel request comes in, the domain service publishes a domain event, and different listeners update the offering and the enrollment. That means possible eventually-consistent information, but your users may be fine with it.

As always, there are no right and wrongs, only trade-offs.

6.3 Only depend on things you own

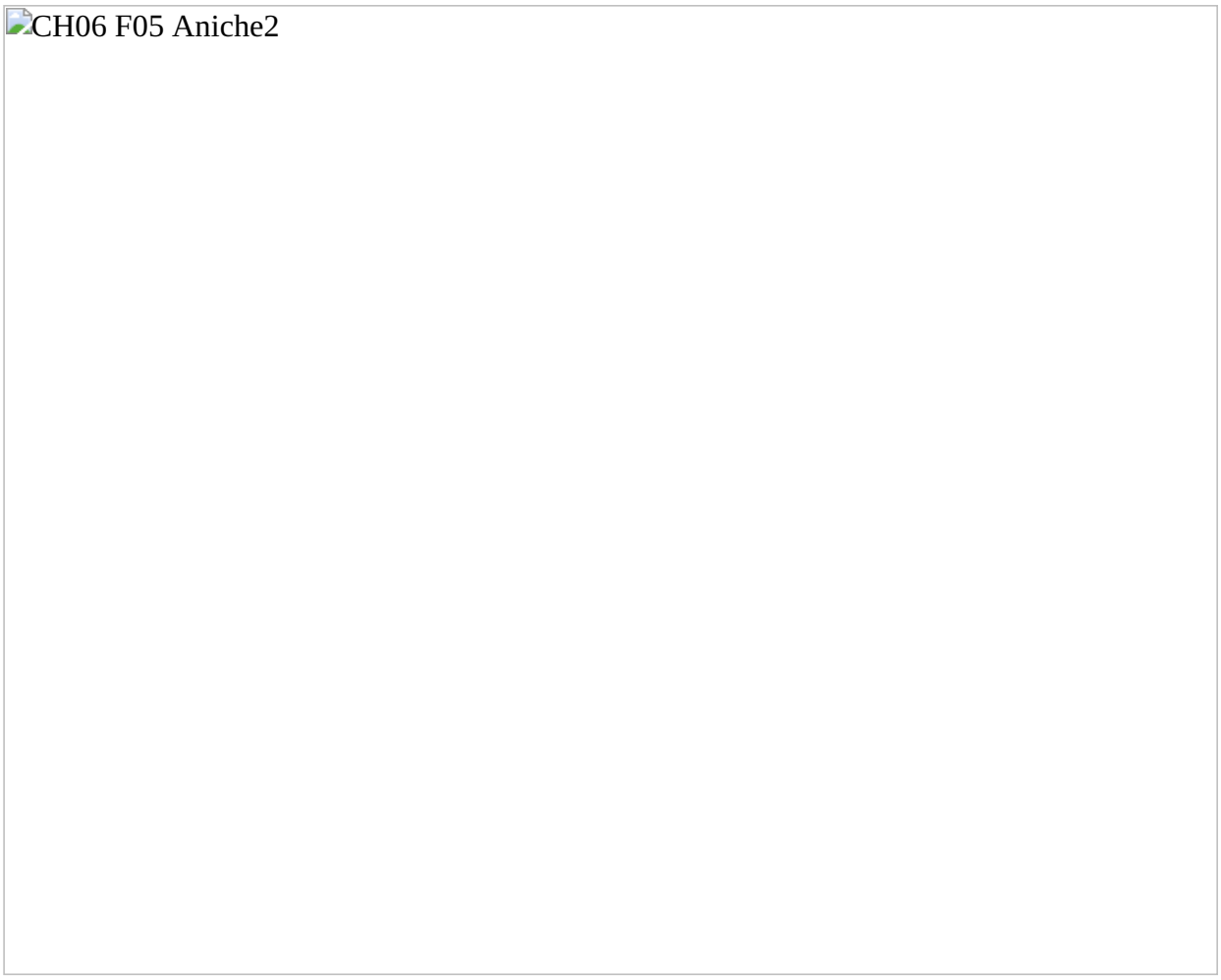
Create wrappers on top of third-party data structures and libraries. This prevents third-party dependencies from spreading too far off in the code base, saving you time whenever such out-of-control classes change.

Our systems greatly benefit from the availability of third-party libraries and systems. Many of these libraries even provide software development kits (SDKs) that simplify the integration process. However, it’s important to remember that these libraries also bring along code that is beyond your control. Consequently, you have no influence over the library’s release cycle or the possibility of introducing breaking changes.

Therefore, when incorporating code from other libraries and systems, it’s crucial to establish a layer that prevents their proliferation throughout your codebase. These wrappers will ensure that any changes in the library code will only impact this specific layer and nothing else.

Let’s consider an example where you are integrating with a payment gateway that offers a clean library to streamline the integration process. To initiate a payment, this library requires you to invoke the `makePayment()` method and provide a `PaymentDetails` data structure containing information such as the payment amount, currency, buyer’s email, and other relevant details. However, you don’t want the `PaymentDetails` class, which you don’t own, to be scattered throughout your entire codebase. In this case, you would need to create your own class that encompasses the same information and pass it to your `PaymentGateway` class (referred to as an "adapter") that would then convert the information into the appropriate API calls expected by the library. I have depicted this concept in Figure 6.4.

Figure 6.4 The adapter prevents the library to get spread into the domain.



At first, these wrappers may appear redundant. After all, your data structure resembles the one used by the library. However, it’s important to recognize that you have complete control over your own data structure and can manage its changes. If the library undergoes modifications, such as requiring an additional function call, you will only need to update the adapter.

It’s possible that certain changes in the library may necessitate modifications in other areas of your application. For instance, let’s say the payment gateway now requires an extra piece of information. In such a case, you would need to incorporate this information into your data structure and find the appropriate location in your code to load it. Nevertheless, these changes are significantly more within your control.

Additionally, it’s important to remember that libraries often become deprecated or undergo complete rewrites. If such a situation occurs, it is much easier to make the necessary adjustments in the adapter rather than having to locate and modify all the different parts of the application that utilize the library.

6.3.1 Don’t fight your frameworks

While it’s important to minimize coupling to external dependencies you don’t control, it’s equally essential to avoid fighting against them. Complete decoupling from all dependencies is an unattainable goal.

Attempting to achieve full decoupling can significantly increase the complexity of your code by orders of magnitude. Moreover, there’s a likelihood that your abstractions may fall short of meeting your requirements at some point.

As a developer, it is necessary to find a balance between accepting dependencies you acknowledge will exist and challenging dependencies you are unwilling to accept. Let me illustrate how I approach this with a few examples:

- If I have chosen Spring MVC as my framework for model-view-controller (MVC), I won’t attempt excessive decoupling from it. I will leverage all the capabilities that Spring provides to their fullest extent. While I will avoid using Spring code in my domain objects (such as entities and services), I won’t shy away from utilizing Spring’s helpful utilities in my controllers.
- If I have selected Hibernate as my persistence framework and Postgres as my database, I will encapsulate Hibernate code within repository or data access object classes to shield it from the rest of the system. However, I won’t ignore the fact that there’s an object-relational mapping mechanism at work or the fact that I have a robust relational database supporting my system.
- If I need to integrate with a third-party payment gateway using their provided library, which may be less stable compared to large-scale open-source frameworks and subject to frequent changes (due to the ever-evolving nature of payment systems), I will add a wrapper layer on top of it and ensure that no other parts of the code directly depend on it.
- If the system requires generating reports in Excel files, I will design domain-specific data structures to represent the report information and encapsulate the Excel generation code within an adapter. I won’t expose the specific library used for generating the file to the rest of the codebase.
- If I have opted for Amazon AWS as my cloud provider, I won’t disregard the powerful capabilities it offers. I will make full use of Amazon SQS (AWS’s queue) and make architectural decisions that favor SQS. However, I will encapsulate the relevant code within an adapter to prevent AWS-specific code from spreading throughout the entire codebase.

Please note that these are personal approaches and may not be absolute truths. You may have different arguments and make different decisions in the given scenarios.

Nonetheless, the general principle still applies: minimize dependencies on external components you don’t own, and work harmoniously with your chosen frameworks and architectural decisions.

6.3.2 Be aware of indirect leakage

Interfaces can be effective in preventing the rest of the code from knowing or depending on implementation details, but there are still ways in which these details can leak into the code.

Let’s consider the example of Object-Relational Mapping (ORM) frameworks. ORMs often perform tasks behind the scenes that developers may not be aware of, and this can impact the rest of their code. For instance, when you return a Hibernate-managed entity to the rest of your code, even if your code isn’t aware that the entity is managed by Hibernate, it remains connected to Hibernate’s session and may be automatically persisted when the transaction scope is closed. Another common scenario is when client code invokes a getter method in a Hibernate-managed entity, triggering the framework to execute additional queries to the database without the developer’s explicit knowledge.

From the code itself, you may not see these behaviors, but the underlying infrastructure’s behavior indirectly seeps into your code. It’s up to you to determine whether this is favorable or undesirable and to what extent you should safeguard your design against it.

In the specific example mentioned, if you find this kind of leakage undesirable, you can introduce an additional layer to ensure that entities used within the domain code are never managed by Hibernate. This requirement would be part of the contract defined by the interface, which the concrete class implementing it and utilizing Hibernate internally must adhere to. Although it requires extra effort, it provides an added level of flexibility. If you ever decide to transition from Hibernate to another framework, your domain code will remain unaffected.

As we mentioned before, these implementation details should be hidden from the code but never hidden from the developer. Once again, developers must have a thorough understanding of their infrastructure choices and the implications they have on the overall design.

6.3.3 Example: Message Bot

If you recall PeopleGrow!'s message bot implementation, you may have noticed that we already ensured that the data structures of the Bot SDK didn’t spread across the code base. Let me repeat the code of our `SDKBot` in listing 6.7.

The `ChatBotV1` class provided by the SDK requires a `BotMessage` class to write a message via the bot. The `ChatBotV1` and `BotMessage` are third-party classes we have little control over, so we did what’s best. The `Bot` interface doesn’t let them leak into the domain. They are well encapsulated into the `SDKBot` concrete implementation.

Listing 6.7 The implementation of the SDKBot, again

```
interface Bot {
  void sendPrivateMessage(String userId, String msg);
}

class SDKBot implements Bot {

  public void sendPrivateMessage(String userId, String msg) {

    var chatBot = new ChatBotV1(); #1
    var message = new BotMessage(userId, msg); #2
    chatBot.writeMessage(message); #3
```

```
  }
}
```

Although this may look like code duplication, because the `sendPrivateMethod` requires the precise information that the `BotMessage` data structure needs, you never know what tomorrow will look like. If a change in these third-party classes happens, you know the only place you'll need to change, and you will be better able to trace any other change that needs to happen in your domain code in case the `Bot` interface is also forced to change.

6.4 Encapsulate low-level infrastructure errors into high-level domain errors

Any errors triggered by the infrastructure must be fully encapsulated into the infrastructure layer, converted to an error that makes sense to the domain, and gracefully handled by the application.

Infrastructure and the libraries that help us communicate with them (for example, think of a Postgres database, a JDBC driver that's used to communicate with it in Java, and Hibernate, a popular object-relational mapping framework that many teams use) may throw all sorts of different errors and exceptions.

This means that such errors must be known by the low-level infrastructure implementation. For example, if your database throws a "unique constraint exception" whenever the constraint is violated, the infrastructure layer must be aware of and catch it.

Some errors might be recoverable, and you can take action directly in the infrastructure layer without even popping it up to higher layers. For example, if your database layer notices that the European cluster isn't available, the layer may switch to the American cluster and execute the query there. Of course, you have to decide whether switching between clusters is desirable, but if so, encapsulating it in the infrastructure layer will save the rest of your code from knowing how this happens and allow this behavior to change in the future more easily.

Other types of errors are unrecoverable, and the best you can do is to show an error message to the user and internally log the low-level details to facilitate debugging. In these cases, the infrastructure may throw a domain-focused exception containing helpful information to be displayed to the user while also logging relevant details to the developer.

Never let exception classes from your framework spread over your codebase. You don't want a `JdbcPostgresUniqueConstraintException` or similar handled in upper layers. This prevents your code from being coupled to infrastructure decisions and the current framework of choice. If you need to bubble up the constraint exception, the best alternative is to create a domain-focused exception—for example, the `EmployeeAlreadyExistsException`, which is free of any infrastructure details.

6.4.1 Example: Handling exceptions in SDKBot

The `writeMessage` method of the `ChatBotV1` API may throw an `IOException`. Although this exception is a native Java one, we don't want to let it spread. Let's handle it properly in the infrastructure code.

See listing 6.8. In case the exception happens, the code throws a `BotException` containing the user id and the message that wasn't delivered, and log the low-level details of the original exception to the developer.

Listing 6.8 The implementation of the SDKBot, again

```
interface Bot {
  void sendPrivateMessage(String userId, String msg);
}

class SDKBot implements Bot {

  public void sendPrivateMessage(String userId, String msg) {
    try {
      var chatBot = new ChatBotV1();
      var message = new BotMessage(userId, msg);
      chatBot.writeMessage(message);
    } catch (IOException e) { #1
      throw new BotException(userId, message);
      LOGGER.error(e);
    }
  }
}

class BotException extends RuntimeException { #2
        ...
}
```

The implementation now prevents infrastructure errors to bubble up to other layers.

6.5 Exercises

Discuss the following questions with a colleague:

6.1. Does your current project separate infrastructure from domain code? If not, what should you do to get there?

6.2. How far do you think one should go when isolating infrastructure code? Can we take compromises? What are the trade-offs involved?

6.3. What do you think of the idea of always encapsulating things you don't own? Is this a good idea or not? Why?

6.6 Summary

- You should decouple any infrastructure handling code from the domain code. Doing this reduces the impact that changes in the infrastructure code may have on the entire code base.
- The infrastructure layer should hide the implementation details from other parts of the code base. Developers should know what's behind the scenes, as understanding helps them develop better software systems.
- You shouldn't let the third-party libraries and data structures of the external systems spread throughout your code base. Create (domain-focused) wrappers around them.
- Don't fight your frameworks of choice. You won't ever be able to fully decouple your system. Instead, decide which tools and technologies you accept being coupled to and which you don't.

7 Achieving modularization

This chapter covers

- Designing modules that provide complex features through simple interfaces
- Reducing the dependency among modules
- Defining ownership and engagement rules

Up to this point in our journey through simple object-oriented design, our discussions have primarily focused on simplicity, consistency, abstractions, and extension points. We discussed how to apply these ideas from small methods to a set of classes. However, as we venture into the realm of large, multi-faceted systems, our scope must necessarily broaden. We must consider not just classes within a single component but also how different components that perform entirely different business operations interact and integrate.

Think of a large-scale business system that takes care of an e-commerce shop. The billing system, which takes care of charging the customers, is complex and most likely developed by one dedicated team. The same happens for the delivery system, which ensures that the goods get to their buyers, and for the inventory system, which controls whether items are still available or if the shop has to refill them. Although these systems are different from each other, they must collaborate. The delivery system must consult with the inventory system before delivering the goods. Billing must notify the Delivery that the invoice was paid and the goods can be delivered.

Picture each component as an individual player on a football team; each has a role to play, but their effectiveness is defined by how they coordinate and work together toward a common goal. Just as football players need to understand their positions, responsibilities, and tactics, software components must also be well-defined. They need clear contracts about how they should interact with others and what others can expect from them. This delineation helps components work harmoniously, ensuring they collaborate without clashing. Importantly, it also allows for components to evolve and improve independently from one another. With a clear contract, a single component can be altered or upgraded without triggering a domino effect of changes across the entire system.

Neglecting the design of your modules can lead to considerable difficulties down the line. Components not designed with adequate care tend to become overly dependent on each other, leading to tight coupling. Changing even a single module can become a Herculean task in a tightly coupled system, because it may require changes in many other interconnected modules. This setup is also a breeding ground for bugs, because developers are forced to understand the intricate workings of multiple modules before they can make even minor maintenance tweaks.

What’s important to remember is that the principles we’ve discussed so far — simplicity, keeping things consistent, good abstractions and extension points, and isolating infrastructure

details — are equally applicable at the module level. As we encapsulate data within a class, we can encapsulate related functionality within a module.

The design of individual components is undoubtedly important, but in a large system, it's their collaboration that brings about true harmony, just like in an orchestra. In this chapter, we'll repurpose the ideas we discussed so far so that they make even more sense at the module-level.

Note

For this chapter, the illustrative example will come in the end, but it will use all the principles introduced here.

7.1 Build deep modules

Modules should provide simple interfaces on top of complex features. The simple interface makes integrating other modules easier, reduces coupling, and simplifies its evolution. Modules should also be cohesive and own everything that's related to the functionality they expose.

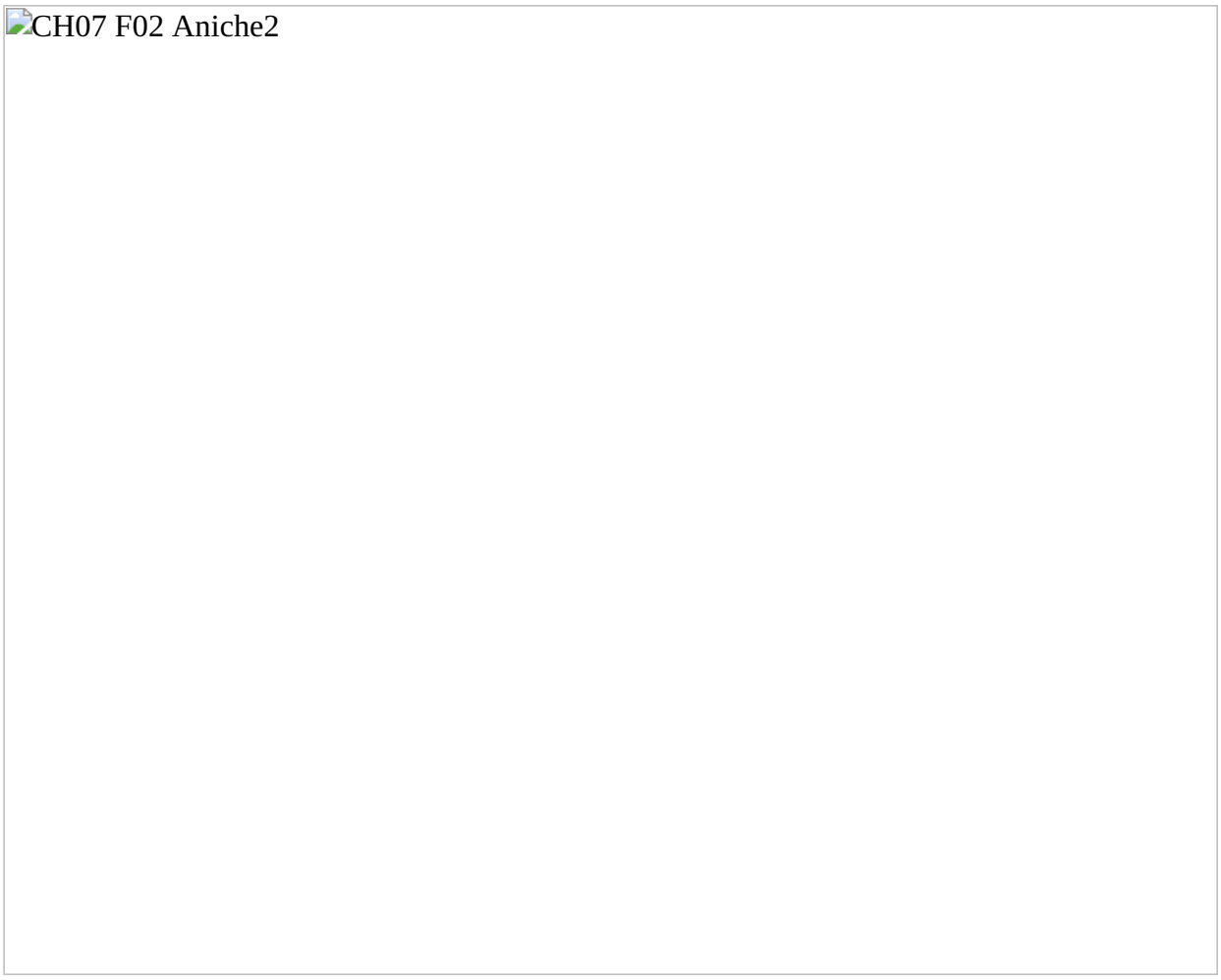
A great module hides all the details of complex features and offers a simple interface that removes all the complexity from clients.

Note that, at this level, I'm not talking about encapsulating some complex business rule in a class. It's much bigger than that. We're talking about hiding an entire business behind it. Repeating the example I gave in the introduction of this chapter, in an e-commerce system, all the rules related to delivery should be in a "Delivery" module, while all the rules related to invoicing and payment should be in a "Billing" module.

Delivery and Billing are very complex modules. If they ever have to collaborate, Delivery shouldn't need to deeply understand about Billing, and vice-versa. The more straightforward an interface a module can provide to the other, the better it is for the clients and for itself. It's easier for the clients to integrate with the new module. For the module itself, the simpler and leaner the interface is, the easier it is for the developers to make changes to it without breaking all its consumers.

Figure 7.1 illustrates what I mean by *deep modules*. Note how big and deep the module is, showing that it offers many complex features but how "thin and simple" its API layer is. Other modules only need to understand the API layer and nothing more.

Figure 7.1 Modules should offer a simple API on top of complex features.



Deep modules

I borrowed the term "deep module" from John Osterhout's *A Philosophy of Software Design*. John says that good modules are always "deep." A deep module provides powerful functionality under a simple interface. A shallow module, on the other hand, is a module that provides an interface that's almost as complex as the functionality it encapsulates. You don't want shallow modules because they increase the system's overall complexity without much benefit.

In the following sub-sections, I'll discuss the main challenges of building deep modules — namely, identifying what should be in the module, keeping related things closer, designing clear and simple interfaces, keeping compatibility, and providing flexible ways to extend the module.

7.1.1 Find ways to reduce the impact of changes

Deciding what should be included in a module is as challenging as determining what should be in a class or method.

Ideally, you want to minimize the number of modules that must be modified when a business domain changes. For example, if a new business rule emerges for "Invoice," we would only need to change the Invoice module; for example, there would be no need to modify the "Delivery" module.

This often entails having modules that encompass entire business domains. All classes related to "Invoice" should be within the same module, while all classes related to "Delivery" should be in a dedicated "Delivery" module.

7.1.2 Keep refining your domain boundaries

It’s surely hard to define a clear boundary between different business domains. Billing, Delivery, and Invoice may look completely different from each other in the simple example of this book, but in real life, domains are often strongly intertwined.

Working in strong collaboration with domain experts (they understand about the business) and technical leads (they understand about the pains that teams are having when trying to deliver working software) and keep revisiting your interpretation of the domain is what I’ve been doing.

A great reference on strategies to identify domain boundaries is the canonical *Domain-Driven Design* book by Eric Evans. In particular, sections 1 (putting the domain model to work) and 4 (strategic design).

7.1.3 Keep related things closer

Another way to approach this principle is by ensuring that related things or components that require simultaneous changes are kept close together. When you know that altering A necessitates modifying B, the change becomes easier and more predictable if A and B are nearby, such as being in the same module.

There are a few advantages in keeping related things together. First, when A and B are in the same module, they are likely part of the same continuous integration, build process, and test pipelines. If a breaking change occurs or one of the classes is modified without considering the other, a robust test suite will detect the problem. Conversely, if A and B are in different modules, these modules could be developed independently. Although there are methods to ensure that compilation or tests will fail if one class is changed but not the other, this approach is less straightforward.

Second, when A and B are in the same module, the module’s developers likely have a comprehensive understanding of both classes. They will know what needs to change and how to do it efficiently. Conversely, suppose A and B are separated and handled by different teams. In that case, no developer may have a complete picture in their mind, which would require them to request changes from the other team or make them themselves with less confidence.

7.1.4 Fight the accidental coupling or document it when you can’t

Although keeping related things in the same module seems straightforward in theory, it is more nuanced in practice. Business domains are highly interconnected, and developers often need to couple different modules.

For example, suppose we need to ensure that changes made to the details of an Invoice (such as adding a new attribute to represent the buyer's preferred address format) also reflect in the PDF generated by the Delivery module.

You might now be contemplating various design alternatives to avoid this coupling, and that's great! The longer you can avoid coupling, the better. However, the reality is that you won't always be able to come up with optimal alternatives, especially considering budget constraints when developing a feature.

We will discuss technical approaches later on to ensure that changes in the Invoice don't break the Delivery module, even if one is implemented before the other. For instance, by designing backward- and forward-compatibility in how modules communicate with each other.

In addition, you must document such couplings in the best way you can. Add code comments in the codebase, create documentation in the team's wiki, automate processes in your pipeline that alert you whenever you make a breaking change—-use whatever methods work best for you and your team.

7.2 Design clear communication interfaces

Modules should offer communication interfaces that are easy to use, require as little information as possible from clients, and are stable. That simplifies the integration between two modules and reduces the chances of breaking changes.

In large-scale software systems, modules need to communicate with each other to deliver the entire business workflow. Ensuring that a module effectively encapsulates entire business domains and that any changes to that domain only require modifications within the module is just part of the challenge. Another significant aspect is offering clear communication interfaces that enable other modules to interact with it.

Think of this communication interface as an API (Application Programming Interface) that a module provides to the external world. Implementation-wise, it can vary from simple classes and methods a module offers for other modules to use, to web APIs that support different request and response formats.

Does it need to be an HTTP call?

Although the term API is commonly associated with remote HTTP calls, an API does not always have to be offered exclusively through a web API. In modular monolithic systems, good old-fashioned method calls are used to exchange messages between modules.

A good communication interface possesses a few key characteristics. First and foremost, it is simple. Clients don't need to have an in-depth understanding of how the module works, as the

interface is designed to be easily understandable by anyone. It also avoids the need for complex input objects; if necessary, the API provides clean mechanisms to handle them.

Additionally, a good communication interface maintains its initial promises and ensures good backward compatibility. It recognizes that clients will not change their code every time a module changes. Furthermore, it avoids breaking clients without ample prior warning.

Moreover, exemplary communication interfaces offer clear extension mechanisms. They allow clients to implement specific variations in the module's behavior without inconveniencing the other module's development team or bloating the module with code that only serves a single client.

We will discuss these characteristics in more detail in the following sections.

7.2.1 Keep the module's interface simple

We should keep the module's interface to the external world as simple as possible. The challenge in designing such interfaces arises from the fact that modules encapsulate complex business logic by their very nature. Ensuring that the complexity does not leak outside the module is crucial.

Let's delve into what constitutes a simple interface for a module. The interface should not require knowledge of how the module operates internally. For example, suppose the Delivery module offers an API that provides information on whether the customer's goods were delivered. In that case, the clients of this API should not be concerned with the underlying mechanisms for retrieving this information. Whether it comes from a database, another module's call, or cache utilization for faster responses should be inconsequential to the clients.

A good interface is also one that is easy to use. Clients should not be burdened with performing complex setups or invoking a convoluted chain of methods in a specific order for the module to fulfill its purpose. All the complexity should be handled within the module, minimizing the client's responsibilities. Suppose you need to expose a feature of the module that requires an intricate setup. In that case, it is important to ensure that most of the complexity resides within the module rather than burdening the client.

When designing interfaces, we must remember that as soon as they are made public, other modules will begin to depend on them. Therefore, a good interface is stable—it undergoes minimal changes and does not force clients to update how they interact with the module continually. Furthermore, a good interface offers backward compatibility, but we will discuss this later.

7.2.2 Backward-compatible modules

Modularization offers numerous advantages. For example, it allows different teams to work on different parts of a product without excessively interfering with one another. However, it also presents the challenge of having different parts of the system residing in various locations, potentially even in different code bases and repositories.

In a system with a single module, it is easier to identify all the places that need to be modified when a method changes. The compiler may signal an error because the method now expects three parameters instead of two. However, determining when an API has changed in systems with multiple modules becomes more complex. Questions like, "Who is using this API?" are harder to answer. Furthermore, in a single module, if you modify a method's contract, you naturally feel compelled to update all the places where it is called; otherwise, your code won't compile. Backward-compatibility doesn't require constant consideration because you can modify the contract and update all its clients together. You have full control over the entire process.

In larger modularized systems, however, you may not want or even have the authority to modify other modules. You would need to request the team responsible for maintaining those modules, and this process can take weeks. Waiting for all the teams to update to the new API before you can deploy your new feature is not ideal.

Backward-compatibility plays a vital role in large modularized systems. The API of a module should be able to understand requests compatible with its previous versions. This is crucial for scalable development. You don't want to wait for other teams to complete their tasks, and other teams don't want their systems to break due to a change in your code.

Maintaining backward-compatibility doesn't come without its challenges. It undoubtedly increases the complexity of the module, because the API needs to understand not just a single type of request but multiple types. Your code may need to handle missing information (which may have been introduced in version 2 of the API) or different information (such as changes to the list of enumerated strings in version 2), and so on.

If you cannot maintain backward-compatibility with an old version, ensure you notify your clients well in advance, allowing them enough time to make the necessary changes. In public APIs, I've encountered cases where we notified our customers 1.5 years before deprecating the API!

Forward-compatibility is also important to consider, depending on how you deploy your software. Some companies deploy software in release trains, where all modules are deployed once a week. Due to the challenges of defining the correct order for module deployment each week, modules are deployed in a predetermined sequence. A client already using the new version 2 of the API may get deployed before the module that supports this new version 2. For a few seconds (or minutes or hours, depending on the duration and complexity of the deployment), clients may be making API calls using version 2 while the module still only understands version 1.

In such situations, you should ensure not only backward-compatibility but also forward-compatibility. This means that your modules should be able to handle requests gracefully, even if they are in formats that the modules do not yet understand.

7.2.3 Provide clean extension points

Modules should offer clean ways for other modules to extend their functionality, especially when the module's behavior needs constant changes, evolution, or customization for other modules.

By providing extension points, teams can avoid the need for frequent synchronization whenever they require minor changes in the module's behavior. For example, consider the delivery team modifying the delivery module whenever the store sells a new item. Such an approach wouldn't scale well.

I once worked with a team that developed billing systems for a company offering various software-as-a-service products. Each time a new product was introduced, we had to adjust to support their payment collection method. This caused significant delays in delivering these new products. To address this, we created a new API that offered product teams a lot of flexibility. Whether they wanted to charge customers with one-off payments or a combination of one-off and recurring monthly payments, they could do it easily using this new API. As a result, teams could release new products without needing to involve us, and we were pleased to no longer receive emergency requests with unreasonable deadlines every month.

We extensively discussed designing flexible classes in chapter 5, and the same principles apply at the module level. In essence, flexibility in modules can range from offering simple interfaces that other modules implement to fully flexible APIs, allowing clients to make complex requests as needed, as I described earlier.

7.2.4 Code as if your module is going to be used by someone else with different needs

It is essential to ensure that modules are as decoupled from each other as possible. One helpful approach to consider is coding your module as if it were to be used "by another company." The only way a module can be designed to work for another company is by avoiding coupling it to the specific decisions of your current company.

You might think this is a waste of time. It is crucial to carefully reason what needs to be decoupled and not abstract everything arbitrarily. As discussed in previous chapters, you need to find the right places to abstract and create extension points. However, five or ten years from now, your module will likely be used by another company. Why? Because your company today will not be the same in five or ten years. Companies change, evolve, and grow; the code must adapt accordingly. The most cost-effective way to allow this growth is by designing modules supporting such changes.

Many companies make the mistake of creating their in-house frameworks, driven by the desire to precisely match their current work methods and needs. However, they forget that work methods will change in a few years, and without investing in flexibility beforehand, refactoring the entire codebase to accommodate those changes will be much more expensive.

Let me provide you with a few examples. Suppose a module requires all other modules to register before using a feature. A future-proof approach would be for this module to offer a register API, which other modules can use. A less future-proof method would be to have a

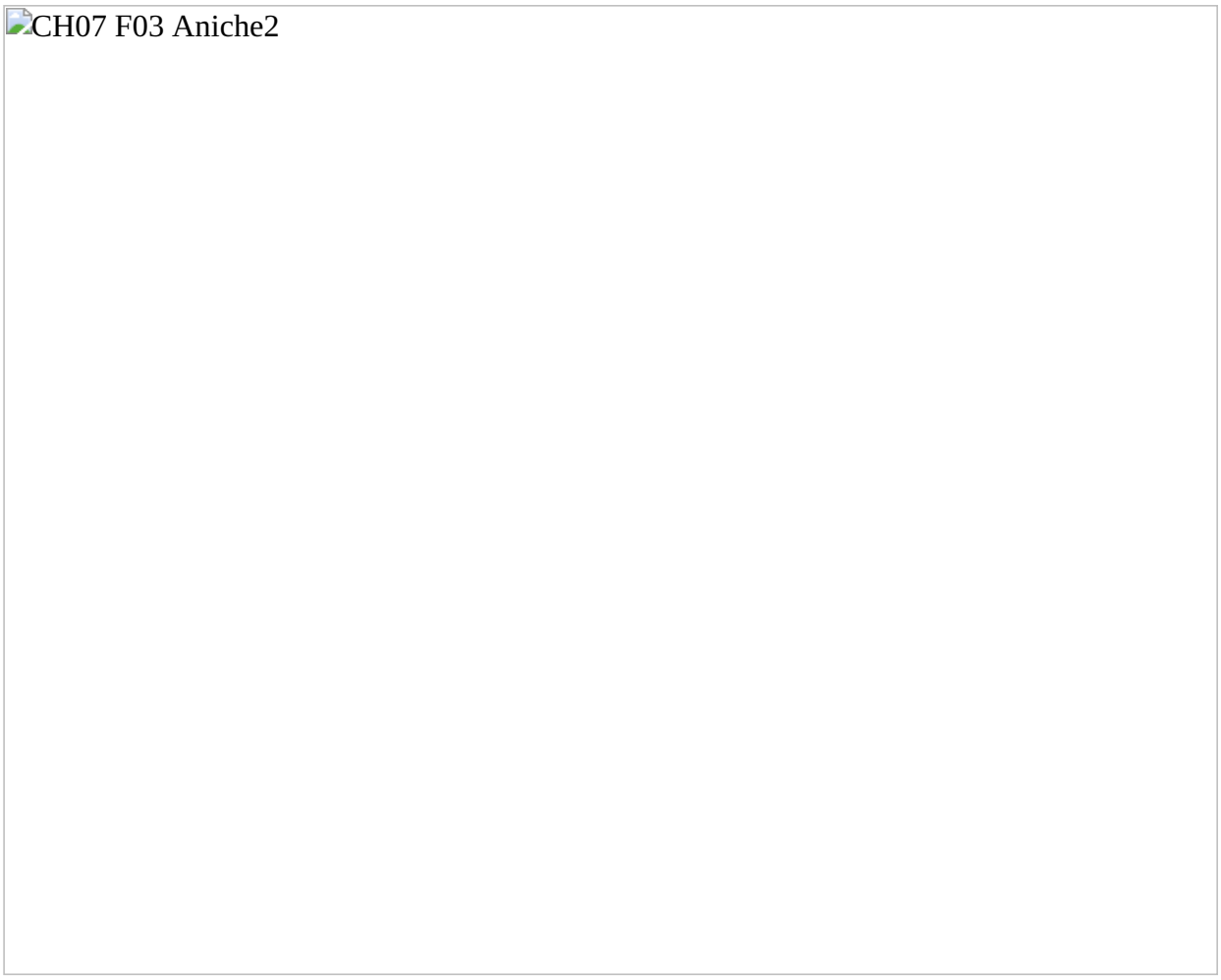
hard-coded array listing other modules, requiring changes whenever a new module emerges. Alternatively, if the company has its specific request-and-response format that precisely fits current needs, a future-proof approach would be to wrap the request-and-response message-creation process in a flexible abstraction, easily adaptable in the future. A less future-proof method would be to hard-code the request-message creation in all modules.

7.2.5 Modules should have clear ownership and engagement rules

Focusing on designing good modules from a technical perspective isn't enough in large, modularized systems. We also need rules to help teams know who to talk to and how to communicate in case of problems. That's why modules should have clear ownership and engagement rules.

By ownership, I mean it has to be evident to every team in the organization who's responsible for it (see figure 7.2). For example, if there's a bug, who fixes it? If some maintenance or evolution needs to be done, who does it? Deciding who owns what may be a trivial problem in small-scale software development, but it becomes a significant challenge as the company grows. The problems arising from the lack of ownership are numerous. The lack of clear ownership leads to the disappearance of knowledge about the module over time, no engineers feel responsible for improving its design, developers never feel confident about changing it, and as a result, bugs start to increase, among other issues.

Figure 7.2 Modules should have clear ownership.



Defining engagement rules is also essential when different teams develop dozens of modules. If I identify a bug in a module that doesn't belong to me, can I fix it or should I open a request? Does it require code review from at least one code owner, or is it enough that the continuous-integration pipeline passes?

I won't discuss which approach is better because that's getting closer to the social aspects rather than technical aspects of object-oriented design, but these things must be clearly defined. A great book on the social-technical challenges of building performant autonomous teams is *Team Topologies* by Matthew Skelton and Manuel Pais, published in 2019. I strongly recommend everyone read it.

Team topologies in a nutshell

In a nutshell, *Team Topologies* proposes four fundamental team topologies, and three ways of interaction. The four topologies are:

- Stream-aligned teams: focused on a segment of the business domain
- Enabling teams: helps stream-aligned teams to overcome obstacles

- Complicated subsystem team: dedicated to overly complex or systems that require specific expertise
- Platform teams: accelerates the delivery of stream-aligned teams by providing internal products

The three ways that these different teams can collaborate are:

- Collaboration: teams work together for a specific amount of time to explore new things or solve a new problem
- X-as-a-Service: one team provides and one team consumes something "as a service"
- Facilitation: one team helps and mentors another team

As the book discusses, explicitly designing the teams' goals, ownership boundaries, and interaction modes is crucial to accelerate and increase their productivity.

7.3 No intimacy between modules

Modules should aim to minimize their level of interdependence with other modules. The less they know about each other, the better. Modules shouldn’t allow others to inspect their internal workings too extensively. This practice ensures that modules can evolve gracefully without causing disruptions in other parts of the system.

In object-oriented design, having too much knowledge about the internal details of other code components is frowned upon, and the same applies at the module level. Modules that are overly aware of how other modules function are more susceptible to breaking if those other modules undergo changes in their internal implementation.

For example, consider a module that uses caching to improve response times. No other module should have to care or be aware of this caching process. By enforcing this rule, the original module remains free to modify its caching mechanism without fear of breaking other modules that rely on it.

All the principles of encapsulation and information hiding we discussed earlier are equally relevant here. Now, let’s focus on ideas that emerge when thinking at the module level.

7.3.1 Make modules and clients responsible for the lack of intimacy

Ensuring that modules remain unaware of the internal details of other modules is a responsibility shared between the module itself and its clients.

Now, let’s talk about the clients. In an ideal world, clients wouldn’t have to worry about this, because modules would be perfectly designed, and no details would ever leak. However, in reality, our designs are often imperfect, and due to a combination of limited knowledge (about what the application needs to do, not lack of design knowledge) and time pressure, we sometimes make suboptimal decisions.

These less-than-ideal decisions may become apparent to clients. Just because the module leaks some information doesn’t mean clients must use it. Instead, the client should ignore it and hope that the module will be fixed one day without causing any negative impact.

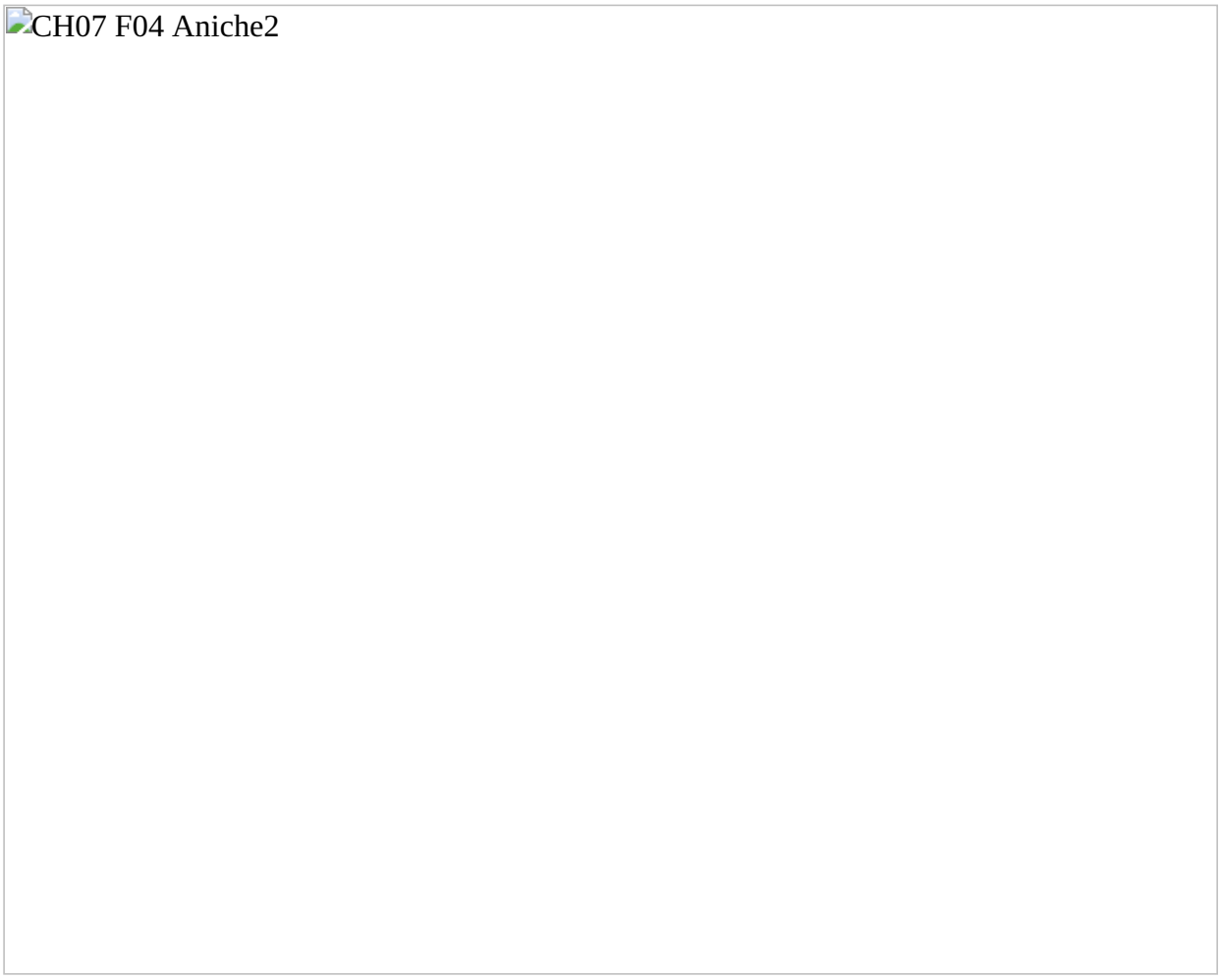
If another team develops the leaking module, it’s a kind gesture to drop that team a message and try to understand why it’s happening. There might be a valid reason behind it, or it could be an oversight, and your message could serve as a helpful refactoring suggestion.

7.3.2 Don’t depend on internal classes

In monoliths, it’s common for all modules to be in the same codebase, and all a developer needs to do to access classes from another module is to declare them in the package manager.

This grants development teams great power, because it requires virtually zero effort to start using a new module. However, it also places a significant responsibility on them to ensure that they don’t use classes that are meant to be internal to a specific module, as shown in figure 7.3.

Figure 7.3 Modules shouldn’t make use of internal classes of another modules.



Modules coupling themselves to the internal implementation of other modules is a recipe for disaster. If an internal class changes, this may break the modules that are coupled to it. Also, as soon as the team learns that other modules are coupled to internal details, they will refrain from changing these details because they don't want to break others. This may hinder their ability to keep improving their code.

I remember watching a talk from Marc Philipp (https://github.com/marcphilipp), one of the lead developers of JUnit 5, the most popular Java testing framework. In the talk, he mentioned that it was too hard to make changes in JUnit 4, as any change could break clients, including famous IDEs like Eclipse. He gave the audience a remarkable example: some IDEs were coupled to the field names of internal classes. This means an IDE could stop working if developers just renamed that field. This clearly illustrates how bad coupling to internal details is for everyone.

Teams can help other teams avoid making this mistake in a few ways. The first is to document the module's API thoroughly. It should be clear to other developers what they can use from that module. Some software designers make it explicit that all classes other modules use are in a `public` or `api` package, and no other class should be used.

Teams can also put tools in place to enforce these rules. For instance, ArchUnit is a well-known Java framework that allows developers to declare architectural constraints. If a module starts depending on a class from a forbidden package, the tool will alert the developers. The Java Modules system also helps developers by preventing code compilation in case they use something they shouldn't.

Depending on an internal class tends to occur when a module needs a functionality that's already implemented by another module but not exposed as a public API. Reusing entities is another common cause of such coupling. For example, if another module already implements an `Invoice` class well, it might be tempting to reuse it instead of reimplementing it.

Although this reuse seems logical and more efficient than reimplementing it, you should ask yourself: what if that team changes that class in a way that I don't expect or want? Remember that the two teams are likely separated for a reason. Both may have different stakeholders and different perspectives on the business. They may change `Invoice` in a way that makes sense to them but not to you.

If you find yourself in such a situation, I suggest you stop and think: can we make this functionality from this other module something other modules can use? Can we transform it into an API? Should we create a new small module to hold this new feature so that it becomes a standard library between both teams, and we both share the responsibility to keep it compatible?

If the answer to these questions is no, how bad would it be if we duplicate the functionality? Duplication can be problematic, but having to synchronize teams every time a class changes may cost you even more.

7.3.3 Monitor the web of dependencies

Monitoring the web of dependencies and how it grows is crucial, especially in large-scale systems with dozens or hundreds of modules. Before you know it, you may have modules coupled with others in ways you don’t want, becoming so intertwined that refactoring becomes virtually impossible.

Besides the maintenance cost, where changes in one module may impact another, complex dependency trees can result in longer build times. It’s not uncommon to see companies with build times exceeding an hour, mainly because the compiler struggles to recompile a large amount of code whenever a highly used module in the system is altered. Although build systems like Bazel have improved over time and become smarter about what to recompile, it’s best to address this issue at the root by preventing the dependency tree from growing wildly in the first place.

Modules that offer a clear API and are designed with flexibility and extensibility in mind—meaning modules that follow the principles discussed in this book—tend to be less problematic. Changes in their internals don’t trigger breaking changes or force other modules to recompile. However, we know that reality isn’t always ideal, and large software systems inevitably have modules that were not well-designed from the beginning. These modules require careful monitoring.

In my experience, the main reason for such a wild growth of coupling is that modules start depending on poorly designed ones just because they need "this one little thing they offer." My suggestion here is the same as in the previous section: search for opportunities not to depend on these large modules. Can you extract this functionality to another module, or at least expose it to clients in a less coupled and more elegant way? Note that there’s no clear answer or silver bullet for this problem, and you also don’t want to end up with a million tiny modules with a single class each. You have to explore, try, monitor, and improve.

7.3.4 Monoliths or microservices?

I’ve refrained from discussing whether modules should all live in a monolith or multiple individual microservices. There are great books about it, especially the two authored by Sam Newman, *Building Microservices* and *Monoliths to Microservices*, so I won’t delve too deeply into the topic.

All the principles discussed in this chapter apply to both approaches. It doesn’t matter whether it’s a monolith or a microservice; you don’t want them to become too intimate, neglect backward-compatibility, or offer a complex API.

Each approach has its challenges. Although it’s much easier to apply the "extract functionality to a module" principle in a monolith rather than in a microservices world, it’s also much easier to become too intimate with another module in a monolith world. Ultimately, you must pick your favorite approach and fight against its complexity growth.

7.3.5 Consider events as a way to decouple modules

Event-based architectures rose in popularity in the past years due to its great way of decoupling modules and services. The idea is that, instead of coupling modules with each other through calls, you publish an event announcing what just happened, and interested modules just subscribes to these stream of events.

For example, let’s say the Billing service makes an explicit call to the Delivery service whenever an invoice is marked as paid. This strongly couples Delivery with Billing. If Delivery changes something in the way it receives this paid notification, Billing may have to change as well.

In an event-driven architecture, Billing would publish an "InvoicePaid" event. The services that are interested in this event, such as Delivery, subscribe to the stream, and whenever a new event pops up, they get the event, and do their jobs. Note that Billing isn’t coupled to Delivery anymore. Billing has no idea who listens to these events. Delivery can change as much as it wants now, but Billing won’t ever be affected. Even new services can start listening to this event without requiring the Billing team.

As with any architectural decision, event-driven architectures design is also full of trade-offs. On the plus side, in addition to decoupling modules, it’s easier to listen and react to events than to provide APIs for every single action you need in your system. We have lots of good architectural patterns and infrastructure to support all this. On the negative side, it can quickly become hard to see all the events that are raised in the system. Monitoring also becomes trickier as following events requires more work than following a sequence of method calls. Backward-compatibility is still a thing, as if Billing changes the content of the event, it may affect modules that listen to it.

If you are curious about event-driven architectures, I suggest you to read *Building Event-Driven Microservices: Leveraging Organizational Data at Scale* by Adam Bellemare or *Microservices Patterns* by Chris Richardson.

7.3.6 Example: The notification system

Notifying participants about the trainings they enrolled in became core to PeopleGrow! These notifications can be sent through e-mail, direct messages, or even WhatsApp messages. The logic around it grew too much, prompting the company to decide that it was time to move notifications to another module and allocate it to a dedicated development team.

The assembled team decided it would be best if Notifications (the new module’s name) were implemented as a separate service. That would allow them to have a different release cycle and set up different SLOs (service level objectives).

The team drafted the first responsibilities of this new service:

- Sending notifications through different channels
- Supporting different notification templates and messages that can be customized by the clients of the service

The team decided that the service should be offered via a web API and would be accessible through a collection of endpoints. JSON was chosen as the format for communication.

In their first modeling session, the team focused on designing the core of their service: how clients should request a notification to be sent. They decided that the service must offer three different APIs to get started:

- The first offers clients a way to create a new notification. Say, they want to notify participants that enroll in a specific training. This API should then be created to create the notification. The API requires the message, the supported media (only e-mail? e-mail and chat), and when it should be sent (now, X days later, X days before the training starts, and so on).
- The second offers clients a way to add the list of participants to the notification. Clients should pass the ID of the notification (which was returned by the previous API) and the list of e-mails. The client can call this API as often as they want with new participants.
- The third offers clients a way to remove participants from a specific notification.

An interface that could represent this API is illustrated in listing 7.1. The team worked hard to ensure that this interface is simple and yet powerful, as we discussed in this chapter. See the three methods, one for each of the actions described above. `Medium` and `DispatchTime` are enumerations, giving users a specific list of options.

Listing 7.1 The Notification API

```
interface NotificationAPI { #1
        Notification createNotification(String message,
          List<Medium> supportedMedium, List<DispatchTime> times);

        void addParticipant(int notificationId, String participantEmail);

        void removeParticipant(int notificationId, String participantEmail);
}

enum Medium { #2
        EMAIL,
        CHAT,
        WHATSAPP;
}

enum DispatchTime { #3
        RIGHT_NOW,
        ONE_WEEK_BEFORE,
        DAY_BEFORE,
        ONE_HOUR_BEFORE;
}
```

To support backward-compatibility (another important topic discussed in this chapter), the team clarified that they can always add more items to the `Medium` and `DispatchTime` enumerators but can't remove any of them. The same is for the `Notification` structure that's returned to the clients whenever they create a new notification: fields can be added but not removed, nor have their semantics changed.

Behind the scenes, the team decided the API would be implemented in Java, with Spring Boot as the framework and Postgres as the database of choice. There'll be an asynchronous job that will be pooling the database for the following notifications to be sent. The team considered introducing a "proper" queue like RabbitMQ but decided that this could be done later. The API doesn't let internal details leak (another principle from this chapter), so this refactoring can eventually be done without breaking the clients.

Security concerns

When designing such an API in a real-world system, you'd have to take care of authentication and authorization, just so only users with the proper permissions can make calls to the API. Security goes beyond the scope of this book, but an interesting thing to think about is whether such security aspects would have to be reflected in the design of the API itself. I'll leave this as an exercise for you.

7.4 Exercises

Discuss the following questions with a colleague:

7.1. Have you ever worked on a microservices or monolithic project with multiple modules? What were the main challenges related to how they were designed to work together?

7.2. In the project you currently work, how intimate are the modules? What should you do to reduce such intimacy?

7.3. Have you ever encountered a problem due to bad backward compatibility? What caused it? What did you do to avoid going through the same problem again?

7.4. What other practices does your company have around ensuring that modules (or services) work well together?

7.5 Summary

- Modules should provide simple interfaces on top of complex features.
- A good module doesn't force its clients to change whenever its internal details change.
- Modules should provide clear communication interfaces that are stable and backward-compatible. If flexibility is needed, modules should offer easy-to-use extension points.
- Don't let modules know about each other's details. For that to happen, modules should do their best to hide their details, and clients should do their best not to couple to any leaked detail.
- Modules should have clear ownership and engagement rules to simplify communication among the different teams working on different modules.

8 Being pragmatic

This chapter covers

- Being pragmatic matters
- Never stop refactoring
- Never stop learning about object-oriented design

Congratulations, you finished this long journey on the six most important characteristics of a simple object-oriented design:

- small units of code
- consistent objects
- proper dependency management
- good abstractions
- properly handled infrastructure
- well modularized

In this chapter, I'll discuss a few pieces of advice that I also consider essential to keep object-oriented designs as simple as possible. Some of them were briefly mentioned in previous chapters, but they are so crucial that I've dedicated a separate section to them.

8.1 Be pragmatic and go only as far as you need

It's easy to get stuck in an infinite loop of design improvements, especially for engineers who appreciate a well-designed system. Even simple design decisions can branch out into numerous possibilities.

Striving for the best possible design is essential for a highly maintainable and simple object-oriented system. However, it's important to remember that our primary goal is not to write beautiful designs but to deliver functional software as efficiently as possible. A good design enables that goal, but it's not the end.

Finding the right trade-off between exploring the perfect design and settling for a "good enough" design is a challenging task that improves with experience over time.

8.2 Refactor aggressively but in small steps

Never stop refactoring. Refactoring is the most effective tool to combat the growth of complexity. The more frequently you refactor, the cheaper and quicker it becomes, because you'll have less code to refactor and your skills will improve.

Engineers less enthusiastic about constant refactoring may argue that it wastes time. They might ask, "Why should I rename this variable? It makes no difference," or "This is just nitpicking!" However, once you realize that refactoring is an investment with significant returns, you'll find it easier to embrace.

Nevertheless, as we discussed earlier, pragmatism is essential. Don't overdo it. You don't need to refactor every bit of functionality that is easy to maintain. You may not need to refactor — at least urgently — pieces of code that never change or that have proven themselves in production. Pay attention to the signals, listen to your code, and refactor when necessary.

8.3 Accept that your code won't ever be perfect

Life as an engineer becomes much easier when you realize that your code, design, and architecture will never be perfect. We do our best with the information and resources available at a given moment. As we learn more about the system, we may discover better possibilities.

As I've mentioned before, your code doesn't have to be perfect. "Good enough", as we've discussed in all the previous chapters, is sufficient in most cases. However, you should never stop writing better code every day.

8.4 Consider re-designs

Don’t dismiss the possibility of re-designs, especially in the early stages of development when refactoring is still cost-effective.

I understand the need for pragmatism. Re-designing and re-implementing something can be costly. However, if you can see that the current approach is leading you to a dead end, it’s best to refactor it as soon as possible.

I’ve encountered code bases where developers talked about re-designing parts of the system for years, but they never found the time to do it. Consequently, the codebase turned into a tangled mess. If you wait until the code becomes unbearable, you may reach a point where refactoring is no longer feasible.

John Osterhout’s book (https://web.stanford.edu/~ouster/cgi-bin/book.php) states that he only makes very good design decisions on the third attempt at designing something. In other words, it takes him two versions to understand what the design should truly look like. Therefore, as soon as you realize there’s a much better way to do something, start planning the refactoring.

8.5 You owe this to your junior developers

I’ve met talented developers who spent their early years in companies that didn’t prioritize code quality. Consequently, these developers faced challenges in upskilling themselves when they moved on to other jobs.

Novice developers learn a lot from the code they work on and the engineers they are surrounded by. It’s our responsibility to help the newer generations understand the importance of good code. We achieve this by setting an example, constantly working to reduce complexity in our code, and continually seeking ways to improve the design of our systems.

8.6 Never stop learning

There are many books about object-oriented design, and I learned a lot from them. I strongly recommend you read other perspectives on it:

- *Domain-Driven Design: Tackling Complexity in the Heart of Software*, by Eric Evans (https://www.domainlanguage.com/ddd/)
- *Implementing Domain Driven Design*, by Vaughn Vernon (https://vaughnvernon.com/)
- *A Philosophy of Software Design*, by John Ousterhout (https://web.stanford.edu/~ouster/cgi-bin/book.php)
- *Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software*, by Eric Freeman and Elisabeth Robson (https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124)
- *Building Maintainable Software, Ten Guidelines for Future-Proof Code*, by Joost Visser et al. (https://www.softwareimprovementgroup.com/publications/ebook-building-maintainable-software/)
- *Refactoring to Patterns*, by Joshua Kerievsky (https://www.industriallogic.com/xp/refactoring/)
- *Object Design Style Guide*, from Python to PHP, by Matthias Noback (https://matthiasnoback.nl/book/style-guide-for-object-design/)
- *Refactoring: Improving the Design of Existing Code*, by Martin Fowler (https://martinfowler.com/books/refactoring.html)
- *Object-Oriented Analysis and Design with Applications*, by Grady Booch et al. (https://www.amazon.nl/-/en/Booch-Grady/dp/020189551X)
- *Growing Object-Oriented Software, Guided by Tests*, by Steve Freeman and Nat Pryce (http://www.growing-object-oriented-software.com/)
- *Team Topologies*, by Matthew Skelton and Manuel Pais (https://teamtopologies.com/)

These are the books that really had an impact on me, but I'm sure there are many others. Never stop learning about object-oriented design!

8.7 Exercises

Discuss the following questions with a colleague:

8.1. How often do you refactor your design? Are you "refactoring addicted" or refactoring is something you only do when really needed?

8.2. Being pragmatic isn’t easy. How do you decide when it’s time to go for a more elegant design solution?

8.3. People say "there’s nothing more permanent than a temporary workaround". How often have you seen a workaround becoming permanent in the code base? Why do you think that happened? Was there something one could do to avoid it from happening?

8.4. Have you ever re-designed (parts of) a software system? What challenges did you face? Did you benefit from it in the end?

8.8 Summary

- Designing object-oriented systems is a fine art, one that requires mastering.
- Complexity tends to grow in any software system. It’s our job to keep fighting against this growth. It’ll cost less if done every day.
- A simple object-oriented system keeps its code simple, objects consistent, manages dependencies properly, has good abstractions, handles infrastructure properly, and is well modularized.
- We should strive for the perfect design, but knowing it doesn’t exist. "Good enough" designs are often what we need.
- This book shows my take on it after 20 years of building good and bad software systems. There are many other good takes on it, which you should read and form your opinions on what constitutes a good design.

About this MEAP

You can download the most up-to-date version of your electronic books from your Manning Account at account.manning.com. For customer support write to support@manning.com.

https://livebook.manning.com/#!/book/simple-object-oriented-design/discussion

Welcome letter

Dear reader,

Thanks for purchasing the MEAP of *Simple Object Oriented Design*. I'm really happy you're curious about my thoughts on object-oriented design!

You'll quickly see that my main goal with this book isn't to discuss what a perfect design or architecture looks like. After almost 20 years of trying to achieve perfection, I don't believe in it that much anymore. Real-life problems always come up with ways to break up whatever great design idea you had. Coming up with the perfect design might also be too expensive, and while designing is fun, we do it because we want to deliver software that works and that's easy to evolve. Instead, you'll see that we'll be talking a lot about pragmatic design.

What I learned over the past years is that the complexity of our code will keep growing if we don't do anything to prevent it. It grows not because we write bad code, but because our systems keep getting more complex. This book is all about principles and patterns that I applied over the years so that I could keep the overall complexity of my design under control.

I divided this book into six chapters, each dedicated to a different type of complexity I believe we should keep under control as much as we can:

- Code complexity: at the implementation level, we should make sure that our code is small and easy to understand
- Keeping objects consistent: making sure that objects are also in a consistent state is key to maintenance, but also quite challenging as our objects often represent complex business data.
- Managing dependencies: classes have to depend on other classes so that, together, they deliver complex functionality, but if we just let classes couple to others as they please, we end up with a highly fragile design.

- Creating good abstractions: we gotta design classes that have extension points in the right places so that we can add new functionality by adding more simple code and not by making existing code more complex.
- Achieving modularisation: when your system becomes too large, it's fundamental that you break it down into highly cohesive components that communicate with each other. All the ideas from the previous chapter apply here, just on a different level of granularity.
- Handling infrastructure: applications rarely live in an isolated bubble, they need databases, they consume web services, and we gotta make sure our designs don't depend so much on these details and yet use the infrastructure to their best.

If you are already well versed in object-oriented best practices, you'll notice a lot of influence from authors like Robert Martin (and his SOLID principles), Eric Evans (and his canonical work on domain-driven design), Grady Booth (and his foundational texts on object-oriented design), Steve Freeman and Nat Pryce (and their awardwinning GOOS book). I stand on the should of these giants. However, I give my own pragmatic twist to all their ideas. So, even if you know their work, I'm sure you can find interesting ideas here.

And this is still the early version. I'm writing this book as we speak. I'm open to all types of feedback. Please, drop your thoughts in the liveBook Discussion forum.

Happy reading!

— Maurício Aniche

In this book

1 It’s all about managing complexity

This chapter covers

- Why software systems get more complex over time
- The different challenges in object-oriented design
- Why we should keep improving our design over time

Information systems have been simplifying so much of our lives. Imagine going to the doctor and having them manually search for the patient’s history. Or going to any store and waiting for the cashier to sum up the price of the items we’re buying. Fortunately, this isn’t the case for a long time now. We use software systems to help us organize complex information and have them available at the snap of our fingers. We, software engineers, must develop such systems and ensure that they can keep evolving.

Building a highly maintainable information system requires good design. Modern businesses tend to be quite extensive, as companies often operate on different fronts and offer different products; to be complex, with business rules full of variations; and evolve fast, requiring developers to keep implementing or changing existing functionality.

Luckily, we don’t have to invent best practices for object-oriented systems from scratch, as our community already has extensive knowledge on the topic. Think of the Gang of Four’s *Design Patterns*, *SOLID principles*, *Clean Architecture* by Robert Martin, Martin Fowler’s refactoring techniques, *Hexagonal Architecture* by Alistair Cockburn, *Domain-Driven Design* by Eric Evans, object-oriented design styles by Matthias Noback, or at an even higher level of abstraction, service decomposition patterns by Sam Newman.

Throughout the years, I learned a lot from my good and bad decisions, especially when applying these principles to information systems. If you don’t know these principles, no worries. They aren’t a pre-requisite to read this book. But I recommend you read about them once you finish this book. If you already know these principles, this book will give you a different and more pragmatic view of them.

Applying any best practice or principle by the book is always difficult. Trade-offs have to be made. Where should I code this business rule? Is this code simple enough, or should I propose a more elegant abstraction? How should I model the interaction between my system and this external web app? Should I make this class depend on this other class, or is that bad coupling? These are all common questions that emerge in the minds of any developer that’s building an information system.

In this book, I’ll share with you my set of patterns that have been helping me deliver high-quality information systems.

Why is this book called "*Simple Object-Oriented Design*," you may ask? Because simple object-oriented designs are always easier to maintain. The main challenge is not to come up with a simple design but keep it this way. As I'll discuss throughout this book, too many forces push complexity down your throat. If you don't explicitly work toward keeping design simple, you'll end up in no time with a big ball of mud (http://www.laputan.org/mud/mud.html) that's hard to maintain.

1.1 Designing simple object-oriented systems

"As software systems evolve, their complexity increases unless work is done to maintain or reduce it." Evolving software systems of any type isn't that straightforward. We know that code tends to decay over time, requiring effort to maintain since the 1980s. This insight comes from Lehman's famous paper on the "laws of software evolution" (https://www.sciencedirect.com/science/article/abs/pii/0164121279900220). And despite 40 years of progress, the maintainability of software systems remains a challenge.

In essence, maintainability is the effort you need to complete tasks like modifying business rules, adding features, identifying bugs, and patching the system. Highly maintainable software enables developers to perform such tasks with reasonable effort, whereas low maintainability makes tasks too difficult, time-consuming, and bug-prone.

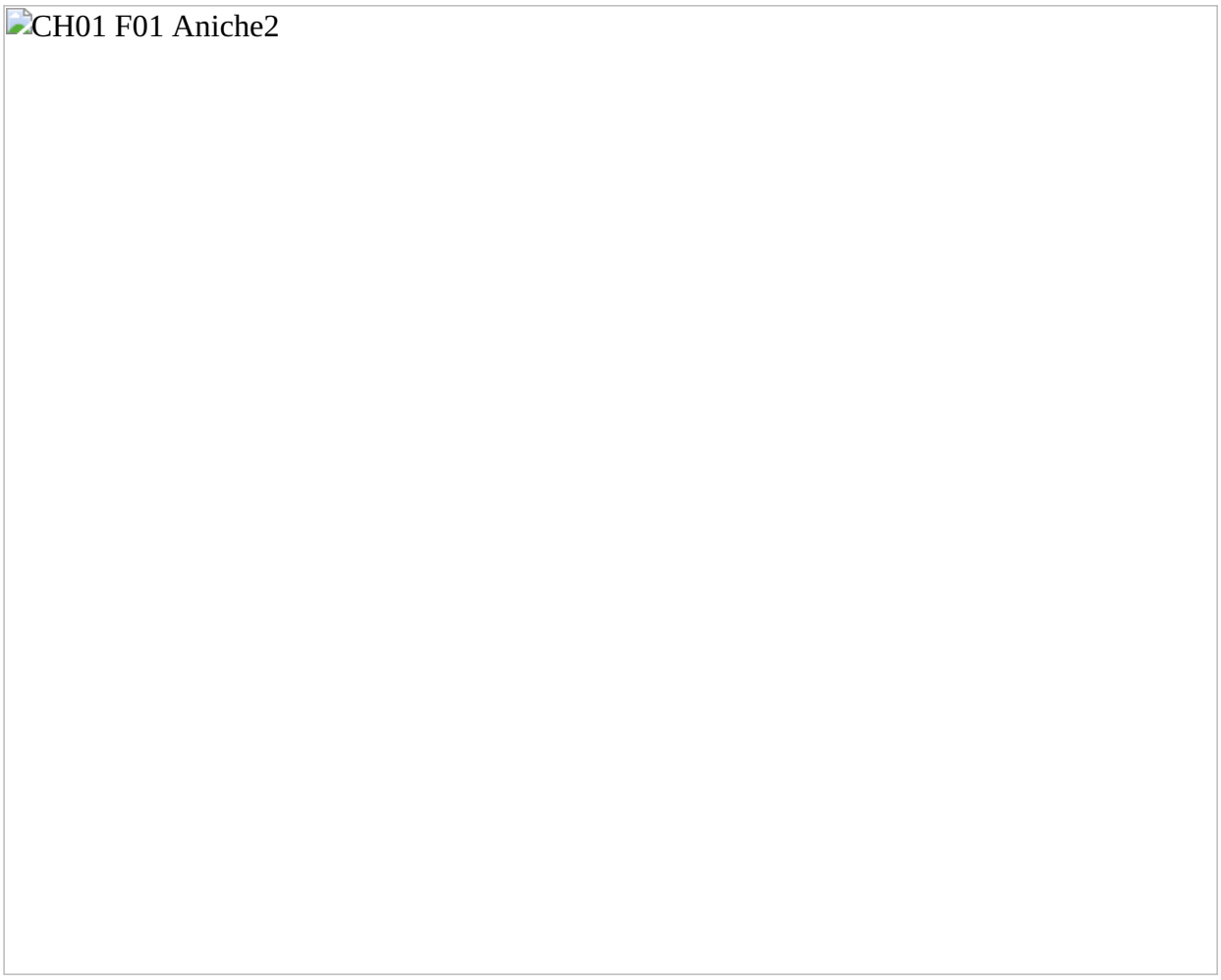
Many factors affect maintainability, from overly complex code to dependency management, poorly designed abstractions, and bad modularization. Systems naturally become more complex over time, so continually adding code without considering its consequences on maintenance can quickly lead to a messy codebase.

Consistently combatting complexity growth is crucial, even if it seems more time-consuming. And I know: it's much more effort than simply "dumping code." But trust me, developers feel way worse when handling big balls of mud the entire day. You may have worked on codebases that are hard to maintain. I did. Doing anything in such systems takes a lot of time. You can't find where to write your code, all code you write feels like it's a workaround, you can't write an automated test for it because the code is untestable, and you are always afraid something will go wrong because you never feel confident about changing it, and on it goes.

What constitutes a simple object-oriented design? Based on my experience, it's a design that presents the following six characteristics, also illustrated in figure 1.1:

- small units of code
- consistent objects
- proper dependency management
- good abstractions
- infrastructure properly handled
- well modularized

Figure 1.1 Characteristics of a simple object-oriented design



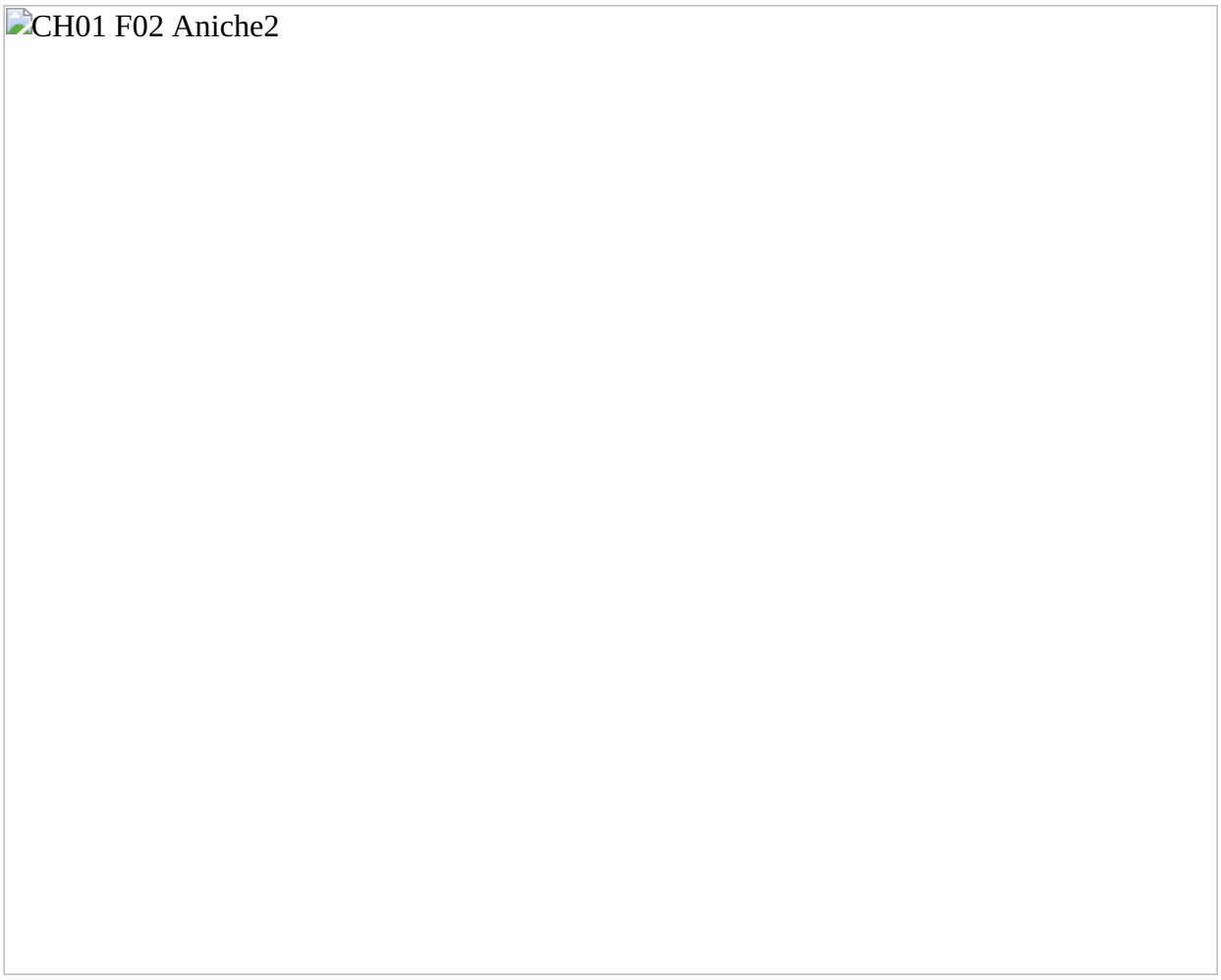
These ideas may sound familiar to you. They are all popular in object-oriented systems. Let’s look at what I mean by each of them and what happens when we lose control, all in a nutshell.

1.1.1 Small units of code

Keeping implementing methods and classes that are simple in essence is a great way to start your journey toward maintainable object-oriented design. Consider a method that began as a few lines with a few conditional statements but grew over time and currently has hundreds of lines and `if`s inside of `if`s. Maintaining such a method is just tricky.

Interestingly, classes and methods usually start simple and manageable. But if we don’t work to keep them like this, they become hard to understand and maintain, like in figure 1.2. Complex code tends to result in bugs, as they are drawn to complex implementations that are difficult to understand. Complex code is also challenging to maintain, refactor, and test, as developers fear breaking something and struggle to identify all possible test cases.

Figure 1.2 Simple code becomes complex over time and, consequently, very hard to maintain.



There are many ways to reduce the complexity of a class or method. For example, clear and expressive variable names help developers to better understand what's going on. However, what I'm going to argue in this part of the book is that the number one rule to keep classes and methods simple is to keep them small. A method shouldn't be too long. A class shouldn't have too many methods. Smaller units of code are always easier to be maintained and evolved.

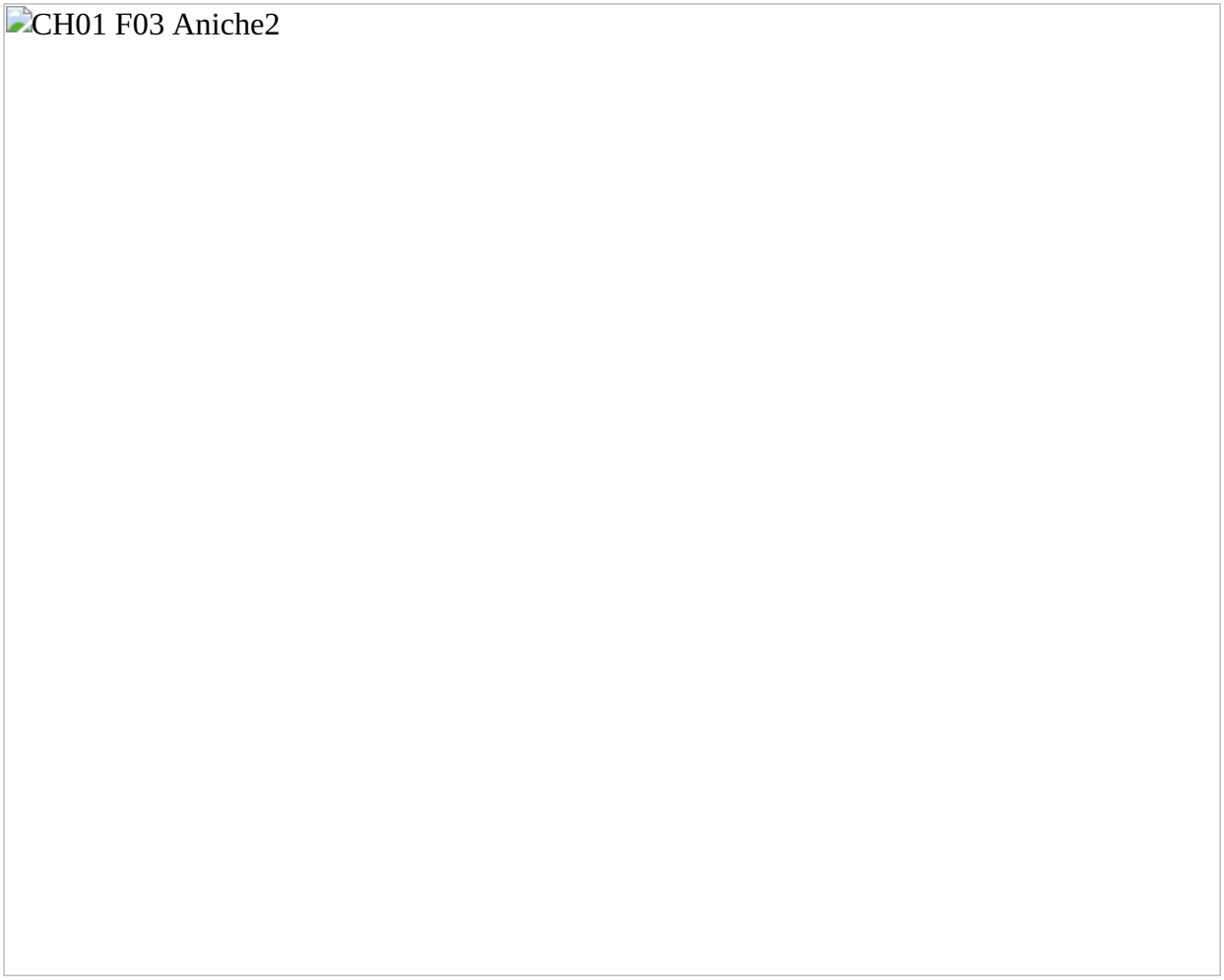
1.1.2 Consistent objects

It's much easier to work on a system where you can trust that objects are always in a consistent state and any attempt to make them inconsistent is denied. When consistency isn't accounted for in the design, objects may hold invalid states, leading to bugs and maintainability issues.

Consider a Basket class in an e-commerce system that tracks the products a person is buying and their final value. The total value must be updated whenever we add or remove a product from the basket. The basket should also reject invalid client requests, like adding a product -1 times or removing a product that isn't there.

In figure 1.3, the left side shows a protected basket, where items can only be added or removed by asking the basket itself. The basket is in complete control and ensures its consistency. On the right side, the unprotected basket allows unrestricted access to its contents. Given the lack of control, that basket can’t always ensure consistency.

Figure 1.3 Two baskets, one that has control over the actions that happen on it, another one that doesn’t. Managing state and consistency is fundamental.



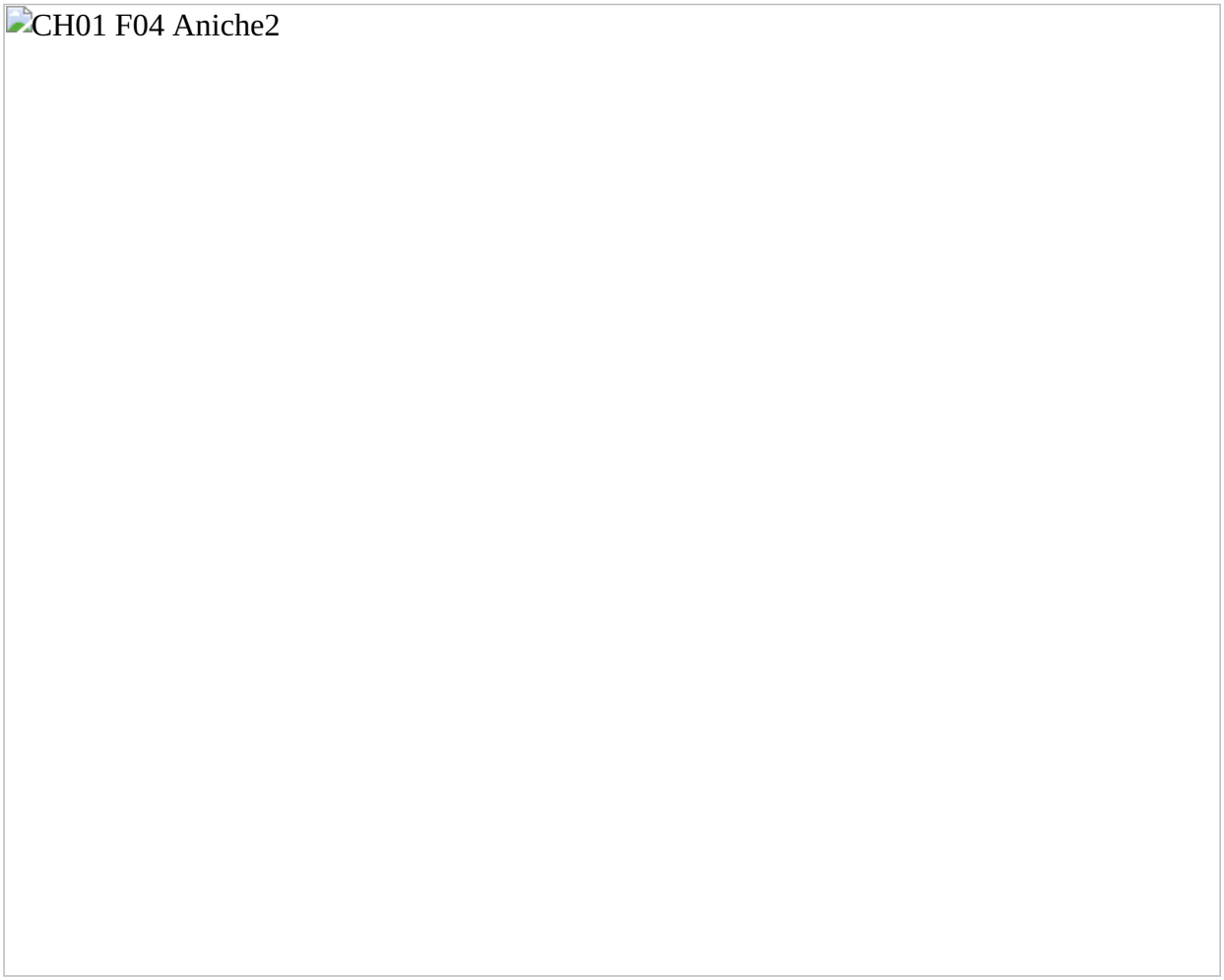
We’ll see that a good design ensures objects can’t ever be in an inconsistent state. Consistency mishandling can happen in many different ways, such as improper setter methods that bypass consistency checks or the lack of flexible validation mechanisms, which we’ll discuss in more detail later.

1.1.3 Proper dependency management

In large-scale object-oriented systems, dependency management becomes key to maintainability. In a system where the coupling is high and no one cares about "which classes are coupled to which classes," any simple change may have unpredicted consequences.

Figure 1.4 shows how the `Basket` class may be impacted by changes in any of its dependencies: `DiscountRules`, `Product`, `Customer`. Even a change in `DiscountRepository`, a transitive dependency, may impact `Basket`. If, say, class `Product` frequently changes, `Basket` is always at risk of having to change as well.

Figure 1.4 Dependency management and change propagation



Simple object-oriented designs aim to minimize dependencies among classes. The less they depend on each other and the less they know about each other, the better. Good dependency management also ensures that our classes depend as much as possible on stable components that are less likely to change and, therefore, less likely to provoke cascading changes.

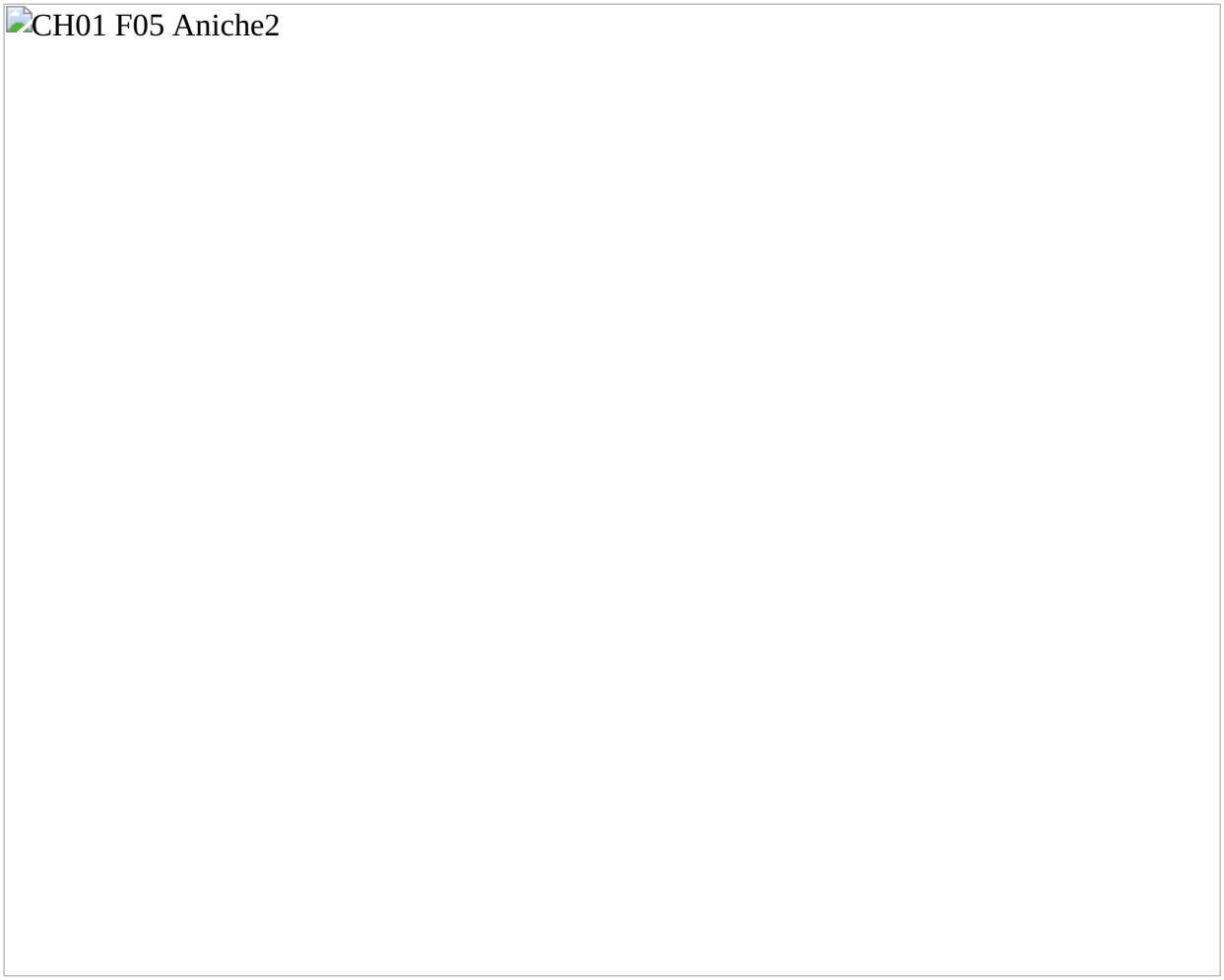
1.1.4 Good abstractions

Simple code is always preferred, but it may not be sufficient for extensibility. Extending a class by adding more code at some point stops being effective and becomes a burden.

Imagine implementing 30 or 40 different business rules in the same class or method. I illustrate that in figure 1.5. Note how the `DiscountRule` class, a class that's responsible for

applying different discounts in our e-commerce system, grows as new discount rules are introduced, making the class much harder to maintain.

Figure 1.5 A class that has no abstractions grows in complexity indefinitely.

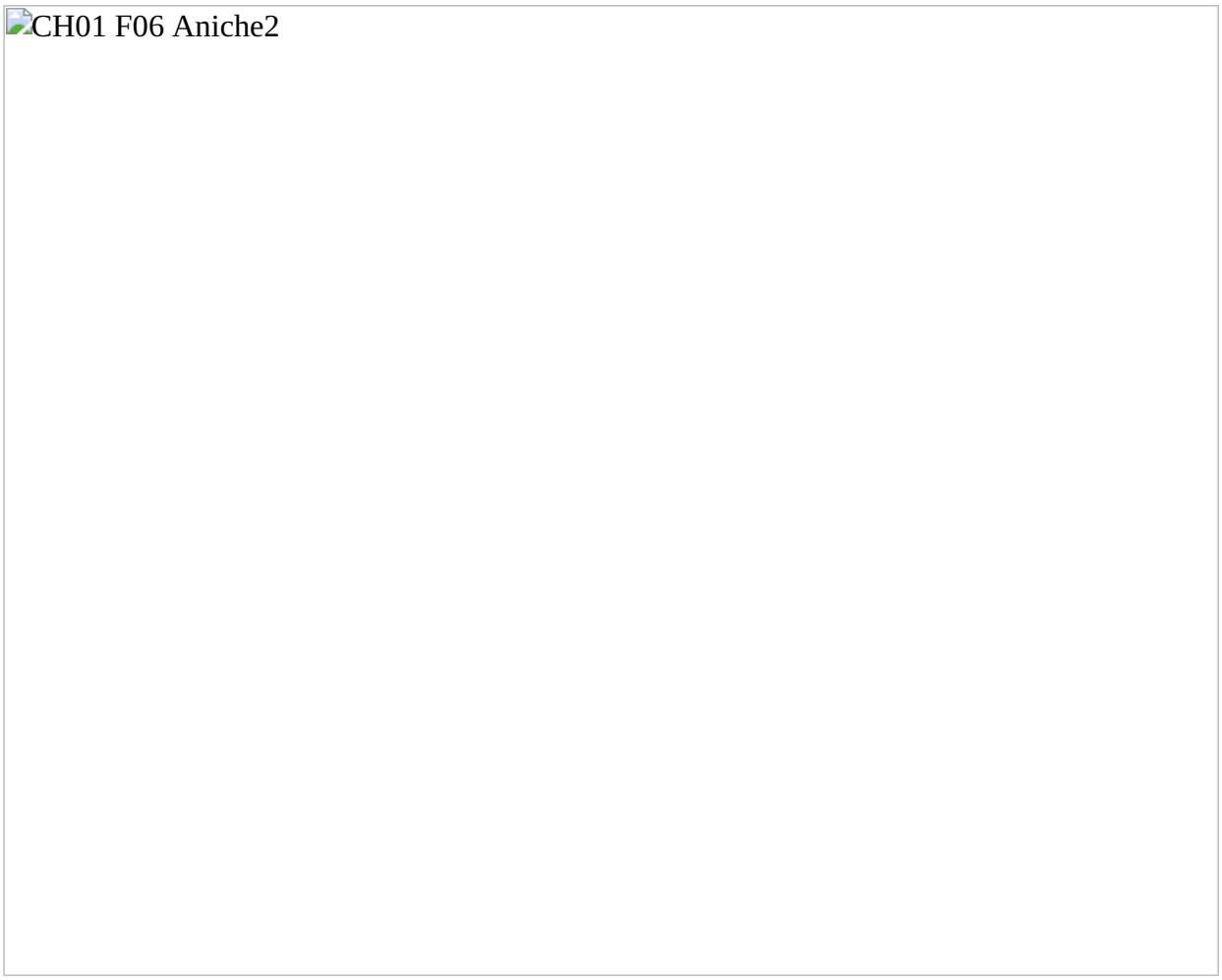


A good design provides developers with abstractions that help them evolve the system without making existing classes more complex.

1.1.5 Infrastructure properly handled

Simple object-oriented designs separate domain code, that contains business logic, from infrastructure code required for communication with external dependencies. Figure 1.6 shows domain classes on the left and infrastructure classes on the right.

Figure 1.6 The architecture of a software system that separates infrastructure from domain code.



Letting infrastructure details leak into your domain code may hinder your ability to make changes in the infrastructure. Imagine all the code to access the database is spread through the code base. Now, you decide you need to add a caching layer to speed up the application's response time. You may have to change the code everywhere for that to happen.

The challenge is to find the right abstraction for the infrastructure code. For instance, if you are using a relational database like Postgres, you may want to completely hide its presence from the domain code, but doing so could limit access to its unique features that enhance productivity or performance. The key is to abstract irrelevant aspects while leveraging valuable features that are provided by your infrastructure.

1.1.6 Well modularized

As software systems grow, fitting everything into a single component or module is challenging. Simple object-oriented designs divide large systems into independent components that interact to achieve a common goal.

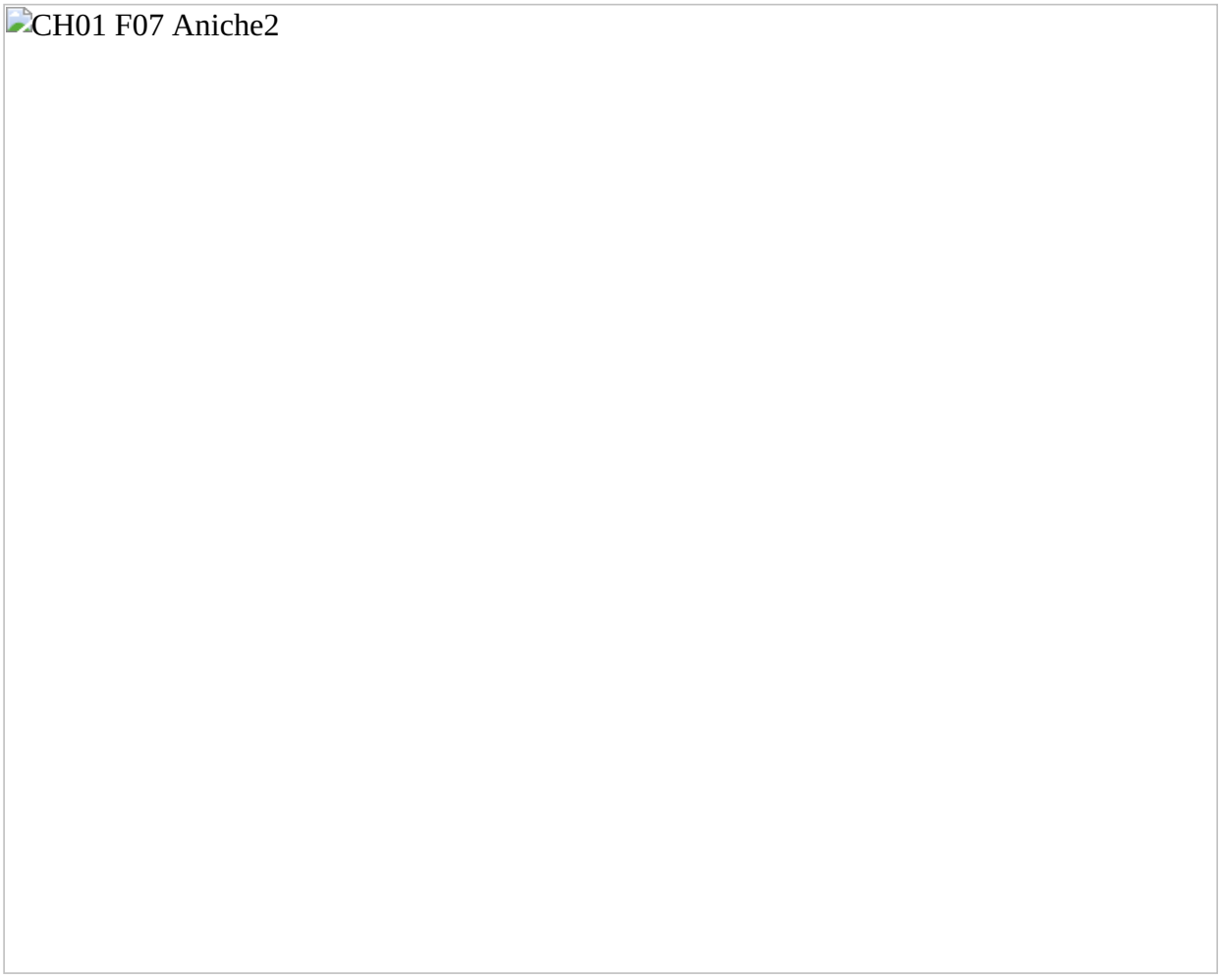
Dividing systems into smaller components makes them easier to maintain and understand. It also helps different teams work on separate components without conflicts. Smaller

components are more manageable and testable.

Consider a software system with three domains: Invoice, Billing, and Delivery. These domains must work together, with Invoice and Delivery requiring information from Billing.

Figure 1.7 shows a system without modules on the left, where classes from different domains mix freely. As complexity increases, this becomes unmanageable. The right side of the figure shows the same system divided into modules—Billing, Invoice, and Delivery. Modules interact through interfaces, ensuring clients only use what's needed without understanding the entire domain.

Figure 1.7 Two software systems with different modularization approaches.



It's not easy to identify the right level of granularity for a module or what its public interface should look like, which we'll talk more about later.

1.2 Simple design as a day-to-day activity

As I said before, the hard part is often not to design a simple design but to keep it that way. We must keep focusing on improving and simplifying our designs as we learn about the system and the more it evolves.

1.2.1 Reducing complexity is similar to personal hygiene

Constantly working towards simplifying the design can be compared to brushing your teeth. While not particularly exciting, it’s necessary to avoid discomfort and costly problems in the future. Similarly, investing a little time in code maintenance daily helps prevent more significant issues down the line.

1.2.2 Consistently addressing complexity is cost-effective

Regularly addressing complexity keeps both the time and cost associated with it within reasonable limits. Delaying complexity management can result in significantly higher expenses and make refactoring more difficult and time-consuming.

The "technical debt" metaphor helps us understand this. Coined by Ward Cunningham, the idea is to see coding issues as financial debt. The additional effort required for maintenance due to past poor decisions represents interest on this debt. This concept relates closely to the book’s focus; complexity escalates if code structure isn’t improved, leading to excessive interest payments.

I’ve been to code bases where everyone knew that parts of it were very complex and hard to maintain, but no one dared to refactor it. Trust me, you don’t want to get there.

1.2.3 High-quality code promotes good practices

When developers work with well-structured code with proper abstractions, straightforward methods, and comprehensive tests, they are more likely to maintain its quality. Conversely, messy code often leads to further disorganization and degradation in quality. This concept is similar to the broken-window theory, which explains how maintaining orderly environments can prevent further disorder.

1.2.4 Controlling complexity isn’t as difficult as it seems

The key to ensuring the complexity doesn’t grow out of hand is recognizing patterns and addressing them early on. With experience and knowledge, developers can detect and resolve most issues in the initial stages of development.

1.2.5 Keep the design simple is a developer’s responsibility

Creating high-quality software systems that are easy to evolve and maintain can be challenging but necessary. As developers, managing complexity is part of our job and contributes to more efficient and sustainable software systems.

Striking the right balance between manageable complexity and overwhelming chaos is challenging. Starting with complex abstraction might prevent issues, but it adds system complexity. A simpler method with two `if` statements is easier to understand than a convoluted interface. On the other hand, at some point, simple code isn't enough anymore. Striving for extensibility in every code piece would create chaos.

It's our job to find the right balance between simplicity and complexity.

1.2.6 Good enough designs

John Ousterhout, in "A Philosophy of Software Design" (https://web.stanford.edu/~ouster/cgi-bin/book.php), says that it takes him at least three rewrites to get to the best design for a given problem. I couldn't agree more.

Often, the most effective designs emerge after several iterations. In many cases, it's more practical to focus on creating "good enough designs" that are easily understood, maintained and evolved rather than striving for perfection from the outset. Again, the key is to identify when simple isn't enough anymore.

1.3 A short dive into the architecture of an information system

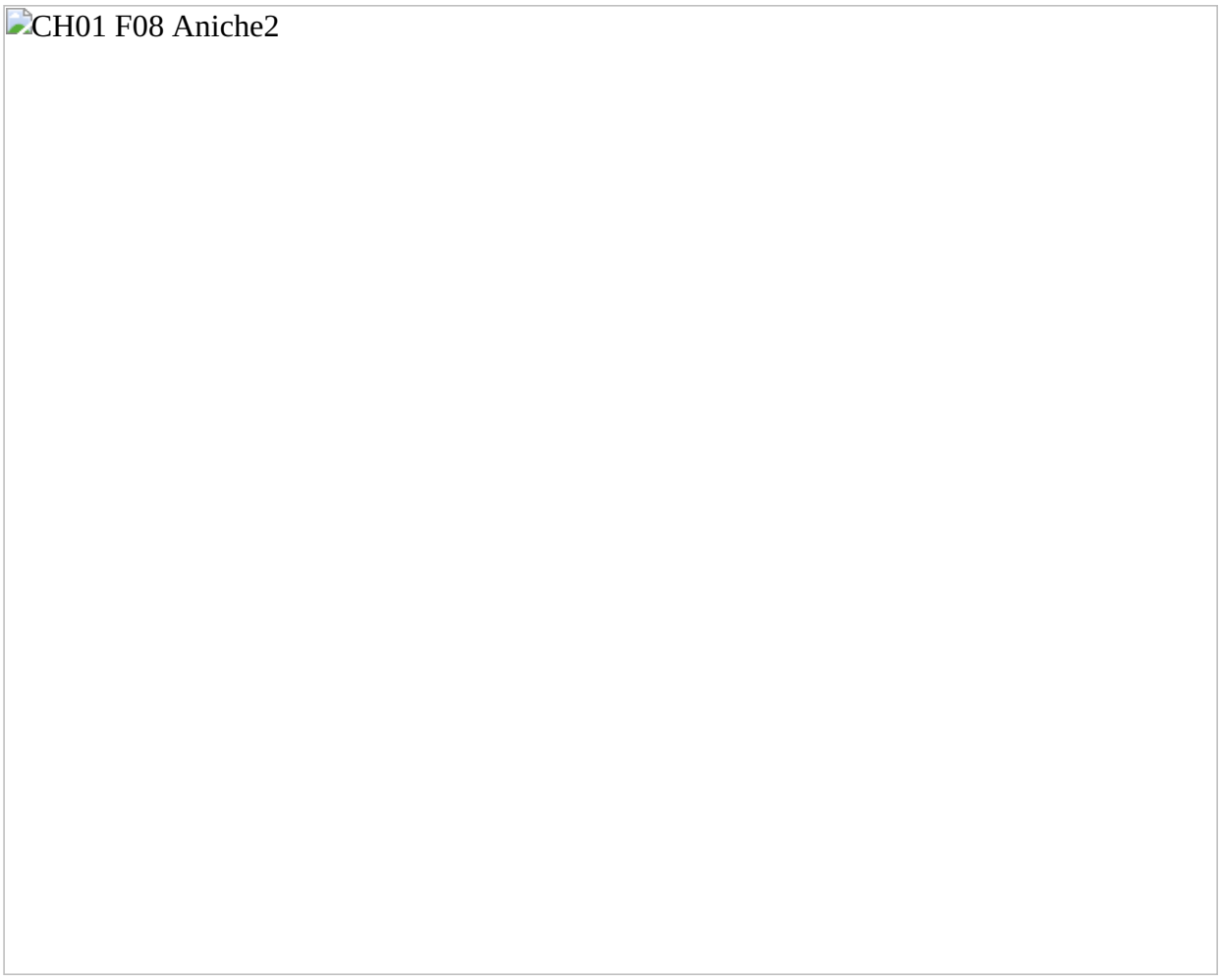
Before discussing the different patterns to help you keep your design simple, let me define a few terms.

First, while I started this chapter by talking about information systems, I'll focus on the back-end part of such systems. I'm using the term back-end to identify the software that runs behind the scenes of every information system.

Figure 1.8 illustrates a traditional information system. Back-end systems are often characterized by the following:

- Having a front-end that displays all the information to the user. This front-end is often implemented as a web page. Developers can use many technologies to build modern front-ends, such as React, Angular, VueJS, or even plain vanilla JavaScript, CSS, and HTML.
- Having a back-end that handles the requests from the front-end. The back-end is the place where most, if not all, business logic lives. The back-end may communicate with other software systems to achieve its tasks, such as external or internal web services.
- Having a database that stores all the information. Back-end systems are strongly database-centric. This means most back-end actions involve retrieving, inserting, updating, or deleting information from the database.

Figure 1.8 A traditional information system with front-end, back-end, and database.



Let me also define the inside of a back-end system, so we can agree on a few terms. See figure 1.9. A back-end system receives requests via any protocol, but usually HTTP, from any other system, for example, a traditional web-based front-end.

This request is first received by a Controller. The Controller's primary responsibility is to convert the user request into a series of commands to the domain model that knows the business rules.

The domain model is composed of different types of classes. This depends on the architectural patterns your application is following, but you often see:

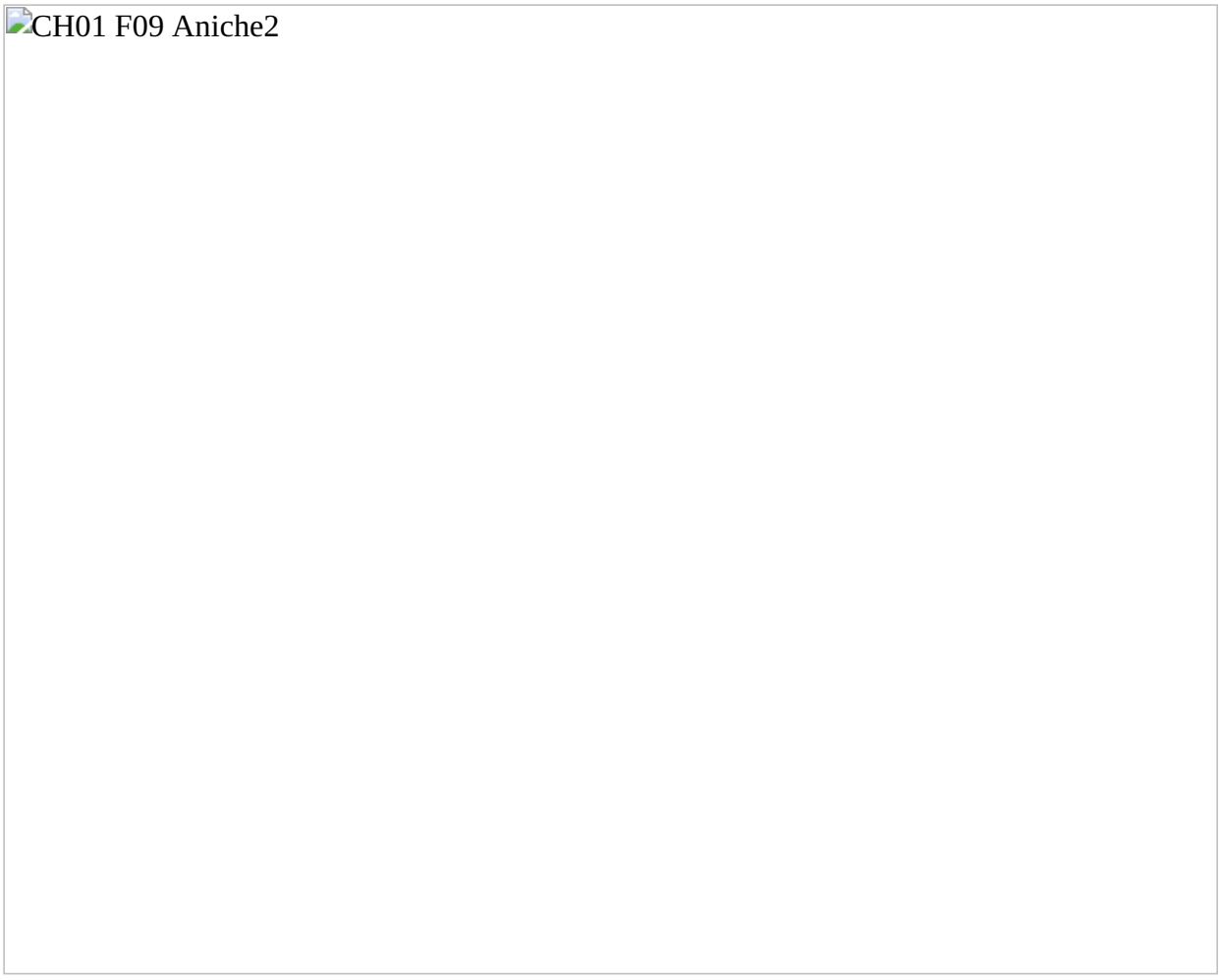
- Entities, that model the business concepts. Think of an `Invoice` class that models what invoices mean to the system. Entity classes contain attributes that describe the concept and methods that consistently manipulate these attributes.
- Services, that encapsulate more complex business rules that involve one or more entities. Think of a `GenerateInvoice` service responsible for generating the final invoice for a user that just decided to pay for all the products in their basket.
- Repositories, which contain all the logic to retrieve and persist information. Behind the scenes, their implementation talks to a database.

- Data Transfer Objects (DTOs), that are classes that hold some information and are used to transfer information from different layers.
- Utility classes, which contain a set of utility methods that are not offered by your programming language or framework of choice.

Back-ends also commonly have to communicate with other external applications, usually via remote calls or specific protocols. Think of a web service that enables the application to send a request to a governmental system. Or to an SMTP server that enables the application to send e-mails. Anything that's outside and somewhat out of control of the back-end, I'll call infrastructure.

A large-scale back-end may also be organized in modules. Each module contains its own domain model with entities, repositories, services, etc. Modules may also send messages to each other.

Figure 1.9 The internal design of a back-end system.



If you're familiar with architectural patterns such as Clean Architecture, Hexagonal Architecture, Domain-Driven Design, or any other layered architecture, you may have

opinions on how things should be organized inside. This diagram is meant to be generic enough so that anyone can fit it into their favorite architecture.

The figure also shows the two modules inside the same back-end, which may lead you to think I'm proposing monoliths over a service-distributed architecture. Again, this figure is meant to be generic. Different modules can live in the same distributed binary and communicate via simple method calls or distributed over the network and communicate via remote procedure calls. It doesn't matter at this point.

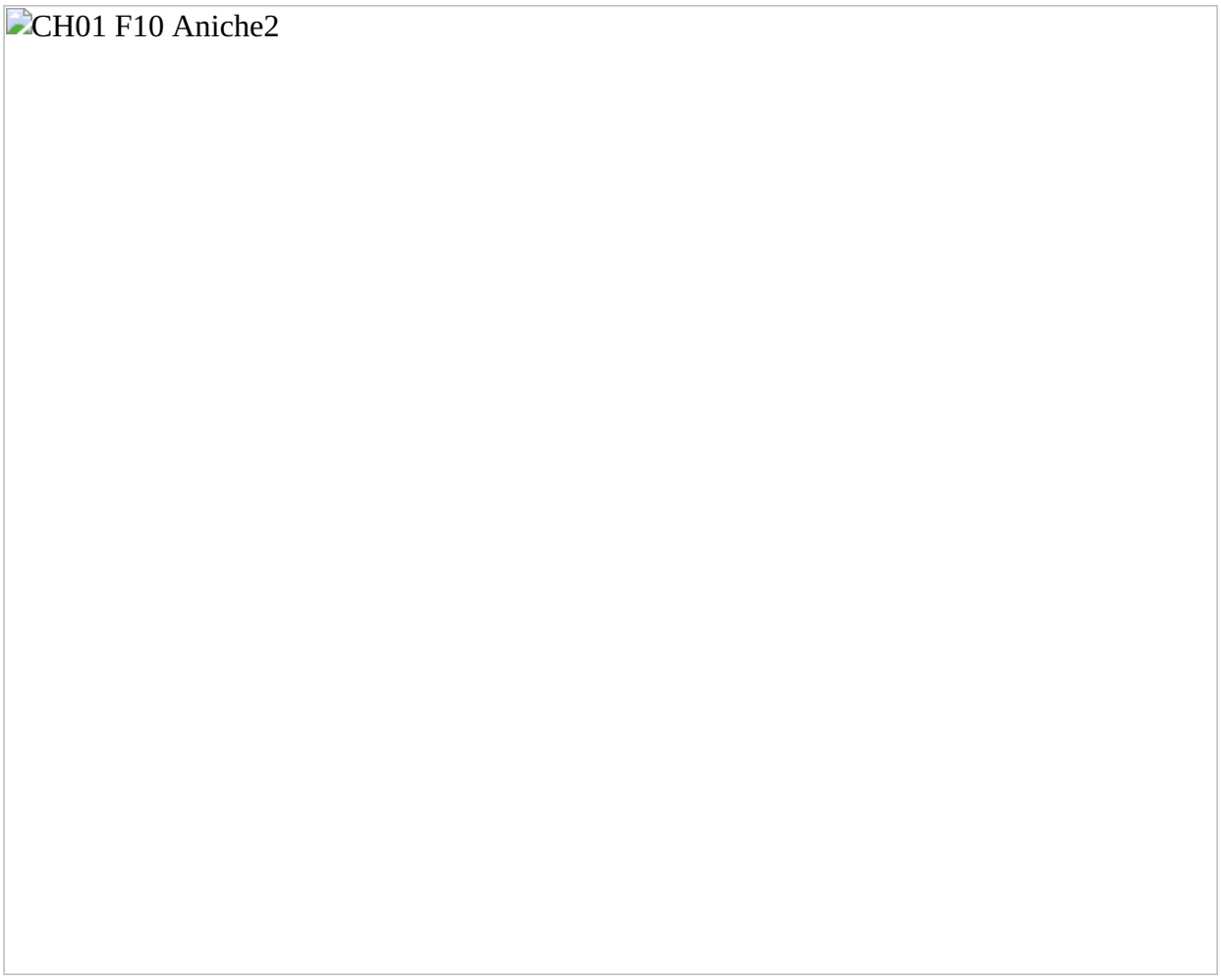
1.4 The example project: PeopleGrow!

To illustrate the design patterns throughout the book, I'll use an imaginary back-end system called PeopleGrow!, illustrated in figure 1.10, an information system that manages employees and their growth through trainings.

The system handles different features. I highlight a few of them below. In bold, domain terms that will often appear in the following chapters:

- The list of **trainings** and **learning paths** that are collections of trainings.
- The **employees** and the trainings that they took or still have to take.
- Trainings are offered multiple times a year. Each **offering** contains the training date and the maximum number of participants allowed.
- Participants can **enroll** for the offerings themselves or be enrolled by the administrator.
- The trainings and the **trainers** that deliver these trainings.
- All sorts of **reports**, such as which trainings miss instructors, which trainings are full, etc.
- All sorts of convenient functionality, such as calendar invites through the company's calendar system, automatic messages via the company's internal chat, and e-mail notifications.
- A front-end available for **administrators** to add new trainings, and learning paths, and see the reports.
- Various APIs available for any internal system to use. For example, employees can enroll in trainings via the company's internal wiki, which uses PeopleGrow!'s APIs.

Figure 1.10 The high-level architecture of PeopleGrow!



Architecturally speaking, PeopleGrow! comprises a front-end, a back-end, and a database. As I said, it also can connect to external systems of the company, such as the internal chat and the calendar system.

The back-end is implemented using object-oriented languages like Java, C#, or Python. It makes use of current frameworks for web development and database access. For example, think of Spring Boot and JPA, if you are a Java developer, ASP.Net MVC / Core and Entity Framework if you are a C# developer, or Django if you are a Python developer. Internally, the back-end models the business. Imagine classes such as `Employee`, `Trainer`, `Enrollment`, and `LearningPath` in the codebase.

The team that's building PeopleGrow! is now facing maintenance challenges. Bugs are emerging. Whatever change is requested by the product team takes days to be done. Developers are always afraid of making changes, and an innocent change often impacts areas of the system one wouldn't expect.

Let's dive into the design decisions of PeopleGrow! and improve them!

1.5 Exercises

Discuss the following questions with a colleague:

1.1. In your opinion, what constitutes a simple object-oriented design? What's the difference between your point of view and the point of view presented in this chapter?

1.2. What types of object-oriented design problems have you faced in your life as a developer? What were their consequences? Do they fit in any of the six categories presented in this chapter?

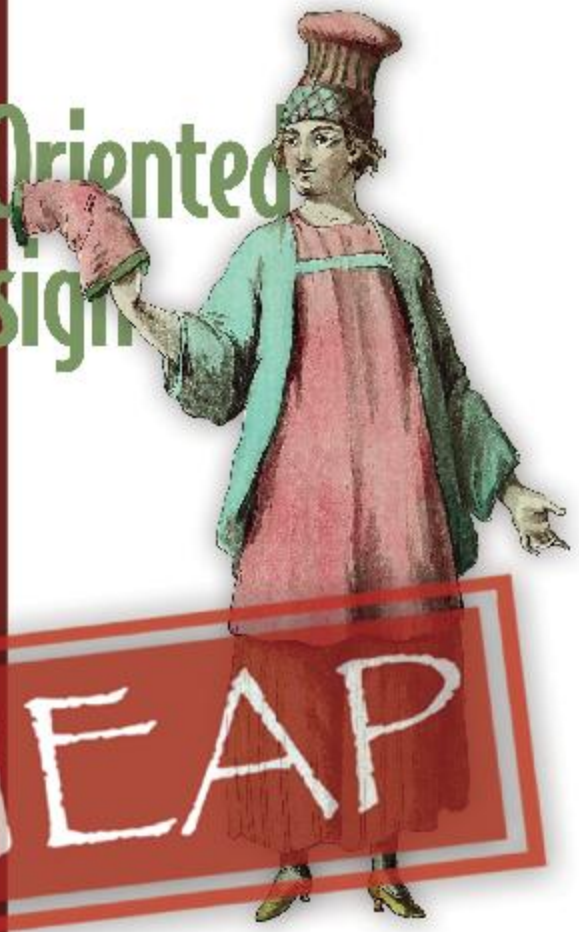
1.3. Do you think it's possible to keep the design always simple as the system evolves? What are the main challenges in keeping it simple?

1.6 Summary

- Building a highly maintainable software system requires a good object-oriented design. Simplicity is the key factor for a highly maintainable software system.
- Building simple object-oriented designs is often simple, but it's hard to keep the design simple as the complexity of the business grows. Managing complexity is essential to maintain and develop software systems effectively.
- Simple and maintainable object-oriented designs have six characteristics: simple code, consistent objects, proper dependency management, good abstractions, infrastructure adequately handled, and well modularized.
- Managing complexity and keeping designs simple is a continuous process that requires daily attention, like brushing your teeth.
- Writing good code is easier when the codebase is already good. Productivity increases, developers feel confident modifying the code, and business value is delivered faster.

A guide to creating clean, maintainable code

SIMPLE Object Oriented Design



Maurício Aniche

MEAP

MANNING