7 Refactoring toward valuable unit tests

This chapter covers

- Recognizing the four types of code
- Understanding the Humble Object pattern
- Writing valuable tests

In chapter 1, I defined the properties of a good unit test suite:

- It is integrated into the development cycle.
- It targets only the most important parts of your code base.
- It provides maximum value with minimum maintenance costs. To achieve this last attribute, you need to be able to:
  - Recognize a valuable test (and, by extension, a test of low value).
  - Write a valuable test.

Chapter 4 covered the topic of recognizing a valuable test using the four attributes: protection against regressions, resistance to refactoring, fast feedback, and maintainability. And chapter 5 expanded on the most important one of the four: resistance to refactoring.

As I mentioned earlier, it's not enough to *recognize* valuable tests, you should also be able to *write* such tests. The latter skill requires the former, but it also requires

that you know code design techniques. Unit tests and the underlying code are highly intertwined, and it's impossible to create valuable tests without putting effort into the code base they cover.

You saw an example of a code base transformation in chapter 6, where we refactored an audit system toward a functional architecture and, as a result, were able to apply output-based testing. This chapter generalizes this approach onto a wider spectrum of applications, including those that can't use a functional architecture. You'll see practical guidelines on how to write valuable tests in almost any software project.

7.1 Identifying the code to refactor

It's rarely possible to significantly improve a test suite without refactoring the underlying code. There's no way around it—test and production code are intrinsically connected. In this section, you'll see how to categorize your code into the four types in order to outline the direction of the refactoring. The subsequent sections show a comprehensive example.

7.1.1 The four types of code

In this section, I describe the four types of code that serve as a foundation for the rest of this chapter.

All production code can be categorized along two dimensions:

- Complexity or domain significance
- The number of collaborators

Code complexity is defined by the number of decision-making (branching) points in the code. The greater that number, the higher the complexity.

How to calculate cyclomatic complexity

In computer science, there's a special term that describes code complexity: *cyclomatic complexity*. Cyclomatic complexity indicates the number of branches in a given program or method. This metric is calculated as

```
1 + <number of branching points>
```

Thus, a method with no control flow statements (such as `if` statements or conditional loops) has a cyclomatic complexity of 1 + 0 = 1.

There's another meaning to this metric. You can think of it in terms of the number of independent paths through the method from an entry to an exit, or the number of tests needed to get a 100% branch coverage.

Note that the number of branching points is counted as the number of simplest predicates involved. For instance, a statement like `IF condition1 AND condition2 THEN ...` is equivalent to `IF condition1 THEN IF condition2 THEN ...` Therefore, its complexity would be `1 + 2 = 3`.

Domain significance shows how significant the code is for the problem domain of your project. Normally, all code in the domain layer has a direct connection to the end users' goals and thus exhibits a high domain significance. On the other hand, utility code doesn't have such a connection.

Complex code and code that has domain significance benefit from unit testing the most because the corresponding tests have great protection against regressions. Note that the domain code doesn't have to be complex, and complex code doesn't have to exhibit domain significance to be test-worthy. The two components are independent of each other. For example, a method calculating an order price can contain no conditional statements and thus have the cyclomatic complexity of `1`. Still, it's important to test such a method because it represents business-critical functionality.

The second dimension is the number of collaborators a class or a method has. As you may remember from chapter 2, a *collaborator* is a dependency that is either mutable or out-of-process (or both). Code with a large number of collaborators is expensive to test. That's due to the maintainability metric, which depends on the size of the test. It takes space to bring collaborators to an expected condition and then check their state or interactions with them afterward. And the more collaborators there are, the larger the test becomes.

The type of the collaborators also matters. Out-of-process collaborators are a no-go when it comes to the domain model. They add additional maintenance costs due to the necessity to maintain complicated mock machinery in tests. You also have to be extra prudent and only use mocks to verify interactions that cross the application boundary in order to maintain proper resistance to refactoring (refer to chapter 5 for more details). It's better to delegate all communications with out-of-process dependencies to classes outside the domain layer. The domain classes then will only work with in-process dependencies.

Notice that both implicit and explicit collaborators count toward this number. It doesn't matter if the system under test (SUT) accepts a collaborator as an argument or refers to it implicitly via a static method, you still have to set up this collaborator in tests. Conversely, immutable dependencies (values or value objects) don't count. Such dependencies are much easier to set up and assert against.

The combination of code complexity, its domain significance, and the number of collaborators give us the four types of code shown in figure 7.1:

- *Domain model and algorithms (figure 7.1, top left)*—Complex code is often part of the domain model but not in 100% of all cases. You might have a complex algorithm that's not directly related to the problem domain.
- *Trivial code (figure 7.1, bottom left)*—Examples of such code in C# are parameterless constructors and one-line properties: they have few (if any) collaborators and exhibit little complexity or domain significance.
- *Controllers (figure 7.1, bottom right)*—This code doesn't do complex or business-critical work by itself but coordinates the work of other components like domain classes and external applications.

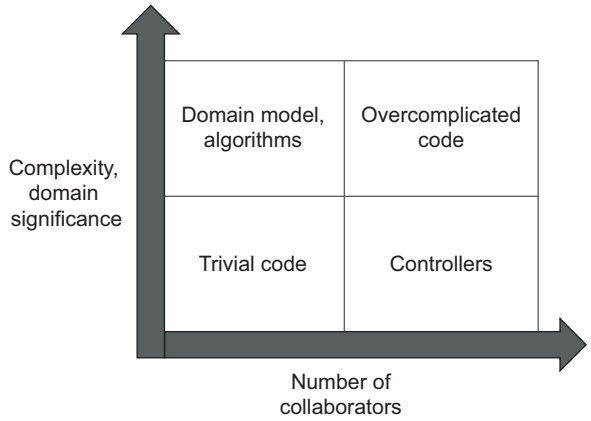

Figure 7.1 The four types of code, categorized by code complexity and domain significance (the vertical axis) and the number of collaborators (the horizontal axis).

- *Overcomplicated code (figure 7.1, top right)*—Such code scores highly on both metrics: it has a lot of collaborators, and it's also complex or important. An example here are *fat controllers* (controllers that don't delegate complex work anywhere and do everything themselves).

Unit testing the top-left quadrant (domain model and algorithms) gives you the best return for your efforts. The resulting unit tests are highly valuable and cheap. They're valuable because the underlying code carries out complex or important logic, thus increasing tests' protection against regressions. And they're cheap because the code has few collaborators (ideally, none), thus decreasing tests' maintenance costs.

Trivial code shouldn't be tested at all; such tests have a close-to-zero value. As for controllers, you should test them briefly as part of a much smaller set of the overarching integration tests (I cover this topic in part 3).

The most problematic type of code is the overcomplicated quadrant. It's hard to unit test but too risky to leave without test coverage. Such code is one of the main reasons many people struggle with unit testing. This whole chapter is primarily devoted to how you can bypass this dilemma. The general idea is to split overcomplicated code into two parts: algorithms and controllers (figure 7.2), although the actual implementation can be tricky at times.

TIP The more important or complex the code, the fewer collaborators it should have.

Getting rid of the overcomplicated code and unit testing only the domain model and algorithms is the path to a highly valuable, easily maintainable test suite. With this approach, you won't have 100% test coverage, but you don't need to—100% coverage shouldn't ever be your goal. Your goal is a test suite where each test adds significant value to the project. Refactor or get rid of all other tests. Don't allow them to inflate the size of your test suite.

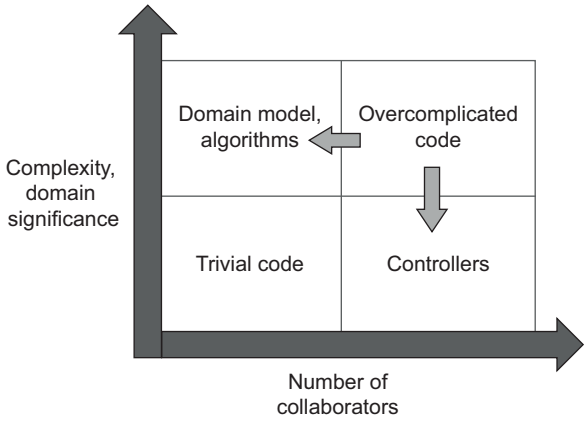

Figure 7.2 Refactor overcomplicated code by splitting it into algorithms and controllers. Ideally, you should have no code in the top-right quadrant.

NOTE Remember that it's better to not write a test at all than to write a bad test.

Of course, getting rid of overcomplicated code is easier said than done. Still, there are techniques that can help you do that. I'll first explain the theory behind those techniques and then demonstrate them using a close-to-real-world example.

7.1.2 Using the Humble Object pattern to split overcomplicated code

To split overcomplicated code, you need to use the Humble Object design pattern. This pattern was introduced by Gerard Meszaros in his book *xUnit Test Patterns: Refactoring Test Code* (Addison-Wesley, 2007) as one of the ways to battle code coupling, but it has a much broader application. You'll see why shortly.

We often find that code is hard to test because it's coupled to a framework dependency (see figure 7.3). Examples include asynchronous or multi-threaded execution, user interfaces, communication with out-of-process dependencies, and so on.

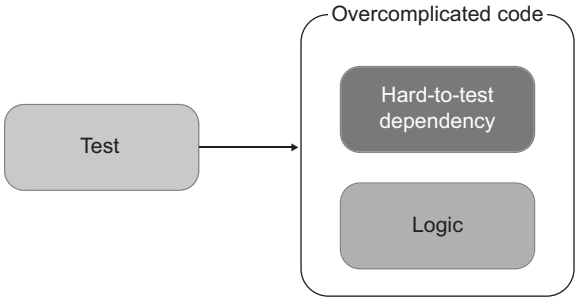

Figure 7.3 It's hard to test code that couples to a difficult dependency. Tests have to deal with that dependency, too, which increases their maintenance cost.

To bring the logic of this code under test, you need to extract a testable part out of it. As a result, the code becomes a thin, *humble* wrapper around that testable part: it glues

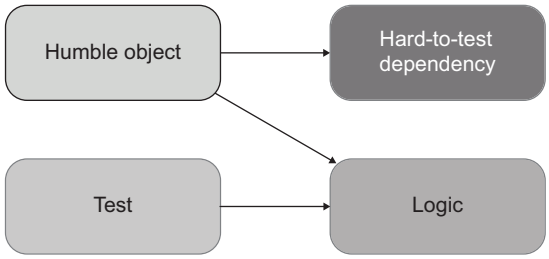

Figure 7.4 The Humble Object pattern extracts the logic out of the overcomplicated code, making that code so humble that it doesn't need to be tested. The extracted logic is moved into another class, decoupled from the hard-to-test dependency.

the hard-to-test dependency and the newly extracted component together, but itself contains little or no logic and thus doesn't need to be tested (figure 7.4).

If this approach looks familiar, it's because you already saw it in this book. In fact, both hexagonal and functional architectures implement this exact pattern. As you may remember from previous chapters, hexagonal architecture advocates for the separation of business logic and communications with out-of-process dependencies. This is what the domain and application services layers are responsible for, respectively.

Functional architecture goes even further and separates business logic from communications with *all* collaborators, not just out-of-process ones. This is what makes functional architecture so testable: its functional core has no collaborators. All dependencies in a functional core are immutable, which brings it very close to the vertical axis on the types-of-code diagram (figure 7.5).

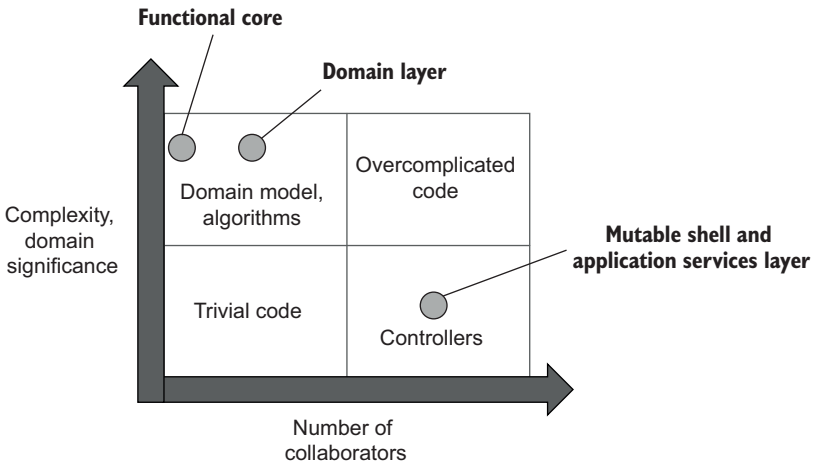

Figure 7.5 The functional core in a functional architecture and the domain layer in a hexagonal architecture reside in the top-left quadrant: they have few collaborators and exhibit high complexity and domain significance. The functional core is closer to the vertical axis because it has no collaborators. The mutable shell (functional architecture) and the application services layer (hexagonal architecture) belong to the controllers' quadrant.

Another way to view the Humble Object pattern is as a means to adhere to the Single Responsibility principle, which states that each class should have only a single responsibility.[1] One such responsibility is always business logic; the pattern can be applied to segregate that logic from pretty much anything.

In our particular situation, we are interested in the separation of business logic and orchestration. You can think of these two responsibilities in terms of *code depth* versus *code width*. Your code can be either deep (complex or important) or wide (work with many collaborators), but never both (figure 7.6).

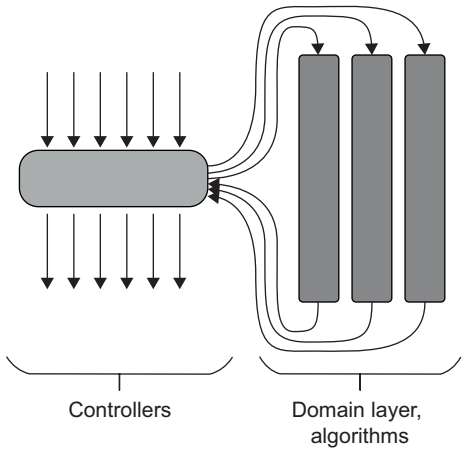

Figure 7.6 Code depth versus code width is a useful metaphor to apply when you think of the separation between the business logic and orchestration responsibilities. Controllers orchestrate many dependencies (represented as arrows in the figure) but aren't complex on their own (complexity is represented as block height). Domain classes are the opposite of that.

I can't stress enough how important this separation is. In fact, many well-known principles and patterns can be described as a form of the Humble Object pattern: they are designed specifically to segregate complex code from the code that does orchestration.

You already saw the relationship between this pattern and hexagonal and functional architectures. Other examples include the Model-View-Presenter (MVP) and the Model-View-Controller (MVC) patterns. These two patterns help you decouple business logic (the *Model* part), UI concerns (the *View*), and the coordination between them (*Presenter* or *Controller*). The Presenter and Controller components are *humble objects*: they glue the view and the model together.

Another example is the Aggregate pattern from *Domain-Driven Design*.[2] One of its goals is to reduce connectivity between classes by grouping them into clusters—*aggregates*. The classes are highly connected inside those clusters, but the clusters themselves are loosely coupled. Such a structure decreases the total number of communications in the code base. The reduced connectivity, in turn, improves testability.

---

[1] See *Agile Principles, Patterns, and Practices in C#* by Robert C. Martin and Micah Martin (Prentice Hall, 2006).
[2] See *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans (Addison-Wesley, 2003).

Note that improved testability is not the only reason to maintain the separation between business logic and orchestration. Such a separation also helps tackle code complexity, which is crucial for project growth, too, especially in the long run. I personally always find it fascinating how a testable design is not only testable but also easy to maintain.

7.2 Refactoring toward valuable unit tests

In this section, I'll show a comprehensive example of splitting overcomplicated code into algorithms and controllers. You saw a similar example in the previous chapter, where we talked about output-based testing and functional architecture. This time, I'll generalize this approach to all enterprise-level applications, with the help of the Humble Object pattern. I'll use this project not only in this chapter but also in the subsequent chapters of part 3.

7.2.1 Introducing a customer management system

The sample project is a customer management system (CRM) that handles user registrations. All users are stored in a database. The system currently supports only one use case: changing a user's email. There are three business rules involved in this operation:

- If the user's email belongs to the company's domain, that user is marked as an employee. Otherwise, they are treated as a customer.
- The system must track the number of employees in the company. If the user's type changes from employee to customer, or vice versa, this number must change, too.
- When the email changes, the system must notify external systems by sending a message to a message bus.

The following listing shows the initial implementation of the CRM system.

Listing 7.1 Initial implementation of the CRM system

```
public class User
{
    public int UserId { get; private set; }
    public string Email { get; private set; }
    public UserType Type { get; private set; }

    public void ChangeEmail(int userId, string newEmail)
    {
        object[] data = Database.GetUserById(userId);
        UserId = userId;
        Email = (string)data[1];
        Type = (UserType)data[2];

        if (Email == newEmail)
            return;
```

Retrieves the user's current email and type from the database

```
        object[] companyData = Database.GetCompany();
        string companyDomainName = (string)companyData[0];
        int numberOfEmployees = (int)companyData[1];

        string emailDomain = newEmail.Split('@')[1];
        bool isEmailCorporate = emailDomain == companyDomainName;
        UserType newType = isEmailCorporate
            ? UserType.Employee
            : UserType.Customer;

        if (Type != newType)
        {
            int delta = newType == UserType.Employee ? 1 : -1;
            int newNumber = numberOfEmployees + delta;
            Database.SaveCompany(newNumber);
        }

        Email = newEmail;
        Type = newType;

        Database.SaveUser(this);
        MessageBus.SendEmailChangedMessage(UserId, newEmail);
    }
}

public enum UserType
{
    Customer = 1,
    Employee = 2
}
```

Retrieves the organization's domain name and the number of employees from the database

Sets the user type depending on the new email's domain name

Updates the number of employees in the organization, if needed

Persists the user in the database

Sends a notification to the message bus

The `User` class changes a user email. Note that, for brevity, I omitted simple validations such as checks for email correctness and user existence in the database. Let's analyze this implementation from the perspective of the types-of-code diagram.

The code's complexity is not too high. The `ChangeEmail` method contains only a couple of explicit decision-making points: whether to identify the user as an employee or a customer, and how to update the company's number of employees. Despite being simple, these decisions are important: they are the application's core business logic. Hence, the class scores highly on the complexity and domain significance dimension.

On the other hand, the `User` class has four dependencies, two of which are explicit and the other two of which are implicit. The explicit dependencies are the `userId` and `newEmail` arguments. These are values, though, and thus don't count toward the class's number of collaborators. The implicit ones are `Database` and `MessageBus`. These two are out-of-process collaborators. As I mentioned earlier, out-of-process collaborators are a no-go for code with high domain significance. Hence, the `User` class scores highly on the collaborators dimension, which puts this class into the overcomplicated category (figure 7.7).

This approach—when a domain class retrieves and persists itself to the database—is called the Active Record pattern. It works fine in simple or short-lived projects but

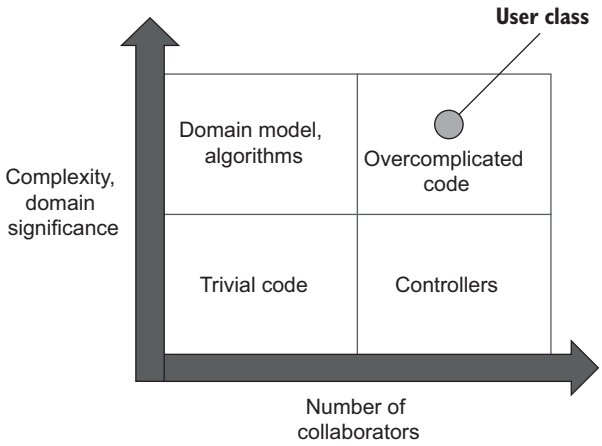

Figure 7.7 The initial implementation of the `User` class scores highly on both dimensions and thus falls into the category of overcomplicated code.

often fails to scale as the code base grows. The reason is precisely this lack of separation between these two responsibilities: business logic and communication with out-of-process dependencies.

7.2.2 Take 1: Making implicit dependencies explicit

The usual approach to improve testability is to make implicit dependencies explicit: that is, introduce interfaces for `Database` and `MessageBus`, inject those interfaces into `User`, and then mock them in tests. This approach does help, and that's exactly what we did in the previous chapter when we introduced the implementation with mocks for the audit system. However, it's not enough.

From the perspective of the types-of-code diagram, it doesn't matter if the domain model refers to out-of-process dependencies directly or via an interface. Such dependencies are still *out-of-process*; they are proxies to data that is not yet in memory. You still need to maintain complicated mock machinery in order to test such classes, which increases the tests' maintenance costs. Moreover, using mocks for the database dependency would lead to test fragility (we'll discuss this in the next chapter).

Overall, it's much cleaner for the domain model not to depend on out-of-process collaborators at all, directly or indirectly (via an interface). That's what the hexagonal architecture advocates as well—the domain model shouldn't be responsible for communications with external systems.

7.2.3 Take 2: Introducing an application services layer

To overcome the problem of the domain model directly communicating with external systems, we need to shift this responsibility to another class, a *humble* controller (an application service, in the hexagonal architecture taxonomy). As a general rule, domain classes should only depend on in-process dependencies, such as other domain classes, or plain values. Here's what the first version of that application service looks like.

Listing 7.2 Application service, version 1

```
public class UserController
{
    private readonly Database _database = new Database();
    private readonly MessageBus _messageBus = new MessageBus();

    public void ChangeEmail(int userId, string newEmail)
    {
        object[] data = _database.GetUserById(userId);
        string email = (string)data[1];
        UserType type = (UserType)data[2];
        var user = new User(userId, email, type);

        object[] companyData = _database.GetCompany();
        string companyDomainName = (string)companyData[0];
        int numberOfEmployees = (int)companyData[1];

        int newNumberOfEmployees = user.ChangeEmail(
            newEmail, companyDomainName, numberOfEmployees);

        _database.SaveCompany(newNumberOfEmployees);
        _database.SaveUser(user);
        _messageBus.SendEmailChangedMessage(userId, newEmail);
    }
}
```

This is a good first try; the application service helped offload the work with out-of-process dependencies from the `User` class. But there are some issues with this implementation:

- The out-of-process dependencies (`Database` and `MessageBus`) are instantiated directly, not injected. That's going to be a problem for the integration tests we'll be writing for this class.
- The controller reconstructs a `User` instance from the raw data it receives from the database. This is complex logic and thus shouldn't belong to the application service, whose sole role is orchestration, not logic of any complexity or domain significance.
- The same is true for the company's data. The other problem with that data is that `User` now returns an updated number of employees, which doesn't look right. The number of company employees has nothing to do with a specific user. This responsibility should belong elsewhere.
- The controller persists modified data and sends notifications to the message bus unconditionally, regardless of whether the new email is different than the previous one.

The `User` class has become quite easy to test because it no longer has to communicate with out-of-process dependencies. In fact, it has no collaborators whatsoever—out-of-process or not. Here's the new version of `User`'s `ChangeEmail` method:

```
public int ChangeEmail(string newEmail,
    string companyDomainName, int numberOfEmployees)
{
    if (Email == newEmail)
        return numberOfEmployees;

    string emailDomain = newEmail.Split('@')[1];
    bool isEmailCorporate = emailDomain == companyDomainName;
    UserType newType = isEmailCorporate
        ? UserType.Employee
        : UserType.Customer;

    if (Type != newType)
    {
        int delta = newType == UserType.Employee ? 1 : -1;
        int newNumber = numberOfEmployees + delta;
        numberOfEmployees = newNumber;
    }

    Email = newEmail;
    Type = newType;

    return numberOfEmployees;
}
```

Figure 7.8 shows where `User` and `UserController` currently stand in our diagram. `User` has moved to the domain model quadrant, close to the vertical axis, because it no longer has to deal with collaborators. `UserController` is more problematic. Although I've put it into the controllers quadrant, it almost crosses the boundary into overcomplicated code because it contains logic that is quite complex.

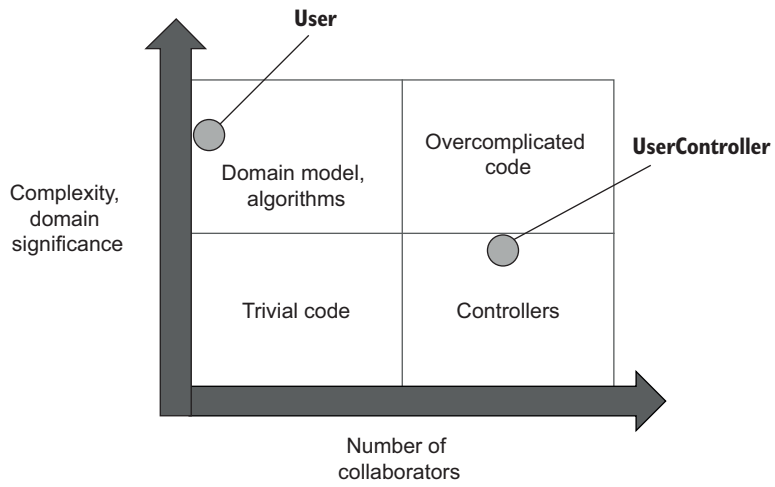

Figure 7.8 Take 2 puts `User` in the domain model quadrant, close to the vertical axis. `UserController` almost crosses the boundary with the overcomplicated quadrant because it contains complex logic.

7.2.4 Take 3: Removing complexity from the application service

To put `UserController` firmly into the controllers quadrant, we need to extract the reconstruction logic from it. If you use an object-relational mapping (ORM) library to map the database into the domain model, that would be a good place to which to attribute the reconstruction logic. Each ORM library has a dedicated place where you can specify how your database tables should be mapped to domain classes, such as attributes on top of those domain classes, XML files, or files with fluent mappings.

If you don't want to or can't use an ORM, create a factory in the domain model that will instantiate the domain classes using raw database data. This factory can be a separate class or, for simpler cases, a static method in the existing domain classes. The reconstruction logic in our sample application is not too complicated, but it's good to keep such things separated, so I'm putting it in a separate `UserFactory` class as shown in the following listing.

Listing 7.3 User factory

```
public class UserFactory
{
    public static User Create(object[] data)
    {
        Precondition.Requires(data.Length >= 3);

        int id = (int)data[0];
        string email = (string)data[1];
        UserType type = (UserType)data[2];

        return new User(id, email, type);
    }
}
```

This code is now fully isolated from all collaborators and therefore easily testable. Notice that I've put a safeguard in this method: a requirement to have at least three elements in the data array. `Precondition` is a simple custom class that throws an exception if the Boolean argument is false. The reason for this class is the more succinct code and the condition inversion: affirmative statements are more readable than negative ones. In our example, the `data.Length >= 3` requirement reads better than

```
if (data.Length < 3)
    throw new Exception();
```

Note that while this reconstruction logic is somewhat complex, it doesn't have domain significance: it isn't directly related to the client's goal of changing the user email. It's an example of the utility code I refer to in previous chapters.

How is the reconstruction logic complex?

How is the reconstruction logic complex, given that there's only a single branching point in the `UserFactory.Create()` method? As I mentioned in chapter 1, there could be a lot of hidden branching points in the underlying libraries used by the code and thus a lot of potential for something to go wrong. This is exactly the case for the `UserFactory.Create()` method.

Referring to an array element by index (`data[0]`) entails an internal decision made by the .NET Framework as to what data element to access. The same is true for the conversion from `object` to `int` or `string`. Internally, the .NET Framework decides whether to throw a cast exception or allow the conversion to proceed. All these hidden branches make the reconstruction logic test-worthy, despite the lack of decision points in it.

7.2.5 Take 4: Introducing a new Company class

Look at this code in the controller once again:

```
object[] companyData = _database.GetCompany();
string companyDomainName = (string)companyData[0];
int numberOfEmployees = (int)companyData[1];

int newNumberOfEmployees = user.ChangeEmail(
    newEmail, companyDomainName, numberOfEmployees);
```

The awkwardness of returning an updated number of employees from `User` is a sign of a misplaced responsibility, which itself is a sign of a missing abstraction. To fix this, we need to introduce another domain class, `Company`, that bundles the company-related logic and data together, as shown in the following listing.

Listing 7.4 The new class in the domain layer

```
public class Company
{
    public string DomainName { get; private set; }
    public int NumberOfEmployees { get; private set; }

    public void ChangeNumberOfEmployees(int delta)
    {
        Precondition.Requires(NumberOfEmployees + delta >= 0);

        NumberOfEmployees += delta;
    }

    public bool IsEmailCorporate(string email)
    {
        string emailDomain = email.Split('@')[1];
        return emailDomain == DomainName;
    }
}
```

There are two methods in this class: `ChangeNumberOfEmployees()` and `IsEmailCorporate()`. These methods help adhere to the tell-don't-ask principle I mentioned in chapter 5. This principle advocates for bundling together data and operations on that data. A `User` instance will *tell* the company to change its number of employees or figure out whether a particular email is corporate; it won't *ask* for the raw data and do everything on its own.

There's also a new `CompanyFactory` class, which is responsible for the reconstruction of `Company` objects, similar to `UserFactory`. This is how the controller now looks.

Listing 7.5 Controller after refactoring

```
public class UserController
{
    private readonly Database _database = new Database();
    private readonly MessageBus _messageBus = new MessageBus();

    public void ChangeEmail(int userId, string newEmail)
    {
        object[] userData = _database.GetUserById(userId);
        User user = UserFactory.Create(userData);

        object[] companyData = _database.GetCompany();
        Company company = CompanyFactory.Create(companyData);

        user.ChangeEmail(newEmail, company);

        _database.SaveCompany(company);
        _database.SaveUser(user);
        _messageBus.SendEmailChangedMessage(userId, newEmail);
    }
}
```

And here's the `User` class.

Listing 7.6 `User` after refactoring

```
public class User
{
    public int UserId { get; private set; }
    public string Email { get; private set; }
    public UserType Type { get; private set; }

    public void ChangeEmail(string newEmail, Company company)
    {
        if (Email == newEmail)
            return;

        UserType newType = company.IsEmailCorporate(newEmail)
            ? UserType.Employee
            : UserType.Customer;
```

```
        if (Type != newType)
        {
            int delta = newType == UserType.Employee ? 1 : -1;
            company.ChangeNumberOfEmployees(delta);
        }

        Email = newEmail;
        Type = newType;
    }
}
```

Notice how the removal of the misplaced responsibility made `User` much cleaner. Instead of operating on company data, it accepts a `Company` instance and delegates two important pieces of work to that instance: determining whether an email is corporate and changing the number of employees in the company.

Figure 7.9 shows where each class stands in the diagram. The factories and both domain classes reside in the domain model and algorithms quadrant. `User` has moved to the right because it now has one collaborator, `Company`, whereas previously it had none. That has made `User` less testable, but not much.

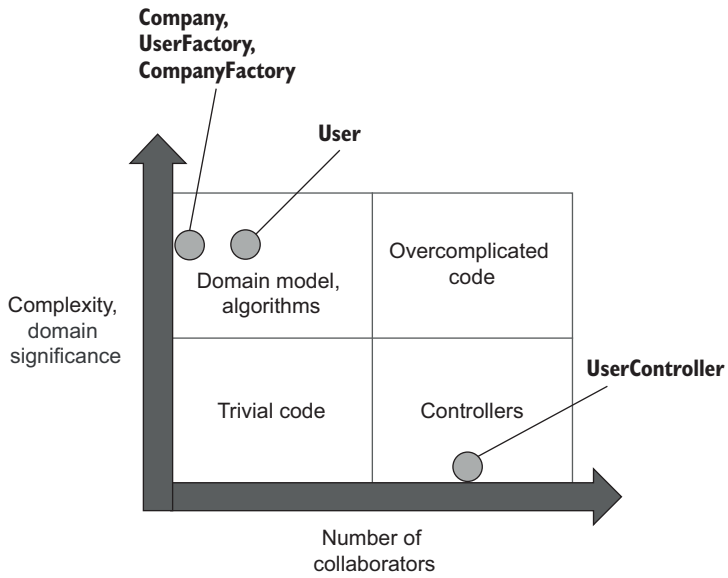

Figure 7.9 `User` has shifted to the right because it now has the `Company` collaborator. `UserController` firmly stands in the controllers quadrant; all its complexity has moved to the factories.

`UserController` now firmly stands in the controllers quadrant because all of its complexity has moved to the factories. The only thing this class is responsible for is gluing together all the collaborating parties.

Note the similarities between this implementation and the functional architecture from the previous chapter. Neither the functional core in the audit system nor the domain layer in this CRM (the `User` and `Company` classes) communicates with out-of-process dependencies. In both implementations, the application services layer is responsible for such communication: it gets the raw data from the filesystem or from the database, passes that data to stateless algorithms or the domain model, and then persists the results back to the data storage.

The difference between the two implementations is in their treatment of side effects. The functional core doesn't incur any side effects whatsoever. The CRM's domain model does, but all those side effects remain inside the domain model in the form of the changed user email and the number of employees. The side effects only cross the domain model's boundary when the controller persists the `User` and `Company` objects in the database.

The fact that all side effects are contained in memory until the very last moment improves testability a lot. Your tests don't need to examine out-of-process dependencies, nor do they need to resort to communication-based testing. All the verification can be done using output-based and state-based testing of objects in memory.

7.3 Analysis of optimal unit test coverage

Now that we've completed the refactoring with the help of the Humble Object pattern, let's analyze which parts of the project fall into which code category and how those parts should be tested. Table 7.1 shows all the code from the sample project grouped by position in the types-of-code diagram.

Table 7.1 Types of code in the sample project after refactoring using the Humble Object pattern

	Few collaborators	Many collaborators
High complexity or domain significance	`ChangeEmail(newEmail, company)` in `User`; `ChangeNumberOfEmployees(delta)` and `IsEmailCorporate(email)` in `Company`; and `Create(data)` in `UserFactory` and `CompanyFactory`	
Low complexity and domain significance	Constructors in `User` and `Company`	`ChangeEmail(userId, newEmail)` in `UserController`

With the full separation of business logic and orchestration at hand, it's easy to decide which parts of the code base to unit test.

7.3.1 Testing the domain layer and utility code

Testing methods in the top-left quadrant in table 7.1 provides the best results in cost-benefit terms. The code's high complexity or domain significance guarantees great protection against regressions, while having few collaborators ensures the lowest maintenance costs. This is an example of how `User` could be tested:

```
[Fact]
public void Changing_email_from_non_corporate_to_corporate()
{
    var company = new Company("mycorp.com", 1);
    var sut = new User(1, "user@gmail.com", UserType.Customer);

    sut.ChangeEmail("new@mycorp.com", company);

    Assert.Equal(2, company.NumberOfEmployees);
    Assert.Equal("new@mycorp.com", sut.Email);
    Assert.Equal(UserType.Employee, sut.Type);
}
```

To achieve full coverage, you'd need another three such tests:

```
public void Changing_email_from_corporate_to_non_corporate()
public void Changing_email_without_changing_user_type()
public void Changing_email_to_the_same_one()
```

Tests for the other three classes would be even shorter, and you could use parameterized tests to group several test cases together:

```
[InlineData("mycorp.com", "email@mycorp.com", true)]
[InlineData("mycorp.com", "email@gmail.com", false)]
[Theory]
public void Differentiates_a_corporate_email_from_non_corporate(
    string domain, string email, bool expectedResult)
{
    var sut = new Company(domain, 0);

    bool isEmailCorporate = sut.IsEmailCorporate(email);

    Assert.Equal(expectedResult, isEmailCorporate);
}
```

7.3.2 Testing the code from the other three quadrants

Code with low complexity and few collaborators (bottom-left quadrant in table 7.1) is represented by the constructors in `User` and `Company`, such as

```
public User(int userId, string email, UserType type)
{
    UserId = userId;
    Email = email;
    Type = type;
}
```

These constructors are trivial and aren't worth the effort. The resulting tests wouldn't provide great enough protection against regressions.

The refactoring has eliminated all code with high complexity and a large number of collaborators (top-right quadrant in table 7.1), so we have nothing to test there, either. As for the controllers quadrant (bottom-right in table 7.1), we'll discuss testing it in the next chapter.

7.3.3 Should you test preconditions?

Let's take a look at a special kind of branching points—preconditions—and see whether you should test them. For example, look at this method from `Company` once again:

```
public void ChangeNumberOfEmployees(int delta)
{
    Precondition.Requires(NumberOfEmployees + delta >= 0);

    NumberOfEmployees += delta;
}
```

It has a precondition stating that the number of employees in the company should never become negative. This precondition is a safeguard that's activated only in exceptional cases. Such exceptional cases are usually the result of bugs. The only possible reason for the number of employees to go below zero is if there's an error in code. The safeguard provides a mechanism for your software to fail fast and to prevent the error from spreading and being persisted in the database, where it would be much harder to deal with. Should you test such preconditions? In other words, would such tests be valuable enough to have in the test suite?

There's no hard rule here, but the general guideline I recommend is to test all preconditions that have domain significance. The requirement for the non-negative number of employees is such a precondition. It's part of the `Company` class's invariants: conditions that should be held true at all times. But don't spend time testing preconditions that don't have domain significance. For example, `UserFactory` has the following safeguard in its `Create` method:

```
public static User Create(object[] data)
{
    Precondition.Requires(data.Length >= 3);

    /* Extract id, email, and type out of data */
}
```

There's no domain meaning to this precondition and therefore not much value in testing it.

7.4 Handling conditional logic in controllers

Handling conditional logic and simultaneously maintaining the domain layer free of out-of-process collaborators is often tricky and involves trade-offs. In this section, I'll show what those trade-offs are and how to decide which of them to choose in your own project.

The separation between business logic and orchestration works best when a business operation has three distinct stages:

- Retrieving data from storage
- Executing business logic
- Persisting data back to the storage (figure 7.10)

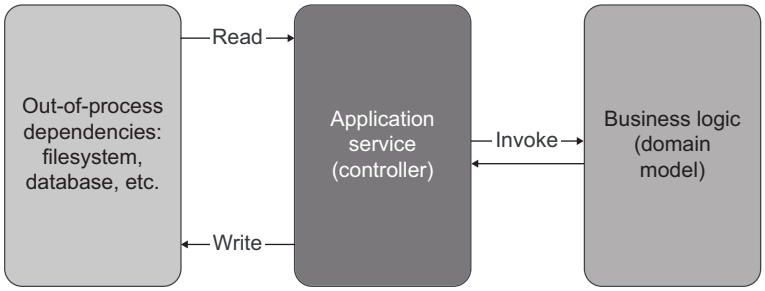

Figure 7.10 Hexagonal and functional architectures work best when all references to out-of-process dependencies can be pushed to the edges of business operations.

There are a lot of situations where these stages aren't as clearcut, though. As we discussed in chapter 6, you might need to query additional data from an out-of-process dependency based on an intermediate result of the decision-making process (figure 7.11). Writing to the out-of-process dependency often depends on that result, too.

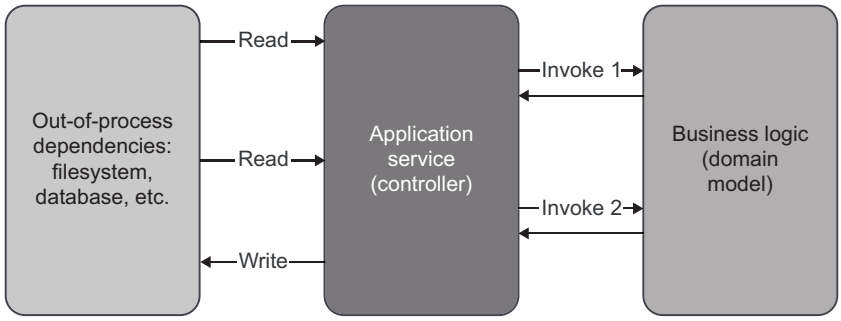

Figure 7.11 A hexagonal architecture doesn't work as well when you need to refer to out-of-process dependencies in the middle of the business operation.

As also discussed in the previous chapter, you have three options in such a situation:

- *Push all external reads and writes to the edges anyway.* This approach preserves the read-decide-act structure but concedes performance: the controller will call out-of-process dependencies even when there's no need for that.
- *Inject the out-of-process dependencies into the domain model* and allow the business logic to directly decide when to call those dependencies.
- *Split the decision-making process into more granular steps* and have the controller act on each of those steps separately.

The challenge is to balance the following three attributes:

- *Domain model testability*, which is a function of the number and type of collaborators in domain classes
- *Controller simplicity*, which depends on the presence of decision-making (branching) points in the controller
- *Performance*, as defined by the number of calls to out-of-process dependencies

Each option only gives you two out of the three attributes (figure 7.12):

- *Pushing all external reads and writes to the edges of a business operation*—Preserves controller simplicity and keeps the domain model isolated from out-of-process dependencies (thus allowing it to remain testable) but concedes performance.
- *Injecting out-of-process dependencies into the domain model*—Keeps performance and the controller's simplicity intact but damages domain model testability.
- *Splitting the decision-making process into more granular steps*—Helps with both performance and domain model testability but concedes controller simplicity. You'll need to introduce decision-making points in the controller in order to manage these granular steps.

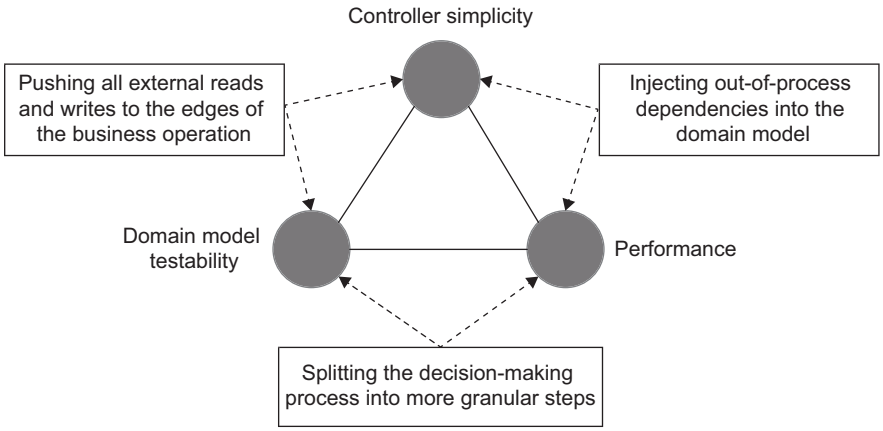

Figure 7.12 There's no single solution that satisfies all three attributes: controller simplicity, domain model testability, and performance. You have to choose two out of the three.

In most software projects, performance *is* important, so the first approach (pushing external reads and writes to the edges of a business operation) is out of the question. The second option (injecting out-of-process dependencies into the domain model) brings most of your code into the overcomplicated quadrant on the types-of-code diagram. This is exactly what we refactored the initial CRM implementation away from. I recommend that you avoid this approach: such code no longer preserves the separation

between business logic and communication with out-of-process dependencies and thus becomes much harder to test and maintain.

That leaves you with the third option: splitting the decision-making process into smaller steps. With this approach, you will have to make your controllers more complex, which will also push them closer to the overcomplicated quadrant. But there are ways to mitigate this problem. Although you will rarely be able to factor *all* the complexity out of controllers as we did previously in the sample project, you *can* keep that complexity manageable.

7.4.1 Using the CanExecute/Execute pattern

The first way to mitigate the growth of the controllers' complexity is to use the CanExecute/Execute pattern, which helps avoid leaking of business logic from the domain model to controllers. This pattern is best explained with an example, so let's expand on our sample project.

Let's say that a user can change their email only until they confirm it. If a user tries to change the email after the confirmation, they should be shown an error message. To accommodate this new requirement, we'll add a new property to the `User` class.

Listing 7.7 `User` with a new property

```
public class User
{
    public int UserId { get; private set; }
    public string Email { get; private set; }
    public UserType Type { get; private set; }
    public bool IsEmailConfirmed
        { get; private set; }                    // New property

    /* ChangeEmail(newEmail, company) method */
}
```

There are two options for where to put this check. First, you could put it in `User`'s `ChangeEmail` method:

```
public string ChangeEmail(string newEmail, Company company)
{
    if (IsEmailConfirmed)
        return "Can't change a confirmed email";

    /* the rest of the method */
}
```

Then you could make the controller either return an error or incur all necessary side effects, depending on this method's output.

Listing 7.8 The controller, still stripped of all decision-making

```
public string ChangeEmail(int userId, string newEmail)
{
```

```
    object[] userData = _database.GetUserById(userId);
    User user = UserFactory.Create(userData);

    object[] companyData = _database.GetCompany();
    Company company = CompanyFactory.Create(companyData);

    string error = user.ChangeEmail(newEmail, company);
    if (error != null)
        return error;

    _database.SaveCompany(company);
    _database.SaveUser(user);
    _messageBus.SendEmailChangedMessage(userId, newEmail);

    return "OK";
}
```

Prepares the data

Makes a decision

Acts on the decision

This implementation keeps the controller free of decision-making, but it does so at the expense of a performance drawback. The `Company` instance is retrieved from the database unconditionally, even when the email is confirmed and thus can't be changed. This is an example of pushing all external reads and writes to the edges of a business operation.

> **NOTE** I don't consider the new `if` statement analyzing the `error` string an increase in complexity because it belongs to the acting phase; it's not part of the decision-making process. All the decisions are made by the `User` class, and the controller merely acts on those decisions.

The second option is to move the check for `IsEmailConfirmed` from `User` to the controller.

Listing 7.9 Controller deciding whether to change the user's email

```
public string ChangeEmail(int userId, string newEmail)
{
    object[] userData = _database.GetUserById(userId);
    User user = UserFactory.Create(userData);

    if (user.IsEmailConfirmed)
        return "Can't change a confirmed email";

    object[] companyData = _database.GetCompany();
    Company company = CompanyFactory.Create(companyData);

    user.ChangeEmail(newEmail, company);

    _database.SaveCompany(company);
    _database.SaveUser(user);
    _messageBus.SendEmailChangedMessage(userId, newEmail);

    return "OK";
}
```

Decision-making moved here from User.

With this implementation, the performance stays intact: the `Company` instance is retrieved from the database only after it is certain that the email can be changed. But now the decision-making process is split into two parts:

- Whether to proceed with the change of email (performed by the controller)
- What to do during that change (performed by `User`)

Now it's also possible to change the email without verifying the `IsEmailConfirmed` flag first, which diminishes the domain model's encapsulation. Such fragmentation hinders the separation between business logic and orchestration and moves the controller closer to the overcomplicated danger zone.

To prevent this fragmentation, you can introduce a new method in `User`, `CanChangeEmail()`, and make its successful execution a precondition for changing an email. The modified version in the following listing follows the CanExecute/Execute pattern.

Listing 7.10 Changing an email using the CanExecute/Execute pattern

```
public string CanChangeEmail()
{
    if (IsEmailConfirmed)
        return "Can't change a confirmed email";

    return null;
}

public void ChangeEmail(string newEmail, Company company)
{
    Precondition.Requires(CanChangeEmail() == null);

    /* the rest of the method */
}
```

This approach provides two important benefits:

- The controller no longer needs to know anything about the process of changing emails. All it needs to do is call the `CanChangeEmail()` method to see if the operation can be done. Notice that this method can contain multiple validations, all encapsulated away from the controller.
- The additional precondition in `ChangeEmail()` guarantees that the email won't ever be changed without checking for the confirmation first.

This pattern helps you to consolidate all decisions in the domain layer. The controller no longer has an option not to check for the email confirmation, which essentially eliminates the new decision-making point from that controller. Thus, although the controller still contains the `if` statement calling `CanChangeEmail()`, you don't need to test that `if` statement. Unit testing the precondition in the `User` class itself is enough.

NOTE For simplicity's sake, I'm using a `string` to denote an error. In a real-world project, you may want to introduce a custom `Result` class to indicate the success or failure of an operation.

7.4.2 Using domain events to track changes in the domain model

It's sometimes hard to deduct what steps led the domain model to the current state. Still, it might be important to know these steps because you need to inform external systems about what exactly has happened in your application. Putting this responsibility on the controllers would make them more complicated. To avoid that, you can track important changes in the domain model and then convert those changes into calls to out-of-process dependencies after the business operation is complete. *Domain events* help you implement such tracking.

> **DEFINITION** A *domain event* describes an event in the application that is meaningful to domain experts. The meaningfulness for domain experts is what differentiates domain events from regular events (such as button clicks). Domain events are often used to inform external applications about important changes that have happened in your system.

Our CRM has a tracking requirement, too: it has to notify external systems about changed user emails by sending messages to the message bus. The current implementation has a flaw in the notification functionality: it sends messages even when the email is not changed, as shown in the following listing.

Listing 7.11 Sends a notification even when the email has not changed

```
// User
public void ChangeEmail(string newEmail, Company company)
{
    Precondition.Requires(CanChangeEmail() == null);

    if (Email == newEmail)
        return;

    /* the rest of the method */
}

// Controller
public string ChangeEmail(int userId, string newEmail)
{
    /* preparations */

    user.ChangeEmail(newEmail, company);

    _database.SaveCompany(company);
    _database.SaveUser(user);
    _messageBus.SendEmailChangedMessage(
        userId, newEmail);

    return "OK";
}
```

User email may not change.

The controller sends a message anyway.

You could resolve this bug by moving the check for email sameness to the controller, but then again, there are issues with the business logic fragmentation. And you can't

put this check to `CanChangeEmail()` because the application shouldn't return an error if the new email is the same as the old one.

Note that this particular check probably doesn't introduce too much business logic fragmentation, so I personally wouldn't consider the controller overcomplicated if it contained that check. But you may find yourself in a more difficult situation in which it's hard to prevent your application from making unnecessary calls to out-of-process dependencies without passing those dependencies to the domain model, thus overcomplicating that domain model. The only way to prevent such overcomplication is the use of domain events.

From an implementation standpoint, a *domain event* is a class that contains data needed to notify external systems. In our specific example, it is the user's ID and email:

```
public class EmailChangedEvent
{
    public int UserId { get; }
    public string NewEmail { get; }
}
```

NOTE Domain events should always be named in the past tense because they represent things that already happened. Domain events are values—they are immutable and interchangeable.

`User` will have a collection of such events to which it will add a new element when the email changes. This is how its `ChangeEmail()` method looks after the refactoring.

Listing 7.12 `User` adding an event when the email changes

```
public void ChangeEmail(string newEmail, Company company)
{
    Precondition.Requires(CanChangeEmail() == null);

    if (Email == newEmail)
        return;

    UserType newType = company.IsEmailCorporate(newEmail)
        ? UserType.Employee
        : UserType.Customer;

    if (Type != newType)
    {
        int delta = newType == UserType.Employee ? 1 : -1;
        company.ChangeNumberOfEmployees(delta);
    }

    Email = newEmail;
    Type = newType;
    EmailChangedEvents.Add(                                  A new event indicates
        new EmailChangedEvent(UserId, newEmail));            the change of email.
}
```

The controller then will convert the events into messages on the bus.

Listing 7.13 The controller processing domain events

```
public string ChangeEmail(int userId, string newEmail)
{
    object[] userData = _database.GetUserById(userId);
    User user = UserFactory.Create(userData);

    string error = user.CanChangeEmail();
    if (error != null)
        return error;

    object[] companyData = _database.GetCompany();
    Company company = CompanyFactory.Create(companyData);

    user.ChangeEmail(newEmail, company);

    _database.SaveCompany(company);
    _database.SaveUser(user);
    foreach (var ev in user.EmailChangedEvents)
    {
        _messageBus.SendEmailChangedMessage(
            ev.UserId, ev.NewEmail);
    }

    return "OK";
}
```

Domain event processing

Notice that the `Company` and `User` instances are still persisted in the database unconditionally: the persistence logic doesn't depend on domain events. This is due to the difference between changes in the database and messages in the bus.

Assuming that no application has access to the database other than the CRM, communications with that database are not part of the CRM's observable behavior—they are implementation details. As long as the final state of the database is correct, it doesn't matter how many calls your application makes to that database. On the other hand, communications with the message bus *are* part of the application's observable behavior. In order to maintain the contract with external systems, the CRM should put messages on the bus only when the email changes.

There are performance implications to persisting data in the database unconditionally, but they are relatively insignificant. The chances that after all the validations the new email is the same as the old one are quite small. The use of an ORM can also help. Most ORMs won't make a round trip to the database if there are no changes to the object state.

You can generalize the solution with domain events: extract a `DomainEvent` base class and introduce a base class for all domain classes, which would contain a collection of such events: `List<DomainEvent> events`. You can also write a separate event dispatcher instead of dispatching domain events manually in controllers. Finally, in larger projects, you might need a mechanism for merging domain events before

dispatching them. That topic is outside the scope of this book, though. You can read about it in my article "Merging domain events before dispatching" at http://mng.bz/YeVe.

Domain events remove the decision-making responsibility from the controller and put that responsibility into the domain model, thus simplifying unit testing communications with external systems. Instead of verifying the controller itself and using mocks to substitute out-of-process dependencies, you can test the domain event creation directly in unit tests, as shown next.

Listing 7.14 Testing the creation of a domain event

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    var company = new Company("mycorp.com", 1);
    var sut = new User(1, "user@mycorp.com", UserType.Employee, false);

    sut.ChangeEmail("new@gmail.com", company);

    company.NumberOfEmployees.Should().Be(0);
    sut.Email.Should().Be("new@gmail.com");
    sut.Type.Should().Be(UserType.Customer);
    sut.EmailChangedEvents.Should().Equal(
        new EmailChangedEvent(1, "new@gmail.com"));
}
```

Simultaneously asserts the collection size and the element in the collection

Of course, you'll still need to test the controller to make sure it does the orchestration correctly, but doing so requires a much smaller set of tests. That's the topic of the next chapter.

7.5 Conclusion

Notice a theme that has been present throughout this chapter: abstracting away the application of side effects to external systems. You achieve such abstraction by keeping those side effects in memory until the very end of the business operation, so that they can be tested with plain unit tests without involving out-of-process dependencies. Domain events are abstractions on top of upcoming messages in the bus. Changes in domain classes are abstractions on top of upcoming modifications in the database.

NOTE It's easier to test abstractions than the things they abstract.

Although we were able to successfully contain all the decision-making in the domain model with the help of domain events and the CanExecute/Execute pattern, you won't be able to always do that. There are situations where business logic fragmentation is inevitable.

For example, there's no way to verify email uniqueness outside the controller without introducing out-of-process dependencies in the domain model. Another example is failures in out-of-process dependencies that should alter the course of the business

operation. The decision about which way to go can't reside in the domain layer because it's not the domain layer that calls those out-of-process dependencies. You will have to put this logic into controllers and then cover it with integration tests. Still, even with the potential fragmentation, there's a lot of value in separating business logic from orchestration because this separation drastically simplifies the unit testing process.

Just as you can't avoid having some business logic in controllers, you will rarely be able to remove all collaborators from domain classes. And that's fine. One, two, or even three collaborators won't turn a domain class into overcomplicated code, as long as these collaborators don't refer to out-of-process dependencies.

Don't use mocks to verify interactions with such collaborators, though. These interactions have nothing to do with the domain model's observable behavior. Only the very first call, which goes from a controller to a domain class, has an immediate connection to that controller's goal. All the subsequent calls the domain class makes to its neighbor domain classes within the same operation are implementation details.

Figure 7.13 illustrates this idea. It shows the communications between components in the CRM and their relationship to observable behavior. As you may remember from chapter 5, whether a method is part of the class's observable behavior depends on whom the client is and what the goals of that client are. To be part of the observable behavior, the method must meet one of the following two criteria:

- Have an immediate connection to one of the client's goals
- Incur a side effect in an out-of-process dependency that is visible to external applications

The controller's `ChangeEmail()` method is part of its observable behavior, and so is the call it makes to the message bus. The first method is the entry point for the external client, thereby meeting the first criterion. The call to the bus sends messages to external applications, thereby meeting the second criterion. You should verify both of

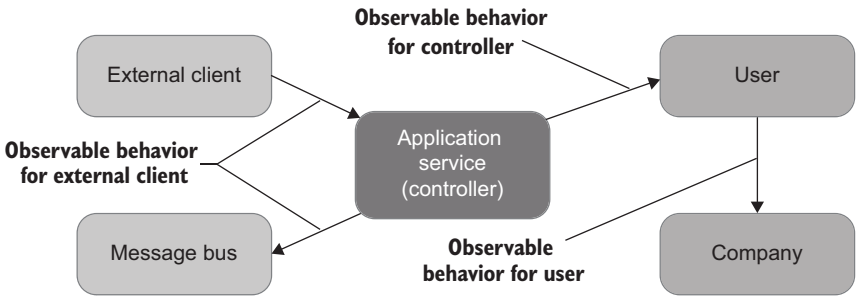

Figure 7.13 A map that shows communications among components in the CRM and the relationship between these communications and observable behavior

these method calls (which is the topic of the next chapter). However, the subsequent call from the controller to `User` doesn't have an immediate connection to the goals of the external client. That client doesn't care how the controller decides to implement the change of email as long as the final state of the system is correct and the call to the message bus is in place. Therefore, you shouldn't verify calls the controller makes to `User` when testing that controller's behavior.

When you step one level down the call stack, you get a similar situation. Now it's the controller who is the client, and the `ChangeEmail` method in `User` has an immediate connection to that client's goal of changing the user email and thus should be tested. But the subsequent calls from `User` to `Company` are implementation details from the controller's point of view. Therefore, the test that covers the `ChangeEmail` method in `User` shouldn't verify what methods `User` calls on `Company`. The same line of reasoning applies when you step one more level down and test the two methods in `Company` from `User`'s point of view.

Think of the observable behavior and implementation details as onion layers. Test each layer from the outer layer's point of view, and disregard how that layer talks to the underlying layers. As you peel these layers one by one, you switch perspective: what previously was an implementation detail now becomes an observable behavior, which you then cover with another set of tests.

Summary

- Code complexity is defined by the number of decision-making points in the code, both explicit (made by the code itself) and implicit (made by the libraries the code uses).
- Domain significance shows how significant the code is for the problem domain of your project. Complex code often has high domain significance and vice versa, but not in 100% of all cases.
- Complex code and code that has domain significance benefit from unit testing the most because the corresponding tests have greater protection against regressions.
- Unit tests that cover code with a large number of collaborators have high maintenance costs. Such tests require a lot of space to bring collaborators to an expected condition and then check their state or interactions with them afterward.
- All production code can be categorized into four types of code by its complexity or domain significance and the number of collaborators:
  - Domain model and algorithms (high complexity or domain significance, few collaborators) provide the best return on unit testing efforts.
  - Trivial code (low complexity and domain significance, few collaborators) isn't worth testing at all.

  - Controllers (low complexity and domain significance, large number of collaborators) should be tested briefly by integration tests.
  - Overcomplicated code (high complexity or domain significance, large number of collaborators) should be split into controllers and complex code.
- The more important or complex the code is, the fewer collaborators it should have.
- The Humble Object pattern helps make overcomplicated code testable by extracting business logic out of that code into a separate class. As a result, the remaining code becomes a controller—a thin, *humble* wrapper around the business logic.
- The hexagonal and functional architectures implement the Humble Object pattern. *Hexagonal architecture* advocates for the separation of business logic and communications with out-of-process dependencies. *Functional architecture* separates business logic from communications with *all* collaborators, not just out-of-process ones.
- Think of the business logic and orchestration responsibilities in terms of code depth versus code width. Your code can be either deep (complex or important) or wide (work with many collaborators), but never both.
- Test preconditions if they have a domain significance; don't test them otherwise.
- There are three important attributes when it comes to separating business logic from orchestration:
  - *Domain model testability*—A function of the number and the type of collaborators in domain classes
  - *Controller simplicity*—Depends on the presence of decision-making points in the controller
  - *Performance*—Defined by the number of calls to out-of-process dependencies
- You can have a maximum of two of these three attributes at any given moment:
  - *Pushing all external reads and writes to the edges of a business operation*—Preserves controller simplicity and keeps the domain model testability, but concedes performance
  - *Injecting out-of-process dependencies into the domain model*—Keeps performance and the controller's simplicity, but damages domain model testability
  - *Splitting the decision-making process into more granular steps*—Preserves performance and domain model testability, but gives up controller simplicity
- *Splitting the decision-making process into more granular steps*—Is a trade-off with the best set of pros and cons. You can mitigate the growth of controller complexity using the following two patterns:
  - The CanExecute/Execute pattern introduces a `CanDo()` for each `Do()` method and makes its successful execution a precondition for `Do()`. This pattern essentially eliminates the controller's decision-making because there's no option not to call `CanDo()` before `Do()`.

  - Domain events help track important changes in the domain model, and then convert those changes to calls to out-of-process dependencies. This pattern removes the tracking responsibility from the controller.
- It's easier to test abstractions than the things they abstract. Domain events are abstractions on top of upcoming calls to out-of-process dependencies. Changes in domain classes are abstractions on top of upcoming modifications in the data storage.