

Why integration testing?

This chapter covers

- Understanding the role of integration testing
- Diving deeper into the Test Pyramid concept
- Writing valuable integration tests

You can never be sure your system works as a whole if you rely on unit tests exclusively. Unit tests are great at verifying business logic, but it's not enough to check that logic in a vacuum. You have to validate how different parts of it integrate with each other and external systems: the database, the message bus, and so on.

In this chapter, you'll learn the role of integration tests: when you should apply them and when it's better to rely on plain old unit tests or even other techniques such as the Fail Fast principle. You will see which out-of-process dependencies to use as-is in integration tests and which to replace with mocks. You will also see integration testing best practices that will help improve the health of your code base in general: making domain model boundaries explicit, reducing the number of layers in the application, and eliminating circular dependencies. Finally, you'll learn why interfaces with a single implementation should be used sporadically, and how and when to test logging functionality.

8.1 What is an integration test?

Integration tests play an important role in your test suite. It's also crucial to balance the number of unit and integration tests. You will see shortly what that role is and how to maintain the balance, but first, let me give you a refresher on what differentiates an integration test from a unit test.

8.1.1 The role of integration tests

As you may remember from chapter 2, a *unit test* is a test that meets the following three requirements:

- Verifies a single unit of behavior,
- Does it quickly,
- And does it in isolation from other tests.

A test that doesn't meet at least one of these three requirements falls into the category of integration tests. An *integration test* then is any test that is not a unit test.

In practice, integration tests almost always verify how your system works in integration with out-of-process dependencies. In other words, these tests cover the code from the controllers quadrant (see chapter 7 for more details about code quadrants). The diagram in figure 8.1 shows the typical responsibilities of unit and integration tests. Unit tests cover the domain model, while integration tests check the code that glues that domain model with out-of-process dependencies.

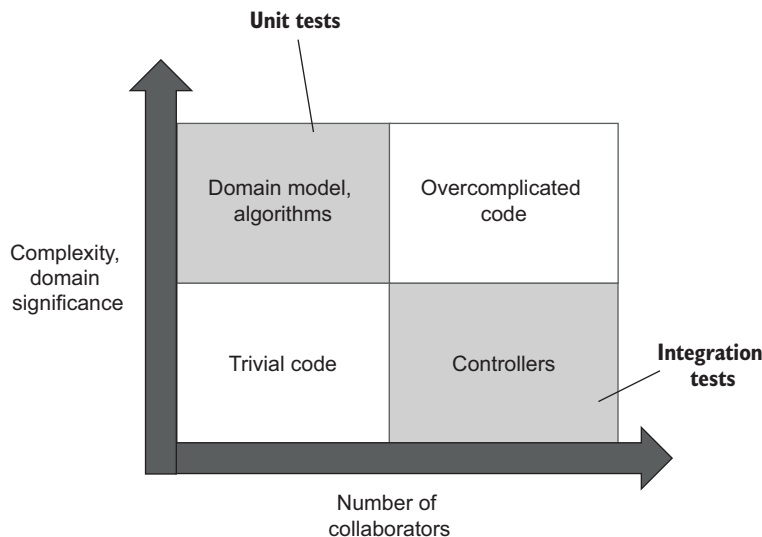


Figure 8.1 Integration tests cover controllers, while unit tests cover the domain model and algorithms. Trivial and overcomplicated code shouldn't be tested at all.

Note that tests covering the controllers quadrant can sometimes be unit tests too. If all out-of-process dependencies are replaced with mocks, there will be no dependencies shared between tests, which will allow those tests to remain fast and maintain their isolation from each other. Most applications do have an out-of-process dependency that can't be replaced with a mock, though. It's usually a database—a dependency that is not visible to other applications.

As you may also remember from chapter 7, the other two quadrants from figure 8.1 (trivial code and overcomplicated code) shouldn't be tested at all. Trivial code isn't worth the effort, while overcomplicated code should be refactored into algorithms and controllers. Thus, all your tests must focus on the domain model and the controllers quadrants exclusively.

8.1.2 The Test Pyramid revisited

It's important to maintain a balance between unit and integration tests. Working directly with out-of-process dependencies makes integration tests slow. Such tests are also more expensive to maintain. The increase in maintainability costs is due to

- The necessity to keep the out-of-process dependencies operational
- The greater number of collaborators involved, which inflates the test's size

On the other hand, integration tests go through a larger amount of code (both your code and the code of the libraries used by the application), which makes them better than unit tests at protecting against regressions. They are also more detached from the production code and therefore have better resistance to refactoring.

The ratio between unit and integration tests can differ depending on the project's specifics, but the general rule of thumb is the following: check as many of the business scenario's edge cases as possible with unit tests; use integration tests to cover one happy path, as well as any edge cases that can't be covered by unit tests.

DEFINITION A *happy path* is a successful execution of a business scenario. An *edge case* is when the business scenario execution results in an error.

Shifting the majority of the workload to unit tests helps keep maintenance costs low. At the same time, having one or two overarching integration tests per business scenario ensures the correctness of your system as a whole. This guideline forms the pyramid-like ratio between unit and integration tests, as shown in figure 8.2 (as discussed in chapter 2, end-to-end tests are a subset of integration tests).

The Test Pyramid can take different shapes depending on the project's complexity. Simple applications have little (if any) code in the domain model and algorithms quadrant. As a result, tests form a rectangle instead of a pyramid, with an equal number of unit and integration tests (figure 8.3). In the most trivial cases, you might have no unit tests whatsoever.

Note that integration tests retain their value even in simple applications. Regardless of how simple your code is, it's still important to verify how it works in integration with other subsystems.

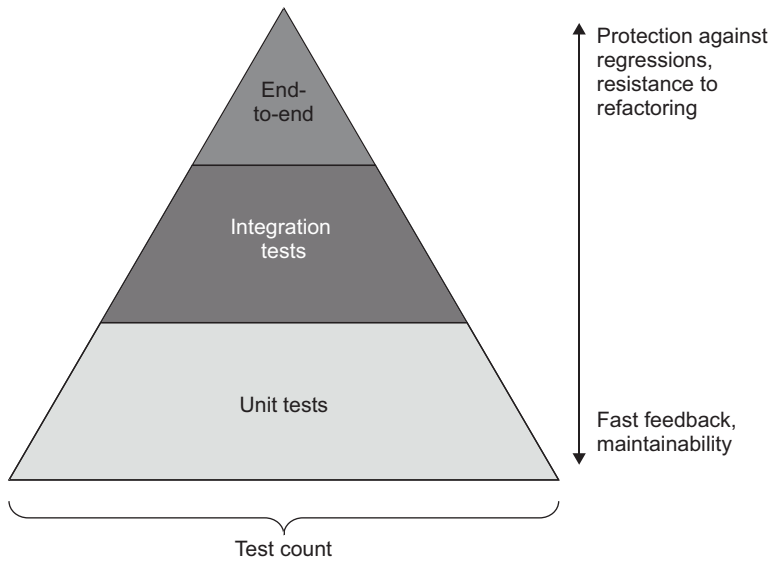


Figure 8.2 The Test Pyramid represents a trade-off that works best for most applications. Fast, cheap unit tests cover the majority of edge cases, while a smaller number of slow, more expensive integration tests ensure the correctness of the system as a whole.

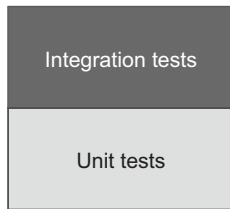


Figure 8.3 The Test Pyramid of a simple project. Little complexity requires a smaller number of unit tests compared to a normal pyramid.

8.1.3 *Integration testing vs. failing fast*

This section elaborates on the guideline of using integration tests to cover one happy path per business scenario and any edge cases that can't be covered by unit tests.

For an integration test, select the longest happy path in order to verify interactions with *all* out-of-process dependencies. If there's no one path that goes through all such interactions, write additional integration tests—as many as needed to capture communications with *every* external system.

As with the edge cases that can't be covered by unit tests, there are exceptions to this part of the guideline, too. There's no need to test an edge case if an incorrect execution of that edge case immediately fails the entire application. For example, you saw in chapter 7 how User from the sample CRM system implemented a `CanChangeEmail` method and made its successful execution a precondition for `ChangeEmail()`:

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

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

The controller invokes `CanChangeEmail()` and interrupts the operation if that method returns an error:

```
// UserController
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;

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

Edge case

This example shows the edge case you could theoretically cover with an integration test. Such a test doesn't provide a significant enough value, though. If the controller tries to change the email without consulting with `CanChangeEmail()` first, the application crashes. This bug reveals itself with the first execution and thus is easy to notice and fix. It also doesn't lead to data corruption.

TIP It's better to not write a test at all than to write a bad test. A test that doesn't provide significant value is a *bad* test.

Unlike the call from the controller to `CanChangeEmail()`, the presence of the precondition in `User` *should* be tested. But that is better done with a unit test; there's no need for an integration test.

Making bugs manifest themselves quickly is called the *Fail Fast principle*, and it's a viable alternative to integration testing.

The Fail Fast principle

The *Fail Fast principle* stands for stopping the current operation as soon as any unexpected error occurs. This principle makes your application more stable by

- *Shortening the feedback loop*—The sooner you detect a bug, the easier it is to fix. A bug that is already in production is orders of magnitude more expensive to fix compared to a bug found during development.
- *Protecting the persistence state*—Bugs lead to corruption of the application's state. Once that state penetrates into the database, it becomes much harder to fix. Failing fast helps you prevent the corruption from spreading.

(continued)

Stopping the current operation is normally done by throwing exceptions, because exceptions have semantics that are perfectly suited for the Fail Fast principle: they interrupt the program flow and pop up to the highest level of the execution stack, where you can log them and shut down or restart the operation.

Preconditions are one example of the Fail Fast principle in action. A failing precondition signifies an incorrect assumption made about the application state, which is always a bug. Another example is reading data from a configuration file. You can arrange the reading logic such that it will throw an exception if the data in the configuration file is incomplete or incorrect. You can also put this logic close to the application startup, so that the application doesn't launch if there's a problem with its configuration.

8.2 Which out-of-process dependencies to test directly

As I mentioned earlier, integration tests verify how your system integrates with out-of-process dependencies. There are two ways to implement such verification: use the real out-of-process dependency, or replace that dependency with a mock. This section shows when to apply each of the two approaches.

8.2.1 The two types of out-of-process dependencies

All out-of-process dependencies fall into two categories:

- *Managed dependencies (out-of-process dependencies you have full control over)*—These dependencies are only accessible through your application; interactions with them aren't visible to the external world. A typical example is a database. External systems normally don't access your database directly; they do that through the API your application provides.
- *Unmanaged dependencies (out-of-process dependencies you don't have full control over)*—Interactions with such dependencies *are* observable externally. Examples include an SMTP server and a message bus: both produce side effects visible to other applications.

I mentioned in chapter 5 that communications with managed dependencies are implementation details. Conversely, communications with unmanaged dependencies are part of your system's observable behavior (figure 8.4). This distinction leads to the difference in treatment of out-of-process dependencies in integration tests.

IMPORTANT Use real instances of managed dependencies; replace unmanaged dependencies with mocks.

As discussed in chapter 5, the requirement to preserve the communication pattern with unmanaged dependencies stems from the necessity to maintain backward compatibility with those dependencies. Mocks are perfect for this task. With mocks, you can ensure communication pattern permanence in light of any possible refactorings.

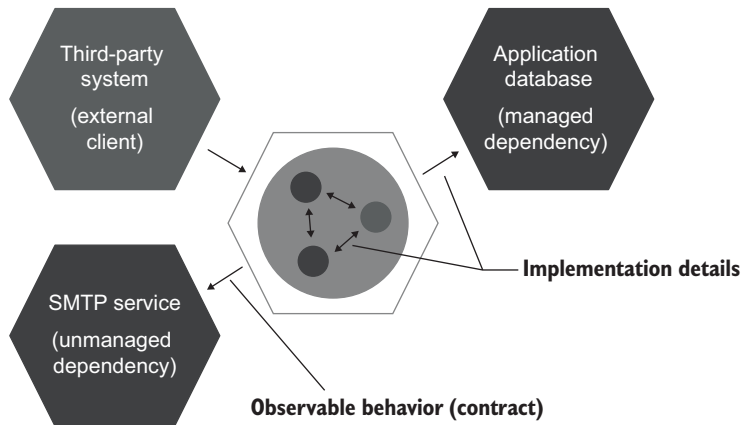


Figure 8.4 Communications with managed dependencies are implementation details; use such dependencies as-is in integration tests. Communications with unmanaged dependencies are part of your system’s observable behavior. Such dependencies should be mocked out.

However, there’s no need to maintain backward compatibility in communications with managed dependencies, because your application is the only one that talks to them. External clients don’t care how you organize your database; the only thing that matters is the final state of your system. Using real instances of managed dependencies in integration tests helps you verify that final state from the external client’s point of view. It also helps during database refactorings, such as renaming a column or even migrating from one database to another.

8.2.2 Working with both managed and unmanaged dependencies

Sometimes you’ll encounter an out-of-process dependency that exhibits attributes of both managed and unmanaged dependencies. A good example is a database that other applications have access to.

The story usually goes like this. A system begins with its own dedicated database. After a while, another system begins to require data from the same database. And so the team decides to share access to a limited number of tables just for ease of integration with that other system. As a result, the database becomes a dependency that is both managed and unmanaged. It still contains parts that are visible to your application only; but, in addition to those parts, it also has a number of tables accessible by other applications.

The use of a database is a poor way to implement integration between systems because it couples these systems to each other and complicates their further development. Only resort to this approach when all other options are exhausted. A better way to do the integration is via an API (for synchronous communications) or a message bus (for asynchronous communications).

But what do you do when you already have a shared database and can’t do anything about it in the foreseeable future? In this case, treat tables that are visible to

other applications as an unmanaged dependency. Such tables in effect act as a message bus, with their rows playing the role of messages. Use mocks to make sure the communication pattern with these tables remains unchanged. At the same time, treat the rest of your database as a managed dependency and verify its final state, not the interactions with it (figure 8.5).

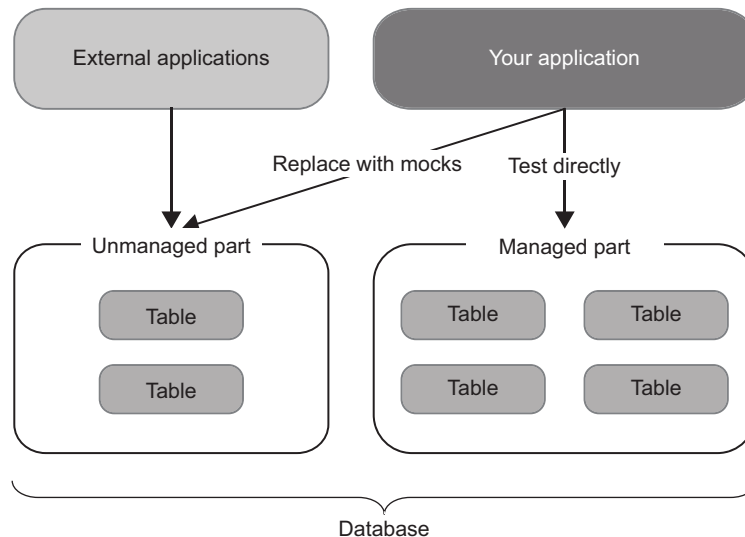


Figure 8.5 Treat the part of the database that is visible to external applications as an unmanaged dependency. Replace it with mocks in integration tests. Treat the rest of the database as a managed dependency. Verify its final state, not interactions with it.

It's important to differentiate these two parts of your database because, again, the shared tables are observable externally, and you need to be careful about how your application communicates with them. Don't change the way your system interacts with those tables unless absolutely necessary! You never know how other applications will react to such a change.

8.2.3 *What if you can't use a real database in integration tests?*

Sometimes, for reasons outside of your control, you just can't use a real version of a managed dependency in integration tests. An example would be a legacy database that you can't deploy to a test automation environment, not to mention a developer machine, because of some IT security policy, or because the cost of setting up and maintaining a test database instance is prohibitive.

What should you do in such a situation? Should you mock out the database anyway, despite it being a managed dependency? No, because mocking out a managed dependency compromises the integration tests' resistance to refactoring. Furthermore, such

tests no longer provide as good protection against regressions. And if the database is the only out-of-process dependency in your project, the resulting integration tests would deliver no additional protection compared to the existing set of unit tests (assuming these unit tests follow the guidelines from chapter 7).

The only thing such integration tests would do, in addition to unit tests, is check what repository methods the controller calls. In other words, you wouldn't really gain confidence about anything other than those three lines of code in your controller being correct, while still having to do a lot of plumbing.

If you can't test the database as-is, don't write integration tests at all, and instead, focus exclusively on unit testing of the domain model. Remember to always put all your tests under close scrutiny. Tests that don't provide a high enough value should have no place in your test suite.

8.3 Integration testing: An example

Let's get back to the sample CRM system from chapter 7 and see how it can be covered with integration tests. As you may recall, this system implements one feature: changing the user's email. It retrieves the user and the company from the database, delegates the decision-making to the domain model, and then saves the results back to the database and puts a message on the bus if needed (figure 8.6).

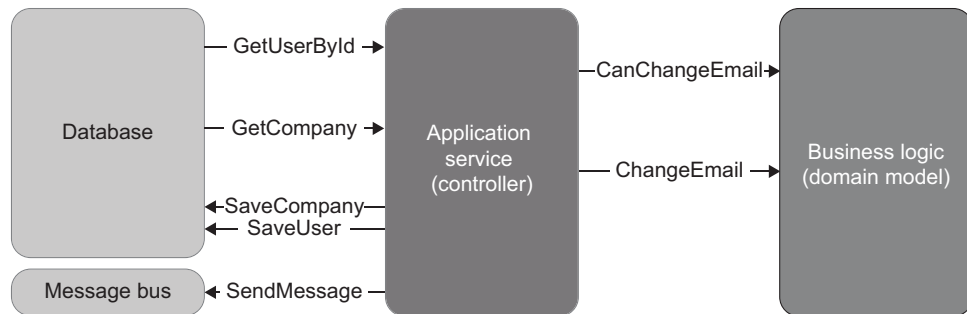


Figure 8.6 The use case of changing the user's email. The controller orchestrates the work between the database, the message bus, and the domain model.

The following listing shows how the controller currently looks.

Listing 8.1 The user controller

```

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

    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 (EmailChangedEvent ev in user.EmailChangedEvents)
{
    _messageBus.SendEmailChangedMessage(ev.UserId, ev.NewEmail);
}

return "OK";
}
}

```

In the following section, I'll first outline scenarios to verify using integration tests. Then I'll show you how to work with the database and the message bus in tests.

8.3.1 *What scenarios to test?*

As I mentioned earlier, the general guideline for integration testing is to cover the longest happy path and any edge cases that can't be exercised by unit tests. The *longest happy path* is the one that goes through all out-of-process dependencies.

In the CRM project, the longest happy path is a change from a corporate to a non-corporate email. Such a change leads to the maximum number of side effects:

- In the database, both the user and the company are updated: the user changes its type (from corporate to non-corporate) and email, and the company changes its number of employees.
- A message is sent to the message bus.

As for the edge cases that aren't tested by unit tests, there's only one such edge case: the scenario where the email can't be changed. There's no need to test this scenario, though, because the application will fail fast if this check isn't present in the controller. That leaves us with a single integration test:

```
public void Changing_email_from_corporate_to_non_corporate()
```

8.3.2 Categorizing the database and the message bus

Before writing the integration test, you need to categorize the two out-of-process dependencies and decide which of them to test directly and which to replace with a mock. The application database is a managed dependency because no other system can access it. Therefore, you should use a real instance of it. The integration test will

- Insert a user and a company into the database.
- Run the change of email scenario on that database.
- Verify the database state.

On the other hand, the message bus is an unmanaged dependency—its sole purpose is to enable communication with other systems. The integration test will mock out the message bus and verify the interactions between the controller and the mock afterward.

8.3.3 What about end-to-end testing?

There will be no end-to-end tests in our sample project. An end-to-end test in a scenario with an API would be a test running against a deployed, fully functioning version of that API, which means no mocks for any of the out-of-process dependencies (figure 8.7). On the other hand, integration tests host the application within the same process and substitute unmanaged dependencies with mocks (figure 8.8).

As I mentioned in chapter 2, whether to use end-to-end tests is a judgment call. For the most part, when you include managed dependencies in the integration testing scope and mock out only unmanaged dependencies, integration tests provide a level

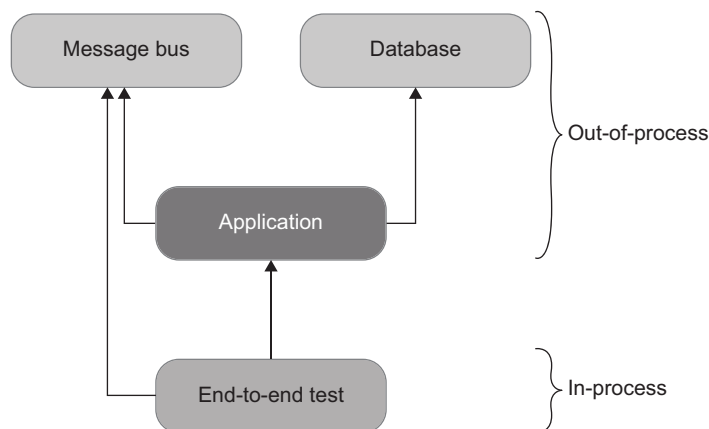


Figure 8.7 End-to-end tests emulate the external client and therefore test a deployed version of the application with all out-of-process dependencies included in the testing scope. End-to-end tests shouldn't check managed dependencies (such as the database) directly, only indirectly through the application.

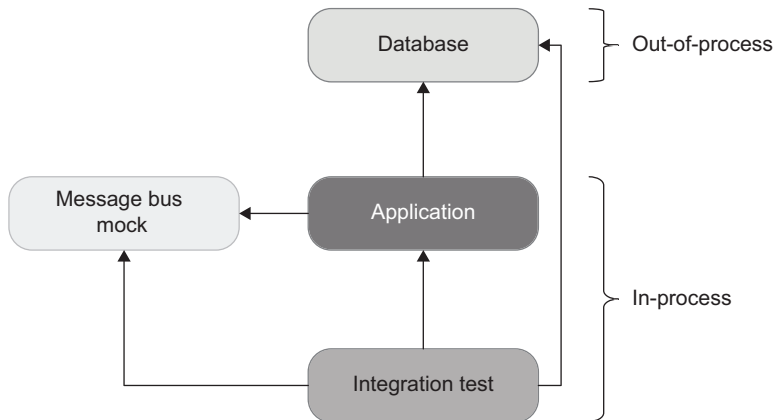


Figure 8.8 Integration tests host the application within the same process. Unlike end-to-end tests, integration tests substitute unmanaged dependencies with mocks. The only out-of-process components for integration tests are managed dependencies.

of protection that is close enough to that of end-to-end tests, so you can skip end-to-end testing. However, you could still create one or two overarching end-to-end tests that would provide a sanity check for the project after deployment. Make such tests go through the longest happy path, too, to ensure that your application communicates with all out-of-process dependencies properly. To emulate the external client's behavior, check the message bus directly, but verify the database's state through the application itself.

8.3.4 *Integration testing: The first try*

Here's the first version of the integration test.

Listing 8.2 The integration test

```

[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    // Arrange
    var db = new Database(connectionString);
    User user = CreateUser(
        "user@mycorp.com", UserType.Employee, db);
    CreateCompany("mycorp.com", 1, db);

    var messageBusMock = new Mock<IMessageBus>();
    var sut = new UserController(db, messageBusMock.Object);

    // Act
    string result = sut.ChangeEmail(user.UserId, "new@gmail.com");
  
```

Database repository

Creates the user and company in the database

Sets up a mock for the message bus

```
// Assert
Assert.Equal("OK", result);

object[] userData = db.GetUserById(user.UserId);
User userFromDb = UserFactory.Create(userData);
Assert.Equal("new@gmail.com", userFromDb.Email);
Assert.Equal(UserType.Customer, userFromDb.Type);

object[] companyData = db.GetCompany();
Company companyFromDb = CompanyFactory
    .Create(companyData);
Assert.Equal(0, companyFromDb.NumberOfEmployees);

messageBusMock.Verify(
    x => x.SendEmailChangedMessage(
        user.UserId, "new@gmail.com"),
    Times.Once);
}
```

Asserts the
user's state

Asserts the
company's
state

Checks the
interactions
with the mock

TIP Notice that in the arrange section, the test doesn't insert the user and the company into the database on its own but instead calls the `CreateUser` and `CreateCompany` helper methods. These methods can be reused across multiple integration tests.

It's important to check the state of the database independently of the data used as input parameters. To do that, the integration test queries the user and company data separately in the assert section, creates new `userFromDb` and `companyFromDb` instances, and only then asserts their state. This approach ensures that the test exercises both writes to and reads from the database and thus provides the maximum protection against regressions. The reading itself must be implemented using the same code the controller uses internally: in this example, using the `Database`, `UserFactory`, and `CompanyFactory` classes.

This integration test, while it gets the job done, can still benefit from some improvement. For instance, you could use helper methods in the assertion section, too, in order to reduce this section's size. Also, `messageBusMock` doesn't provide as good protection against regressions as it potentially could. We'll talk about these improvements in the subsequent two chapters where we discuss mocking and database testing best practices.

8.4 Using interfaces to abstract dependencies

One of the most misunderstood subjects in the sphere of unit testing is the use of interfaces. Developers often ascribe invalid reasons to why they introduce interfaces and, as a result, tend to overuse them. In this section, I'll expand on those invalid reasons and show in what circumstances the use of interfaces is and isn't preferable.

8.4.1 Interfaces and loose coupling

Many developers introduce interfaces for out-of-process dependencies, such as the database or the message bus, even when these interfaces have only one implementation. This practice has become so widespread nowadays that hardly anyone questions it. You'll often see class-interface pairs similar to the following:

```
public interface IMessageBus
public class MessageBus : IMessageBus

public interface IUserRepository
public class UserRepository : IUserRepository
```

The common reasoning behind the use of such interfaces is that they help to

- Abstract out-of-process dependencies, thus achieving loose coupling
- Add new functionality without changing the existing code, thus adhering to the Open-Closed principle (OCP)

Both of these reasons are misconceptions. Interfaces with a single implementation are not abstractions and don't provide loose coupling any more than concrete classes that implement those interfaces. Genuine abstractions are *discovered*, not *invented*. The discovery, by definition, takes place post factum, when the abstraction already exists but is not yet clearly defined in the code. Thus, for an interface to be a genuine abstraction, it must have at least two implementations.

The second reason (the ability to add new functionality without changing the existing code) is a misconception because it violates a more foundational principle: YAGNI. YAGNI stands for “You aren't gonna need it” and advocates against investing time in functionality that's not needed right now. You shouldn't develop this functionality, nor should you modify your existing code to account for the appearance of such functionality in the future. The two major reasons are as follows:

- *Opportunity cost*—If you spend time on a feature that business people don't need at the moment, you steer that time away from features they do need right now. Moreover, when the business people finally come to require the developed functionality, their view on it will most likely have evolved, and you will still need to adjust the already-written code. Such activity is wasteful. It's more beneficial to implement the functionality from scratch when the actual need for it emerges.
- *The less code in the project, the better*. Introducing code *just in case* without an immediate need unnecessarily increases your code base's cost of ownership. It's better to postpone introducing new functionality until as late a stage of your project as possible.

TIP Writing code is an expensive way to solve problems. The less code the solution requires and the simpler that code is, the better.

There are exceptional cases where YAGNI doesn't apply, but these are few and far between. For those cases, see my article “OCP vs YAGNI,” at <https://enterprisecraftsmanship.com/posts/ocp-vs-yagni>.

8.4.2 Why use interfaces for out-of-process dependencies?

So, why use interfaces for out-of-process dependencies at all, assuming that each of those interfaces has only one implementation? The real reason is much more practical and down-to-earth. It's to enable mocking—as simple as that. Without an interface, you can't create a test double and thus can't verify interactions between the system under test and the out-of-process dependency.

Therefore, *don't introduce interfaces for out-of-process dependencies unless you need to mock out those dependencies*. You only mock out unmanaged dependencies, so the guideline can be boiled down to this: *use interfaces for unmanaged dependencies only*. Still inject managed dependencies into the controller explicitly, but use concrete classes for that.

Note that genuine abstractions (abstractions that have more than one implementation) can be represented with interfaces regardless of whether you mock them out. Introducing an interface with a single implementation for reasons other than mocking is a violation of YAGNI, however.

And you might have noticed in listing 8.2 that `UserController` now accepts both the message bus and the database explicitly via the constructor, but only the message bus has a corresponding interface. The database is a managed dependency and thus doesn't require such an interface. Here's the controller:

```
public class UserController
{
    private readonly Database _database;
    private readonly IMessageBus _messageBus;

    public UserController(Database database, IMessageBus messageBus)
    {
        _database = database;
        _messageBus = messageBus;
    }

    public string ChangeEmail(int userId, string newEmail)
    {
        /* the method uses _database and _messageBus */
    }
}
```

NOTE You can mock out a dependency without resorting to an interface by making methods in that dependency virtual and using the class itself as a base for the mock. This approach is inferior to the one with interfaces, though. I explain more on this topic of interfaces versus base classes in chapter 11.

8.4.3 Using interfaces for in-process dependencies

You sometimes see code bases where interfaces back not only out-of-process dependencies but in-process dependencies as well. For example:

```
public interface IUser
{
    int UserId { get; set; }
```

```

    string Email { get; }
    string CanChangeEmail();
    void ChangeEmail(string newEmail, Company company);
}

public class User : IUser
{
    /* ... */
}

```

Assuming that `IUser` has only one implementation (and such specific interfaces always have only one implementation), this is a huge red flag. Just like with out-of-process dependencies, the only reason to introduce an interface with a single implementation for a domain class is to enable mocking. But unlike out-of-process dependencies, you should never check interactions between domain classes, because doing so results in brittle tests: tests that couple to implementation details and thus fail on the metric of resisting to refactoring (see chapter 5 for more details about mocks and test fragility).

8.5 *Integration testing best practices*

There are some general guidelines that can help you get the most out of your integration tests:

- Making domain model boundaries explicit
- Reducing the number of layers in the application
- Eliminating circular dependencies

As usual, best practices that are beneficial for tests also tend to improve the health of your code base in general.

8.5.1 *Making domain model boundaries explicit*

Try to always have an explicit, well-known place for the domain model in your code base. The *domain model* is the collection of domain knowledge about the problem your project is meant to solve. Assigning the domain model an explicit boundary helps you better visualize and reason about that part of your code.

This practice also helps with testing. As I mentioned earlier in this chapter, unit tests target the domain model and algorithms, while integration tests target controllers. The explicit boundary between domain classes and controllers makes it easier to tell the difference between unit and integration tests.

The boundary itself can take the form of a separate assembly or a namespace. The particulars aren't that important as long as all of the domain logic is put under a single, distinct umbrella and not scattered across the code base.

8.5.2 *Reducing the number of layers*

Most programmers naturally gravitate toward abstracting and generalizing the code by introducing additional layers of indirection. In a typical enterprise-level application, you can easily observe several such layers (figure 8.9).

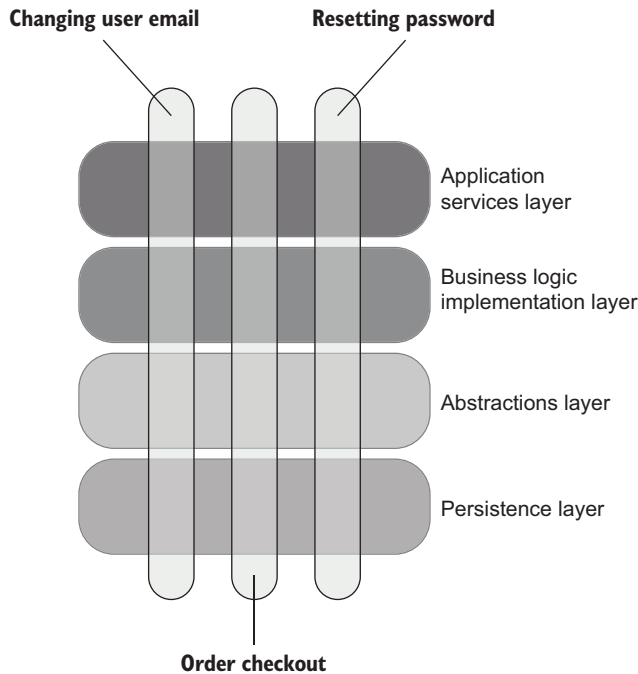


Figure 8.9 Various application concerns are often addressed by separate layers of indirection. A typical feature takes up a small portion of each layer.

In extreme cases, an application gets so many abstraction layers that it becomes too hard to navigate the code base and understand the logic behind even the simplest operations. At some point, you just want to get to the specific solution of the problem at hand, not some generalization of that solution in a vacuum.

All problems in computer science can be solved by another layer of indirection, except for the problem of too many layers of indirection.

—David J. Wheeler

Layers of indirection negatively affect your ability to reason about the code. When every feature has a representation in each of those layers, you have to expend significant effort assembling all the pieces into a cohesive picture. This creates an additional mental burden that handicaps the entire development process.

An excessive number of abstractions doesn't help unit or integration testing, either. Code bases with many layers of indirections tend not to have a clear boundary between controllers and the domain model (which, as you might remember from chapter 7, is a precondition for effective tests). There's also a much stronger tendency to verify each layer separately. This tendency results in a lot of low-value integration tests, each of which exercises only the code from a specific layer and mocks out layers

underneath. The end result is always the same: insufficient protection against regressions combined with low resistance to refactoring.

Try to have as few layers of indirection as possible. In most backend systems, you can get away with just three: the domain model, application services layer (controllers), and infrastructure layer. The infrastructure layer typically consists of algorithms that don't belong to the domain model, as well as code that enables access to out-of-process dependencies (figure 8.10).

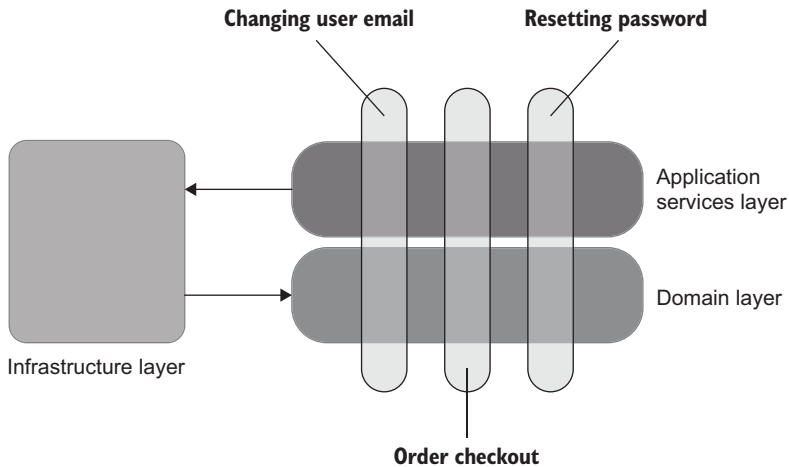


Figure 8.10 You can get away with just three layers: the domain layer (contains domain logic), application services layers (provides an entry point for the external client, and coordinates the work between domain classes and out-of-process dependencies), and infrastructure layer (works with out-of-process dependencies; database repositories, ORM mappings, and SMTP gateways reside in this layer).

8.5.3 *Eliminating circular dependencies*

Another practice that can drastically improve the maintainability of your code base and make testing easier is eliminating circular dependencies.

DEFINITION A *circular dependency* (also known as *cyclic dependency*) is two or more classes that directly or indirectly depend on each other to function properly.

A typical example of a circular dependency is a callback:

```
public class CheckOutService
{
    public void CheckOut(int orderId)
    {
        var service = new ReportGenerationService();
        service.GenerateReport(orderId, this);
    }
}
```

```

        /* other code */
    }
}

public class ReportGenerationService
{
    public void GenerateReport(
        int orderId,
        CheckOutService checkOutService)
    {
        /* calls checkOutService when generation is completed */
    }
}

```

Here, `CheckOutService` creates an instance of `ReportGenerationService` and passes itself to that instance as an argument. `ReportGenerationService` calls `CheckOutService` back to notify it about the result of the report generation.

Just like an excessive number of abstraction layers, circular dependencies add tremendous cognitive load when you try to read and understand the code. The reason is that circular dependencies don't give you a clear starting point from which you can begin exploring the solution. To understand just one class, you have to read and understand the whole graph of its siblings all at once. Even a small set of interdependent classes can quickly become too hard to grasp.

Circular dependencies also interfere with testing. You often have to resort to interfaces and mocking in order to split the class graph and isolate a single unit of behavior, which, again, is a no-go when it comes to testing the domain model (more on that in chapter 5).

Note that the use of interfaces only masks the problem of circular dependencies. If you introduce an interface for `CheckOutService` and make `ReportGenerationService` depend on that interface instead of the concrete class, you remove the circular dependency at compile time (figure 8.11), but the cycle still persists at runtime. Even though the compiler no longer regards this class composition as a circular reference, the cognitive load required to understand the code doesn't become any smaller. If anything, it increases due to the additional interface.

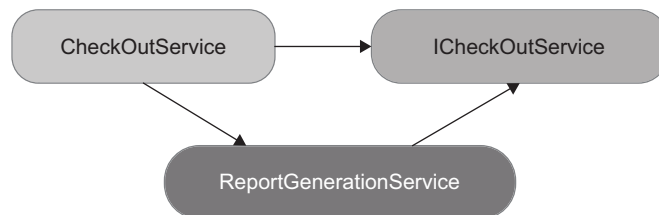


Figure 8.11 With an interface, you remove the circular dependency at compile time, but not at runtime. The cognitive load required to understand the code doesn't become any smaller.

A better approach to handle circular dependencies is to get rid of them. Refactor `ReportGenerationService` such that it depends on neither `CheckOutService` nor the `ICheckOutService` interface, and make `ReportGenerationService` return the result of its work as a plain value instead of calling `CheckOutService`:

```
public class CheckOutService
{
    public void CheckOut(int orderId)
    {
        var service = new ReportGenerationService();
        Report report = service.GenerateReport(orderId);

        /* other work */
    }
}

public class ReportGenerationService
{
    public Report GenerateReport(int orderId)
    {
        /* ... */
    }
}
```

It's rarely possible to eliminate all circular dependencies in your code base. But even then, you can minimize the damage by making the remaining graphs of interdependent classes as small as possible.

8.5.4 *Using multiple act sections in a test*

As you might remember from chapter 3, having more than one arrange, act, or assert section in a test is a code smell. It's a sign that this test checks multiple units of behavior, which, in turn, hinders the test's maintainability. For example, if you have two related use cases—say, user registration and user deletion—it might be tempting to check both of these use cases in a single integration test. Such a test could have the following structure:

- *Arrange*—Prepare data with which to register a user.
- *Act*—Call `UserController.RegisterUser()`.
- *Assert*—Query the database to see if the registration is completed successfully.
- *Act*—Call `UserController.DeleteUser()`.
- *Assert*—Query the database to make sure the user is deleted.

This approach is compelling because the user states naturally flow from one another, and the first act (registering a user) can simultaneously serve as an arrange phase for the subsequent act (user deletion). The problem is that such tests lose focus and can quickly become too bloated.

It's best to split the test by extracting each act into a test of its own. It may seem like unnecessary work (after all, why create two tests where one would suffice?), but this

work pays off in the long run. Having each test focus on a single unit of behavior makes those tests easier to understand and modify when necessary.

The exception to this guideline is tests working with out-of-process dependencies that are hard to bring to a desirable state. Let's say for example that registering a user results in creating a bank account in an external banking system. The bank has provisioned a sandbox for your organization, and you want to use that sandbox in an end-to-end test. The problem is that the sandbox is too slow, or maybe the bank limits the number of calls you can make to that sandbox. In such a scenario, it becomes beneficial to combine multiple acts into a single test and thus reduce the number of interactions with the problematic out-of-process dependency.

Hard-to-manage out-of-process dependencies are the only legitimate reason to write a test with more than one act section. This is why you should never have multiple acts in a unit test—unit tests don't work with out-of-process dependencies. Even integration tests should rarely have several acts. In practice, multistep tests almost always belong to the category of end-to-end tests.

8.6 How to test logging functionality

Logging is a gray area, and it isn't obvious what to do with it when it comes to testing. This is a complex topic that I'll split into the following questions:

- Should you test logging at all?
- If so, how should you test it?
- How much logging is enough?
- How do you pass around logger instances?

We'll use our sample CRM project as an example.

8.6.1 Should you test logging?

Logging is a cross-cutting functionality, which you can require in any part of your code base. Here's an example of logging in the `User` class.

Listing 8.3 An example of logging in `User`

```
public class User
{
    public void ChangeEmail(string newEmail, Company company)
    {
        _logger.Info(
            $"Changing email for user {UserId} to {newEmail}");

        Precondition.Requires(CanChangeEmail() == null);

        if (Email == newEmail)
            return;

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

Start of the method →

```

        if (Type != newType)
        {
            int delta = newType == UserType.Employee ? 1 : -1;
            company.ChangeNumberOfEmployees(delta);
            _logger.Info(
                $"User {UserId} changed type " +
                $"from {Type} to {newType}");
        }

        Email = newEmail;
        Type = newType;
        EmailChangedEvents.Add(new EmailChangedEvent(UserId, newEmail));

        _logger.Info(
            $"Email is changed for user {UserId}");
    }
}

```

End of the method →

Changes the user type

The User class records in a log file each beginning and ending of the `ChangeEmail` method, as well as the change of the user type. Should you test this functionality?

On the one hand, logging generates important information about the application's behavior. But on the other hand, logging can be so ubiquitous that it's not obvious whether this functionality is worth the additional, quite significant, testing effort. The answer to the question of whether you should test logging comes down to this: *Is logging part of the application's observable behavior, or is it an implementation detail?*

In that sense, it isn't different from any other functionality. Logging ultimately results in side effects in an out-of-process dependency such as a text file or a database. If these side effects are meant to be observed by your customer, the application's clients, or anyone else other than the developers themselves, then logging is an observable behavior and thus must be tested. If the only audience is the developers, then it's an implementation detail that can be freely modified without anyone noticing, in which case it shouldn't be tested.

For example, if you write a logging library, then the logs this library produces are the most important (and the only) part of its observable behavior. Another example is when business people insist on logging key application workflows. In this case, logs also become a business requirement and thus have to be covered by tests. However, in the latter example, you might also have separate logging just for developers.

Steve Freeman and Nat Pryce, in their book *Growing Object-Oriented Software, Guided by Tests* (Addison-Wesley Professional, 2009), call these two types of logging support logging and diagnostic logging:

- *Support logging* produces messages that are intended to be tracked by support staff or system administrators.
- *Diagnostic logging* helps developers understand what's going on inside the application.

8.6.2 How should you test logging?

Because logging involves out-of-process dependencies, when it comes to testing it, the same rules apply as with any other functionality that touches out-of-process dependencies. You need to use mocks to verify interactions between your application and the log storage.

INTRODUCING A WRAPPER ON TOP OF ILOGGER

But don't just mock out the ILogger interface. Because support logging is a business requirement, reflect that requirement explicitly in your code base. Create a special DomainLogger class where you explicitly list all the support logging needed for the business; verify interactions with that class instead of the raw ILogger.

For example, let's say that business people require you to log all changes of the users' types, but the logging at the beginning and the end of the method is there just for debugging purposes. The next listing shows the User class after introducing a DomainLogger class.

Listing 8.4 Extracting support logging into the DomainLogger class

```

public void ChangeEmail(string newEmail, Company company)
{
    Diagnostic logging → _logger.Info(
        $"Changing email for user {UserId} to {newEmail}");

    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);
        Support logging | _domainLogger.UserTypeHasChanged(
            UserId, Type, newType);
    }

    Email = newEmail;
    Type = newType;
    EmailChangedEvents.Add(new EmailChangedEvent(UserId, newEmail));

    Diagnostic logging → _logger.Info(
        $"Email is changed for user {UserId}");
}

```

The diagnostic logging still uses the old logger (which is of type ILogger), but the support logging now uses the new domainLogger instance of type IDomainLogger. The following listing shows the implementation of IDomainLogger.

Listing 8.5 DomainLogger as a wrapper on top of ILogger

```
public class DomainLogger : IDomainLogger
{
    private readonly ILogger _logger;

    public DomainLogger(ILogger logger)
    {
        _logger = logger;
    }

    public void UserTypeHasChanged(
        int userId, UserType oldType, UserType newType)
    {
        _logger.Info(
            $"User {userId} changed type " +
            $"from {oldType} to {newType}");
    }
}
```

DomainLogger works on top of ILogger: it uses the domain language to declare specific log entries required by the business, thus making support logging easier to understand and maintain. In fact, this implementation is very similar to the concept of structured logging, which enables great flexibility when it comes to log file post-processing and analysis.

UNDERSTANDING STRUCTURED LOGGING

Structured logging is a logging technique where capturing log data is decoupled from the rendering of that data. Traditional logging works with simple text. A call like

```
logger.Info("User Id is " + 12);
```

first forms a string and then writes that string to a log storage. The problem with this approach is that the resulting log files are hard to analyze due to the lack of structure. For example, it's not easy to see how many messages of a particular type there are and how many of those relate to a specific user ID. You'd need to use (or even write your own) special tooling for that.

On the other hand, structured logging introduces structure to your log storage. The use of a structured logging library looks similar on the surface:

```
logger.Info("User Id is {UserId}", 12);
```

But its underlying behavior differs significantly. Behind the scenes, this method computes a hash of the message template (the message itself is stored in a lookup storage for space efficiency) and combines that hash with the input parameters to form a set of *captured data*. The next step is the rendering of that data. You can still have a flat log file, as with traditional logging, but that's just one possible rendering. You could also configure the logging library to render the captured data as a JSON or a CSV file, where it would be easier to analyze (figure 8.12).

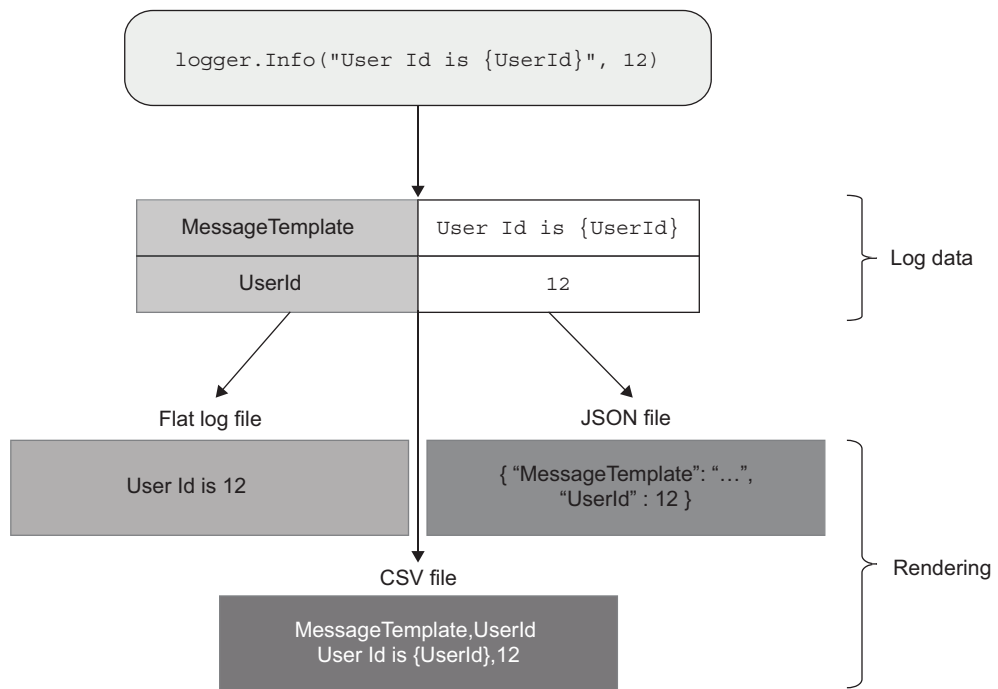


Figure 8.12 Structured logging decouples log data from renderings of that data. You can set up multiple renderings, such as a flat log file, JSON, or CSV file.

DomainLogger in listing 8.5 isn't a structured logger per se, but it operates in the same spirit. Look at this method once again:

```
public void UserTypeHasChanged(
    int userId, UserType oldType, UserType newType)
{
    _logger.Info(
        $"User {userId} changed type " +
        $"from {oldType} to {newType}");
}
```

You can view `UserTypeHasChanged()` as the message template's hash. Together with the `userId`, `oldType`, and `newType` parameters, that hash forms the log data. The method's implementation renders the log data into a flat log file. And you can easily create additional renderings by also writing the log data into a JSON or a CSV file.

WRITING TESTS FOR SUPPORT AND DIAGNOSTIC LOGGING

As I mentioned earlier, DomainLogger represents an out-of-process dependency—the log storage. This poses a problem: User now interacts with that dependency and thus violates the separation between business logic and communication with out-of-process dependencies. The use of DomainLogger has transitioned User to the category of

overcomplicated code, making it harder to test and maintain (refer to chapter 7 for more details about code categories).

This problem can be solved the same way we implemented the notification of external systems about changed user emails: with the help of domain events (again, see chapter 7 for details). You can introduce a separate domain event to track changes in the user type. The controller will then convert those changes into calls to Domain-Logger, as shown in the following listing.

Listing 8.6 Replacing DomainLogger in User with a domain event

```
public void ChangeEmail(string newEmail, Company company)
{
    _logger.Info(
        $"Changing email for user {UserId} to {newEmail}");

    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);
        AddDomainEvent(
            new UserTypeChangedEvent(
                UserId, Type, newType));
    }

    Email = newEmail;
    Type = newType;
    AddDomainEvent(new EmailChangedEvent(UserId, newEmail));

    _logger.Info($"Email is changed for user {UserId}");
}
```

**Uses a domain
event instead of
DomainLogger**

Notice that there are now two domain events: `UserTypeChangedEvent` and `EmailChangedEvent`. Both of them implement the same interface (`IDomainEvent`) and thus can be stored in the same collection.

And here is how the controller looks.

Listing 8.7 Latest version of UserController

```
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);
_eventDispatcher.Dispatch(user.DomainEvents);
return "OK";
}

```



EventDispatcher is a new class that converts domain events into calls to out-of-process dependencies:

- EmailChangedEvent translates into `_messageBus.SendEmailChangedMessage()`.
- UserTypeChangedEvent translates into `_domainLogger.UserTypeHasChanged()`.

The use of UserTypeChangedEvent has restored the separation between the two responsibilities: domain logic and communication with out-of-process dependencies. Testing support logging now isn't any different from testing the other *unmanaged* dependency, the message bus:

- Unit tests should check an instance of UserTypeChangedEvent in the User under test.
- The single integration test should use a mock to ensure the interaction with DomainLogger is in place.

Note that if you need to do support logging in the controller and not one of the domain classes, there's no need to use domain events. As you may remember from chapter 7, controllers orchestrate the collaboration between the domain model and out-of-process dependencies. DomainLogger is one of such dependencies, and thus UserController can use that logger directly.

Also notice that I didn't change the way the User class does diagnostic logging. User still uses the logger instance directly in the beginning and at the end of its ChangeEmail method. This is by design. Diagnostic logging is for developers only; you don't need to unit test this functionality and thus don't have to keep it out of the domain model.

Still, refrain from the use of diagnostic logging in User or other domain classes when possible. I explain why in the next section.

8.6.3 *How much logging is enough?*

Another important question is about the optimum amount of logging. How much logging is enough? Support logging is out of the question here because it's a business requirement. You do have control over diagnostic logging, though.

It's important not to overuse diagnostic logging, for the following two reasons:

- *Excessive logging clutters the code.* This is especially true for the domain model. That's why I don't recommend using diagnostic logging in `User` even though such a use is fine from a unit testing perspective: it obscures the code.
- *Logs' signal-to-noise ratio is key.* The more you log, the harder it is to find relevant information. Maximize the signal; minimize the noise.

Try not to use diagnostic logging in the domain model at all. In most cases, you can safely move that logging from domain classes to controllers. And even then, resort to diagnostic logging only temporarily when you need to debug something. Remove it once you finish debugging. Ideally, you should use diagnostic logging for unhandled exceptions only.

8.6.4 *How do you pass around logger instances?*

Finally, the last question is how to pass logger instances in the code. One way to resolve these instances is using static methods, as shown in the following listing.

Listing 8.8 Storing `ILogger` in a static field

```
public class User
{
    private static readonly ILogger _logger =
        LogManager.GetLogger(typeof(User));

    public void ChangeEmail(string newEmail, Company company)
    {
        _logger.Info(
            $"Changing email for user {UserId} to {newEmail}");

        /* ... */

        _logger.Info($"Email is changed for user {UserId}");
    }
}
```

Resolves `ILogger` through a static method, and stores it in a private static field

Steven van Deursen and Mark Seeman, in their book *Dependency Injection Principles, Practices, Patterns* (Manning Publications, 2018), call this type of dependency acquisition *ambient context*. This is an anti-pattern. Two of their arguments are that

- The dependency is hidden and hard to change.
- Testing becomes more difficult.

I fully agree with this analysis. To me, though, the main drawback of ambient context is that it masks potential problems in code. If injecting a logger explicitly into a

domain class becomes so inconvenient that you have to resort to ambient context, that's a certain sign of trouble. You either log too much or use too many layers of indirection. In any case, ambient context is not a solution. Instead, tackle the root cause of the problem.

The following listing shows one way to explicitly inject the logger: as a method argument. Another way is through the class constructor.

Listing 8.9 Injecting the logger explicitly

```
public void ChangeEmail(  
    string newEmail,  
    Company company,  
    ILogger logger)  
{  
    logger.Info(  
        $"Changing email for user {UserId} to {newEmail}");  
  
    /* ... */  
  
    logger.Info($"Email is changed for user {UserId}");  
}
```

**Method
injection**

8.7 Conclusion

View communications with all out-of-process dependencies through the lens of whether this communication is part of the application's observable behavior or an implementation detail. The log storage isn't any different in that regard. Mock logging functionality if the logs are observable by non-programmers; don't test it otherwise. In the next chapter, we'll dive deeper into the topic of mocking and best practices related to it.

Summary

- An *integration test* is any test that is not a unit test. Integration tests verify how your system works in integration with out-of-process dependencies:
 - Integration tests cover controllers; unit tests cover algorithms and the domain model.
 - Integration tests provide better protection against regressions and resistance to refactoring; unit tests have better maintainability and feedback speed.
- The bar for integration tests is higher than for unit tests: the score they have in the metrics of protection against regressions and resistance to refactoring must be higher than that of a unit test to offset the worse maintainability and feedback speed. The *Test Pyramid* represents this trade-off: the majority of tests should be fast and cheap unit tests, with a smaller number of slow and more expensive integration tests that check correctness of the system as a whole:
 - Check as many of the business scenario's edge cases as possible with unit tests. Use integration tests to cover one happy path, as well as any edge cases that can't be covered by unit tests.

- The shape of the Test Pyramid depends on the project’s complexity. Simple projects have little code in the domain model and thus can have an equal number of unit and integration tests. In the most trivial cases, there might be no unit tests.
- The *Fail Fast principle* advocates for making bugs manifest themselves quickly and is a viable alternative to integration testing.
- *Managed dependencies* are out-of-process dependencies that are only accessible through your application. Interactions with managed dependencies aren’t observable externally. A typical example is the application database.
- *Unmanaged dependencies* are out-of-process dependencies that other applications have access to. Interactions with unmanaged dependencies are observable externally. Typical examples include an SMTP server and a message bus.
- Communications with managed dependencies are implementation details; communications with unmanaged dependencies are part of your system’s observable behavior.
- Use real instances of managed dependencies in integration tests; replace unmanaged dependencies with mocks.
- Sometimes an out-of-process dependency exhibits attributes of both managed and unmanaged dependencies. A typical example is a database that other applications have access to. Treat the observable part of the dependency as an unmanaged dependency: replace that part with mocks in tests. Treat the rest of the dependency as a managed dependency: verify its final state, not interactions with it.
- An integration test must go through all layers that work with a managed dependency. In an example with a database, this means checking the state of that database independently of the data used as input parameters.
- Interfaces with a single implementation are not abstractions and don’t provide loose coupling any more than the concrete classes that implement those interfaces. Trying to anticipate future implementations for such interfaces violates the YAGNI (you aren’t gonna need it) principle.
- The only legitimate reason to use interfaces with a single implementation is to enable mocking. Use such interfaces only for unmanaged dependencies. Use concrete classes for managed dependencies.
- Interfaces with a single implementation used for in-process dependencies are a red flag. Such interfaces hint at using mocks to check interactions between domain classes, which leads to coupling tests to the code’s implementation details.
- Have an explicit and well-known place for the domain model in your code base. The explicit boundary between domain classes and controllers makes it easier to tell unit and integration tests apart.
- An excessive number of layers of indirection negatively affects your ability to reason about the code. Have as few layers of indirections as possible. In most

backend systems, you can get away with just three of them: the domain model, an application services layer (controllers), and an infrastructure layer.

- Circular dependencies add cognitive load when you try to understand the code. A typical example is a callback (when a callee notifies the caller about the result of its work). Break the cycle by introducing a value object; use that value object to return the result from the callee to the caller.
- Multiple act sections in a test are only justified when that test works with out-of-process dependencies that are hard to bring into a desirable state. You should never have multiple acts in a unit test, because unit tests don't work with out-of-process dependencies. Multistep tests almost always belong to the category of end-to-end tests.
- Support logging is intended for support staff and system administrators; it's part of the application's observable behavior. Diagnostic logging helps developers understand what's going on inside the application: it's an implementation detail.
- Because support logging is a business requirement, reflect that requirement explicitly in your code base. Introduce a special `DomainLogger` class where you list all the support logging needed for the business.
- Treat support logging like any other functionality that works with an out-of-process dependency. Use domain events to track changes in the domain model; convert those domain events into calls to `DomainLogger` in controllers.
- Don't test diagnostic logging. Unlike support logging, you can do diagnostic logging directly in the domain model.
- Use diagnostic logging sporadically. Excessive diagnostic logging clutters the code and damages the logs' signal-to-noise ratio. Ideally, you should only use diagnostic logging for unhandled exceptions.
- Always inject all dependencies explicitly (including loggers), either via the constructor or as a method argument.