Dependency Injection

Principles, Practices, Patterns

Steven van Deursen
Mark Seemann

MANNING

MEAP Edition
Manning Early Access Program

Dependency Injection

Principles, practices, Patterns

Version 13

Revised edition of Dependency Edition in .NET

welcome

Thank you for purchasing the MEAP for *Dependency Injection: Principles, Practices, Patterns*. This book debunks the myths around dependency injection (DI) and teaches you good practices and anti-patterns concerning DI while applying this to .NET Core. Our goal is to let this book be as influential as the first edition has been.

Although many tools help us with DI, DI is foremost a set of principles and patterns. Tools are useful, but optional. This message is the common theme throughout this book, since, when you have mastered the patterns and principles behind DI, the correct usage of tools becomes an implementation detail. While tools can make working with DI easier, they won't compensate for bad design.

The book is divided into four parts.

Part 1 flies through the basics of Dependency Injection. Chapter 1 covers the "what, why, and how" of DI. After that, chapters 2 and 3 walk through building a simple .NET Core web application. We start by showing how easy it is to accidentally write tightly coupled code, after which, in chapter 3, we'll rewrite the application from scratch, but now using proper DI techniques.

Part 2 focuses on patterns, anti-patterns, and code smells. Chapter 4 dives into the most common patterns Composition Root and Constructor Injection, and the more specialized patterns Method Injection and Property Injection. In chapter 5, we discuss very common anti-patterns, like Service Locator and Ambien Context, while chapter 6 focuses on common code smells and explains how to refactor your code.

Part 3 focuses on the three pillars of DI: Application Composition, Object Lifetime Management, and Interception. We dedicated a chapter to each pillar.

In parts 1 to 3, our discussion is mainly tool agnostic. After part 3, you should be able to design complete loosely coupled applications using pure DI; that is, DI without a tool. In part 4 we dive into the realm of DI Containers and explain how and when you should use these tools, and perhaps even more importantly, when you shouldn't use them. We discuss two commonly used DI Containers; since this book is about .NET Core, we discuss the DI Container that Microsoft built into ASP.NET Core as well.

We hope you find our book useful to read and hope it will guide you towards more maintainable software that is more fun to write and maintain.

—Mark and Steven

brief contents

Part I
Putting Dependency Injection on the map

Dependency Injection (DI) is one of the most misunderstood concepts of object-oriented programming. The confusion is abundant and spans terminology, purpose, and mechanics. Should it be called Dependency Injection, Dependency Inversion, Inversion of Control, or even Third-Party Connect? Is the purpose of DI only to support unit testing or is there a broader purpose? Is DI the same as **Service Location**? Do we need **DI Containers** to apply DI?

There are plenty of blog posts, magazine articles, conference presentations, and so on that discuss DI, but, unfortunately, many of them use conflicting terminology or give bad advice. This is true across the board, and even big and influential actors like Microsoft add to the confusion.

It doesn't have to be this way. In this book, we present and use a consistent terminology. For the most part, we've adopted and clarified existing terminology defined by others, but, occasionally, we add a bit of terminology where none existed previously. This has helped us tremendously in evolving a specification of the scope or boundaries of DI.

One of the underlying reasons behind all the inconsistency and bad advice is that the boundaries of DI are quite blurry. Where does DI end and other object-oriented concepts begin? We think that it's impossible to draw a distinct line between DI and other aspects of writing good object-oriented code. To talk about DI, we have to pull in other concepts such as **SOLID**, Clean Code, and even **Aspect-Oriented Programming**. We don't feel that we can credibly write about DI without also touching on some of these other topics.

The first part of the book helps you understand the place of DI in relation to other facets of software engineering—putting it on the map, so to speak. Chapter 1 gives you a quick tour of DI, covering its purpose, principles, and benefits, as well as providing an outline of the scope for the rest of the book. It's focused on the big picture and doesn't go into a lot of details. If you want to learn what DI is, and why you should be interested in it, this is the place to start. This chapter assumes you have no prior knowledge of DI. Even if you already know about DI, you may still want to read it—it may turn out to be something other than what you expected.

Chapters 2 and 3, on the other hand, are completely reserved for one big example. This example is intended to give you a much more concrete feel for DI. To contrast DI with a more traditional style of programming, chapter 2 showcases a typical, tightly coupled implementation of a sample e-commerce application. Chapter 3 then subsequently reimplements it with DI.

In this part, we discuss DI in general terms. This means we won't use any so-called **DI Container**. It's entirely possible to apply DI without using a **DI Container**. A **DI Container** is a helpful, but optional tool. So parts 1, 2, and 3 more or less ignore **DI Containers** completely, and instead discuss DI in a container-agnostic way. Then, in part 4, we return to **DI Containers** to dissect three specific libraries.

Part 1 establishes the context for the rest of the book. It's aimed at readers who don't have any prior knowledge of DI, but experienced DI practitioners can also benefit from skimming the chapters to get a feeling for the terminology used throughout the book. By the end of part 1, you should have a firm grasp of the vocabulary and overall concepts, even if some of the concrete details are still a little fuzzy. That's OK—the book becomes more concrete as you read on, so parts 2, 3, and 4 should answer the questions you're likely to have after reading part 1.

1 The Basics of Dependency Injection: What, Why and How

In this chapter:

- Common myths about Dependency Injection
- Purpose of Dependency Injection
- Benefits of Dependency Injection
- When to apply Dependency Injection

You may have heard that making a sauce béarnaise is difficult. Even among people who regularly cook, many have never attempted to make one. This is a shame, because the sauce is delicious. (It’s traditionally paired with steak, but it’s also an excellent accompaniment with white asparagus, poached eggs, and other dishes.) Some resort to substitutes like ready-made sauces or instant mixes, but these aren’t nearly as satisfying as the real thing.

A *sauce béarnaise* is an emulsified sauce made from egg yolk and butter that’s flavored with tarragon, chervil, shallots, and vinegar. It contains no water. The biggest challenge to making a sauce béarnaise is that its preparation can fail. The sauce can curdle or separate, and, if that happens, you can’t resurrect it. It takes about 45 mins to prepare, so a failed attempt means that you may not have time for a second try. On the other hand, any chef can prepare a sauce béarnaise. It’s part of their training and, as they’ll tell you, it’s not difficult.

You don’t have to be a professional cook to make sauce béarnaise. Anyone learning to make it will fail at least once, but once you get the hang of it, you’ll succeed every time. We think Dependency Injection (DI) is like sauce béarnaise. It’s assumed to be difficult, and, if you try to use it and fail, it’s likely there won’t be time for a second attempt.

DEFINITION ***Dependency Injection* is a set of software design principles and patterns that enable you to develop loosely coupled code.**

Despite the fear, uncertainty, and doubt (FUD) surrounding DI, it's as easy to learn as making a sauce béarnaise. You may make mistakes while you learn, but once you've mastered the technique, you'll never again fail to apply it successfully.

Stack Overflow, the software development Q&A website, features an answer to the question, *"How to explain Dependency Injection to a 5-year old?"* The most highly rated answer by John Munsch provides a surprisingly accurate analogy targeted at the (imaginary) five-year-old inquisitor:[1]

When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.

What you should be doing is stating a need, *"I need something to drink with lunch,"* and then we will make sure you have something when you sit down to eat.

What this means in terms of object-oriented software development is this: collaborating classes (the five-year-old) should rely on the infrastructure (the parents) to provide the necessary services.

NOTE **In DI terminology, we often talk about services and components. A *service* is typically an Abstraction, a definition for something that provides a *service*. An implementation of such Abstraction is often called a *component*, a class that contains behavior. Because both service and component are such overloaded terms, throughout this book, you'll typically see us use the terms Abstraction and *class* instead.**

This chapter is fairly linear in structure. First, we introduce DI, including its purpose and benefits. Although we include examples, overall, this chapter has less code than any other chapter in the book. Before we introduce DI, we'll discuss the basic purpose of DI—maintainability. This is important because it's easy to misunderstand DI if you aren't properly prepared. Next, after an example (Hello DI!), we'll discuss benefits and scope, laying out a road map for the book. When you're done with this chapter, you should be prepared for the more advanced concepts in the rest of the book.

To most developers, DI may seem like a rather backward way of creating source code, and, like sauce béarnaise, there's much FUD involved. To learn about DI, you must first understand its purpose.

1.1 Writing maintainable code

What purpose does DI serve? DI isn't a goal in itself; rather, it's a means to an end.

[1] See John Munsch et al., *"How to explain Dependency Injection to a 5-year old,"* 2009, stackoverflow.com/questions/1638919/.

Ultimately, the purpose of most programming techniques is to deliver working software as efficiently as possible. One aspect of that is to write maintainable code.

Unless you only write prototypes, or applications that never make it past their first release, you'll soon find yourself maintaining and extending existing code bases. To be able to work effectively with such code bases, in general, the more maintainable they are, the better.

An excellent way to make code more maintainable is through *loose coupling*. As far back as 1995, when the Gang of Four wrote *Design Patterns*, this was already common knowledge:[2]

Program to an interface, not an implementation

This important piece of advice isn't the conclusion, but, rather, the premise of *Design Patterns*. Loose coupling makes code extensible, and extensibility makes it maintainable. DI is nothing more than a technique that enables loose coupling. Moreover, there are many misconceptions about DI, and sometimes they get in the way of proper understanding. Before you can learn, you must *unlearn* what (you think) you already know.

1.1.1 Common myths about DI

You may never have come across or heard of DI before and that's great. Skip this section and go straight to section 1.1.2. But, if you're reading this book, it's likely you've at least come across it in conversation, in a code base you inherited, or in blog posts. You may also have noticed that it comes with a fair amount of heavy opinions. In this section, we're going to look at four of the most common misconceptions about DI that have appeared over the years and why they aren't true. These myths include:

- DI is only relevant for late binding.
- DI is only relevant for unit testing.
- DI is a sort of Abstract Factory on steroids.
- DI requires a **DI Container**.

Although none of these myths are true, they're prevalent nonetheless. We need to dispel them before you can start to learn about DI.

Late binding

In this context, *late binding* refers to the ability to replace parts of an application without recompiling the code. An application that enables third-party add-ins (such as Visual Studio) is one example. Another example is the standard software that supports different runtime environments.

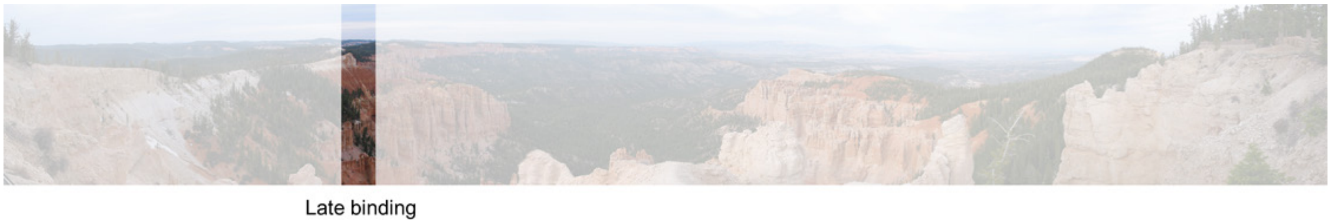
Suppose you have an application that runs on more than one database engine (for example, one that supports both Oracle and SQL Server). To support this feature, the

[2] See page 18, Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).

rest of the application talks to the database through an interface. The code base provides different implementations of this interface to access Oracle and SQL Server, respectively. In this case, you can use a configuration option to control which implementation should be used for a given installation.

It's a common misconception that DI is only relevant for this sort of scenario. That's understandable, because DI enables this scenario. But the fallacy is to think that the relationship is symmetric. Because DI enables late binding doesn't mean that it's only relevant in late-binding scenarios. As figure 1.1 illustrates, late binding is only one of the many aspects of DI.

Figure 1.1. Late binding is enabled by DI, but to assume that it's only applicable in late-binding scenarios is to adopt a narrow view of a much broader vista.



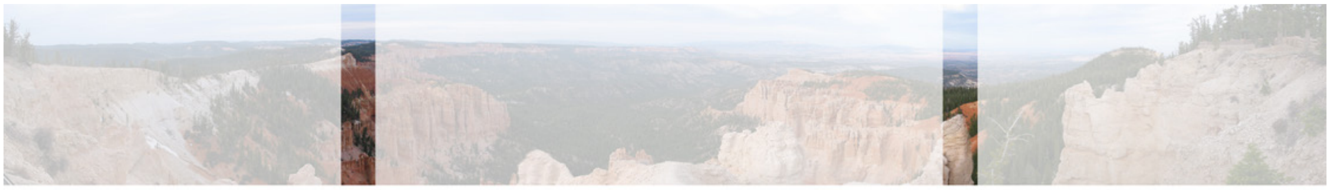
If you thought that DI was only relevant for late-binding scenarios, this is something you need to unlearn. DI does much more than enable late binding.

Unit testing

Some people think that DI is only relevant for supporting unit testing. This isn't true, either, although DI is certainly an important part of support for unit testing. To tell you the truth, our original introduction to DI came from struggling with certain aspects of Test-Driven Development (TDD). During that time, we discovered DI and learned that other people had used it to support some of the same scenarios we were addressing.

Even if you don't write unit tests (if you don't, you should start now), DI is still relevant because of all the other benefits it offers. Claiming that DI is only relevant when supporting unit testing is like claiming that it's only relevant for supporting late binding. Figure 1.2 shows that although this is a different view, it's a view as narrow as figure 1.1. In this book, we'll do our best to show you the whole picture.

Figure 1.2. Perhaps you've been assuming that unit testing is the sole purpose of DI. Although that assumption is a different view than the late binding assumption, it, too, is a narrow view of a much broader vista.



If you thought that DI was only relevant for unit testing, unlearn this assumption. DI does much more than enable unit testing.

An Abstract Factory on steroids

Perhaps the most dangerous fallacy is that DI involves some sort of general-purpose Abstract Factory that you can use to create instances of the **Dependencies** needed in your applications.

Abstract Factory

An *Abstract Factory* is typically an **Abstraction** that contains multiple methods, where each method allows the creation of an object of a certain kind.[3]

A typical use case for the Abstract Factory pattern is for user interface (UI) toolkits or client applications that must be run on multiple platforms. To achieve a high degree of code reusability on all platforms, you could, for example, define an `IUIControlFactory` **Abstraction** that allows the creation of certain kinds of controls like text boxes and buttons for consumers:

```
public interface IUIControlFactory
{
    IButton CreateButton();
    ITextBox CreateTextBox();
}
```

For each operating system (OS), you could have a different implementation of this `IUIControlFactory`. In this case, there are only two factory methods, but depending on the application or toolkit, there could be many more. An important point to note is that an Abstract Factory specifies a predefined list of factory methods.

In the introduction to this chapter, we wrote that *"collaborating classes…should rely on the infrastructure…to provide the necessary services."* What were your initial thoughts about this sentence? Did you think about the infrastructure as some sort of service you could query to get the **Dependencies** you need? If so, you aren't alone. Many developers and architects think about DI as a service that can be used to locate other services. This is called a **Service Locator**, but it's the exact opposite of DI.

[3] See page 87, Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).

A **Service Locator** is often called an Abstract Factory on steroids because, compared to a normal Abstract Factory, the list of resolvable types is unspecified and possibly endless. It typically has one method allowing the creation of all sorts of types, much like in the following:

```
public interface IServiceLocator
{
    object GetService(Type serviceType);
}
```

IMPORTANT **If you thought of DI as a Service Locator (that is, a general-purpose factory), then this is something you need to unlearn. DI is the opposite of a Service Locator; it's a way to structure code so that you never have to imperatively ask for Dependencies. Rather, you require consumers to supply them.**

DI Containers

Closely associated with the previous misconception is the notion that DI requires a **DI Container**. If you held the previous, mistaken belief that DI involves a **Service Locator**, then it's easy to conclude that a **DI Container** can take on the responsibility of the **Service Locator**. This might be the case, but it's not at all how you should use a **DI Container**.

A **DI Container** is an optional library that makes it easier to compose components when you wire up an application, but it's in no way required. When you compose applications without a **DI Container**, it's called **Pure DI**. It might take a little more work, but other than that, you don't have to compromise on any DI principles.[4]

IMPORTANT **If you thought that DI requires a DI Container, this is another notion you need to unlearn. DI is a set of principles and patterns, and a DI Container is a useful, but optional tool.**

We've yet to explain what a **DI Container** is exactly and how and when you should use it. We'll go into more detail on this at the end of chapter 3; part 4 is completely dedicated to it.

You may think that, although we've exposed four myths about DI, we have yet to make a compelling case against any of them. That's true. In a sense, this book is one big argument against these common misconceptions, so we'll certainly be returning to these topics later. For example, section 5.2 discusses why **Service Locator** is an anti-pattern.

In our experience, unlearning is vital because people often try to retrofit what we tell them about DI and align it with what they think they already know. When this happens, it takes time before it finally dawns on them that some of their most basic assumptions are wrong. We want to spare you that experience. If you can, read this book as though

[4] The first edition of this book, *Dependency Injection in .NET*, uses the term *Poor Man's DI*. **Pure DI** replaces this term, but don't be supprised to see the old terminology on the internet. To learn more about why we changed this terminology, see Mark Seemann, *"Pure DI,"*2014, blog.ploeh.dk/2014/06/10/pure-di/.

you know nothing about DI.

1.1.2 Understanding the purpose of DI

DI isn't an end-goal—it's a means to an end. DI enables loose coupling, and loose coupling makes code more maintainable. That's quite a claim, and although we could refer you to well-established authorities like the Gang of Four for details, we find it only fair to explain why this is true.

To get this message across, the next section compares software design and several software design patterns with electrical wiring. We found this to be a powerful analogy. We even use it to explain software design to non-technical people.

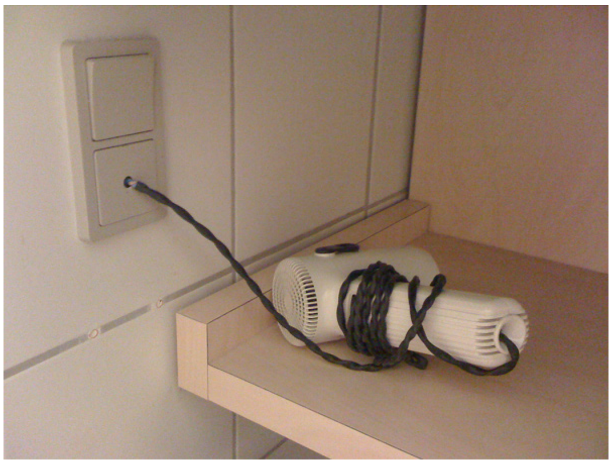
We use four specific design patterns in this analogy because they occur frequently in relation to DI. You'll see many examples of three of those patterns—Decorator, Composite, and Adapter—throughout this book. (We cover the fourth, the Null Object pattern, in chapter 4.) Don't worry if you're not that familiar with these patterns, you will be by the end of the book.

Software development is still a rather new profession, so in many ways we're still figuring out how to implement good architecture. But individuals with expertise in more traditional professions (such as construction) figured it out a long time ago.

Checking into a cheap hotel

If you're staying at a cheap hotel, you might encounter a sight like the one in figure 1.3. Here, the hotel has kindly provided a hair dryer for your convenience, but apparently they don't trust you to leave the hair dryer for the next guest: the appliance is directly attached into the wall outlet. The hotel management decided that the cost of replacing stolen hair dryers is high enough to justify what's otherwise an obviously inferior implementation.

Figure 1.3. In a cheap hotel room, you might find a hair dryer wired directly into the wall outlet. This is equivalent to using the common practice of writing tightly coupled code.



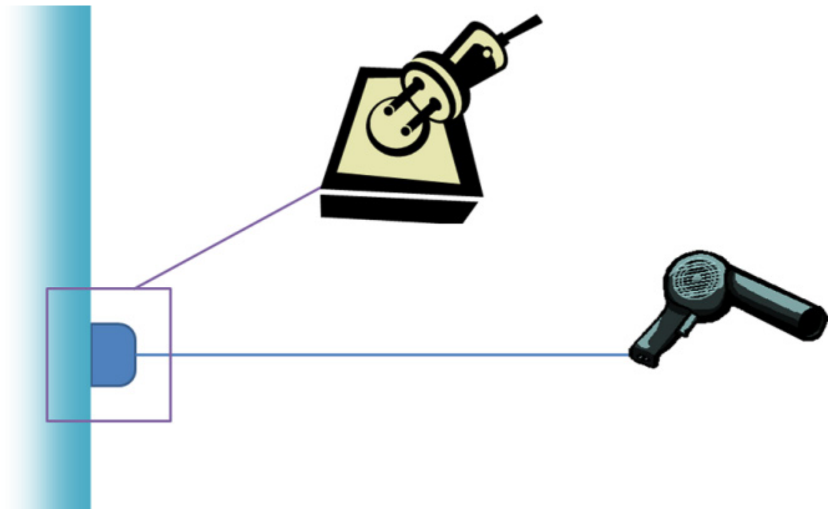
What happens when the hair dryer stops working? The hotel has to call in a skilled professional. To fix the hardwired hair dryer, the power to the room will have to be cut, rendering it temporarily useless. Then, the technician must use special tools to disconnect the hair dryer and replace it with a new one. If you're lucky, the technician will remember to turn the power to the room back on and go back to test whether the new hair dryer works…if you're lucky. Does this procedure sound at all familiar?

This is how you would approach working with tightly coupled code. In this scenario, the hair dryer is tightly coupled to the wall, and you can't easily modify one without impacting the other.

Comparing electrical wiring to design patterns

Usually, we don't wire electrical appliances together by attaching the cable directly to the wall. Instead, as in figure 1.4, we use plugs and sockets. A socket defines a shape that the plug must match.

Figure 1.4. Through the use of sockets and plugs, a hair dryer can be loosely coupled to the wall outlet.

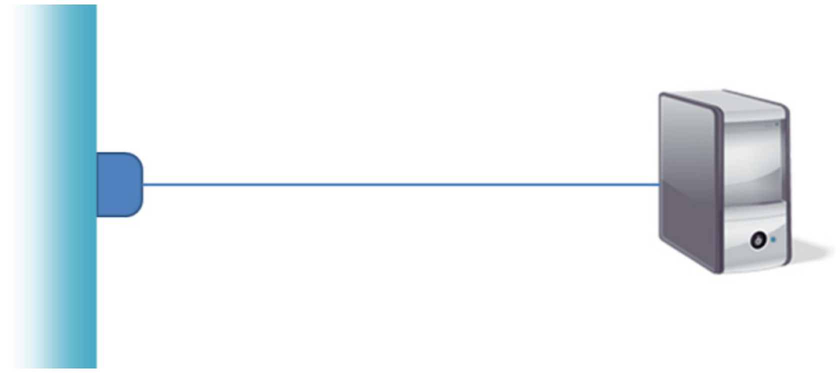


In an analogy to software design, the socket is an interface, and the plug with its appliance is an implementation. This means that the room (the application) has one or (hopefully) more sockets, and the users of the room (the developers) can plug in appliances as they please, potentially even a customer-supplied hair dryer.

In contrast to the hardwired hair dryer, plugs and sockets define a loosely coupled model for connecting electrical appliances. As long as the plug (the implementation) fits into the socket (implements the interface), and it can handle the amount of volts and hertz (obeys the interface contract), we can combine appliances in a variety of ways. What's particularly interesting is that many of these common combinations can be compared to well-known software design principles and patterns.

First, we're no longer constrained to hair dryers. If you're an average reader, we would guess that you need power for a computer much more than you do for a hair dryer. That's not a problem: you unplug the hair dryer and plug a computer into the same socket (figure 1.5).

Figure 1.5. Using sockets and plugs, you can replace the original hair dryer from figure 1.4 with a computer. This corresponds to the Liskov Substitution Principle.



Liskov Substitution Principle

It's amazing that the concept of a socket predates computers by decades, and yet it provides an essential service to computers. The original designers of sockets couldn't possibly have foreseen personal computers, but because the design is so versatile, needs that were originally unanticipated can be met.

The ability to replace one end without changing the other is similar to a central software design principle called the **Liskov Substitution Principle**. This principle states that we should be able to replace one implementation of an interface with another without breaking either the client or the implementation.

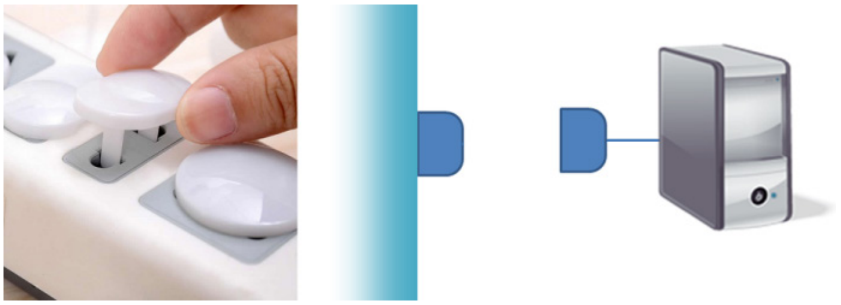
When it comes to DI, the **Liskov Substitution Principle** is one of the most important software design principles. It's this principle that enables us to address requirements that occur in the future, even if we can't foresee them today.

We can unplug the computer if we don't need to use it at the moment. Even though nothing is plugged in, the room doesn't explode. That's to say, if we unplug the computer from the wall, neither the wall outlet nor the computer breaks down.

With software, however, a client often expects a service to be available. If you remove the service, you get a `NullReferenceException`. To deal with this type of situation, you can create an implementation of an interface that does nothing. This design pattern, known as *Null Object*, corresponds to having a children's safety outlet plug (a plug without a wire or appliance that still fits into the socket).[5] And because you're using loose coupling, you can replace a real implementation with something that does nothing without causing trouble. This is illustrated in figure 1.6.

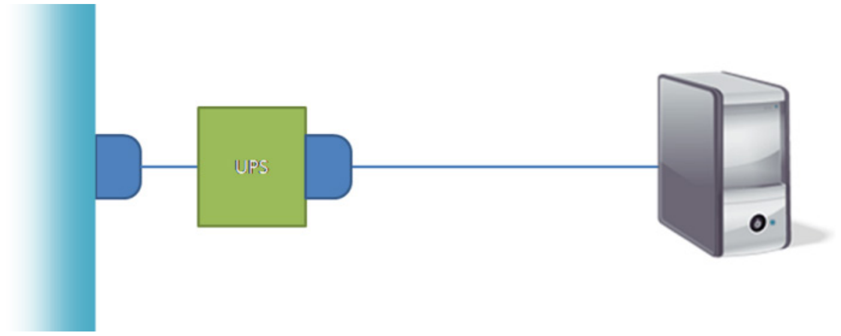
[5] To learn more about the Null Object pattern, see page 5, Robert C. Martin, et al. *Pattern Languages of Program Design 3* (Addison-Wesley, 1998).

Figure 1.6. Unplugging the computer causes neither room nor computer to explode when replaced with a children's safety outlet plug. This can be roughly likened to the Null Object pattern.



As well, there are many other things we can do. If we live in a neighborhood with intermittent power failures, we may want to keep the computer running by plugging in into an Uninterrupted Power Supply (UPS). As shown in figure 1.7, we connect the UPS to the wall outlet and the computer to the UPS.

Figure 1.7. An UPS can be introduced to keep the computer running in case of power failures. This corresponds to the Decorator design pattern.



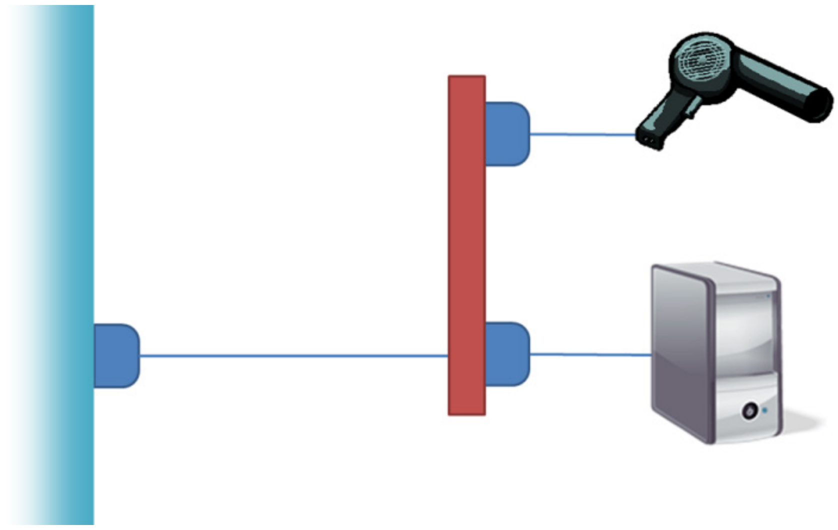
The computer and the UPS serve separate purposes. Each has a **Single Responsibility** that doesn't infringe on the other appliance. The UPS and computer are likely to be produced by two different manufacturers, bought at different times, and plugged in separately. As figure 1.5 demonstrates, you can run the computer without a UPS, and you could also conceivably use the hair dryer during blackouts by plugging it into the UPS.

In software design, this way of **Intercepting** one implementation with another implementation of the same interface is known as the *Decorator design pattern*.[6] It gives you the ability to incrementally introduce new features and **Cross-Cutting Concerns** without having to rewrite or change a lot of existing code.

[6] See page 175, Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).

Another way to add new functionality to an existing code base is to refractor an existing implementation of an interface with a new implementation. When you aggregate several implementations into one, you use the *Composite design pattern*.[7] Figure 1.8 illustrates how this corresponds to plugging diverse appliances into a power strip.

Figure 1.8. A power strip makes it possible to plug several appliances into a single wall outlet. This corresponds to the Composite design pattern.

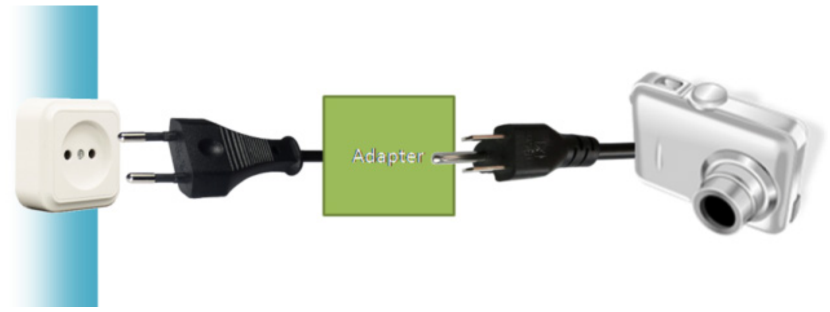


The power strip has a single plug that we can insert into a single socket, and the power strip itself provides several sockets for a variety of appliances. This enables us to add and remove the hair dryer while the computer is running. In the same way, the Composite pattern makes it easy to add or remove functionality by modifying the set of composed interface implementations.

Here's a final example. You sometimes find yourself in situations where a plug doesn't fit into a particular socket. If you've traveled to another country, you've likely noticed that sockets differ across the world. If you bring something like the camera in figure 1.9 along when traveling, you'll need an adapter to charge it. Appropriately, there's a design pattern with the same name.

[7] See page 163, Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).

Figure 1.9. When traveling, we often need to use an adapter to plug an appliance into a foreign socket (for example, to recharge a camera). This corresponds to the *Adapter design pattern*. Sometimes, translation is as simple as changing the shape of the plug, or as complex as changing the electric current from alternating current (AC) to direct current (DC).



The Adapter design pattern works like its physical namesake.[8] You can use it to match two related, yet separate, interfaces to each other. This is particularly useful when you've an existing third-party API that you want to expose as an instance of an interface your application consumes. As with the physical adapter, implementations of the Adapter design pattern can range from simple to extremely complex.

What's amazing about the socket and plug model is that, over decades, it's proven to be an easy and versatile model. Once the infrastructure is in place, it can be used by anyone and adapted to changing needs and unanticipated requirements. What's even more interesting is that, when we relate this model to software development, all the building blocks are already in place in the form of design principles and patterns.

The advantage of loose coupling is the same in software design as it's our physical socket and plug model: Once the infrastructure is in place, it can be used by anyone and adapted to changing needs and unforeseen requirements without having to make large changes to the application's code base and its infrastructure. This means that ideally, a new requirement should only necessitate the addition of a new class, with no changes to other already existing classes of the system.

This concept of being able to extend the application without modifying existing code is called the **Open/Closed Principle**. It's impossible to get to a situation where 100% of your code will always be *open* for extensibility and *closed* for modification. Still, loose coupling does bring you closer to that goal.

And, with every step, it gets easier to add new features and requirements to your system. Being able to add new features without touching existing parts of the system means that problems are isolated. This leads to code that's easier to understand and test, allowing you to manage the complexity of your system. That's what loose coupling can help you with, and that's why it can make a code base much more maintainable. We'll discuss the **Open/Closed Principle** in more detail in chapter 4.

[8] See page 139, Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).

By now you might be wondering how these patterns will look when implemented in code. Don't worry about that. As we stated before, we'll show you plenty of examples of those patterns throughout this book. In fact, later in this chapter, we'll show you an implementation of both the Decorator and Adapter patterns.

The easy part of loose coupling is programming to an interface instead of an implementation. The question is, *"Where do the instances come from?"* In a sense, this is what this entire book is about: it's the core question that DI seeks to answer.

You can't create a new instance of an interface the same way that you create a new instance of a concrete type. Code like this just doesn't compile:

```
IMessageWriter writer =         ❶
    new IMessageWriter();       ❷
```

❶ Program to an interface
❷ Doesn't compile

An interface contains no implementation, so this isn't possible. The `writer` instance must be created using a different mechanism. DI solves this problem. With this outline of the purpose of DI, we think you're ready for an example.

1.2 A simple example: Hello DI!

In the tradition of innumerable programming textbooks, let's take a look at a simple console application that writes *"Hello DI!"* to the screen. Note that the full code is available as part of the download for this book.

In this section, we'll show you what the code looks like and briefly outline some key benefits without going into details. In the rest of the book, we'll get more specific.

1.2.1 Hello DI! code

You're probably used to seeing Hello World examples that are written in a single line of code. Here, we'll take something that's extremely simple and make it more complicated. Why? We'll get to that shortly, but let's first see what Hello World would look like with DI.

Collaborators

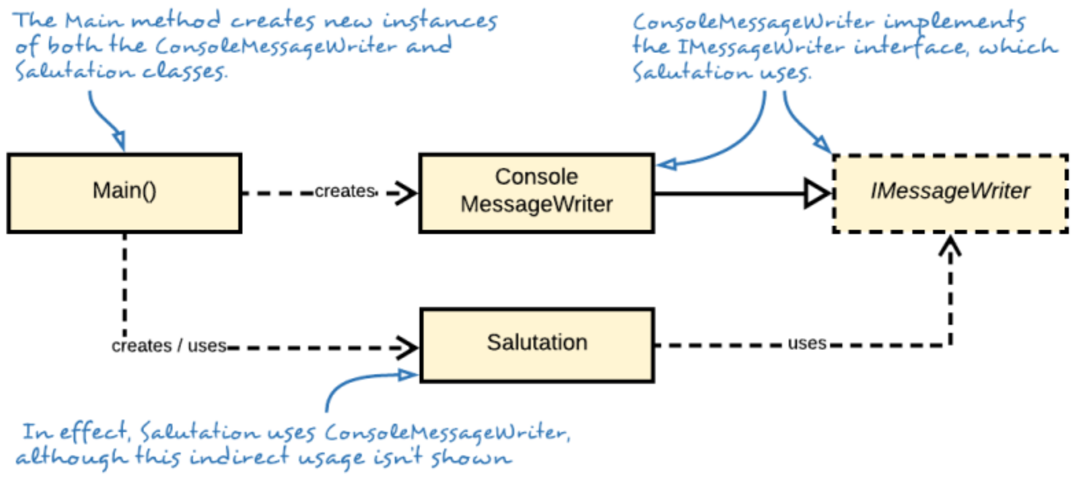
To get a sense of the structure of the program, we'll start by looking at the `Main` method of the console application. Then we'll show you the collaborating classes, but first, here's the `Main`method of the Hello DI! application:

```
private static void Main()
{
    IMessageWriter writer = new ConsoleMessageWriter();
    var salutation = new Salutation(writer);
    salutation.Exclaim();
}
```

Because the program needs to write to the console, it creates a new instance of `ConsoleMessageWriter` that encapsulates that functionality. It passes that message writer to the `Salutation` class so that the salutation instance knows where to write its messages. Because everything is now wired up properly, you can execute the logic via the `Exclaim` method, which results in the message being written to the screen.

The construction of objects inside the `Main` method is a basic example of **Pure DI**. No **DI Container** is used to compose the `Salutation` and its `ConsoleMessageWriter` **Dependency**. Figure 1.10 shows the relationship between the collaborators.

Figure 1.10. Relationship between the collaborators of the Hello DI! application


The Main method creates new instances of both the ConsoleMessageWriter and Salutation classes.
ConsoleMessageWriter implements the IMessageWriter interface, which Salutation uses.
Main()
creates
Console MessageWriter
IMessageWriter
creates / uses
Salutation
uses
In effect, Salutation uses ConsoleMessageWriter, although this indirect usage isn't shown

Implementing the application's logic

The main logic of the application is encapsulated in the `Salutation` class, shown in the following listing.

Listing 1.1. Salutation class encapsulates the main application's logic.

```
public class Salutation
{
    private readonly IMessageWriter writer;

    public Salutation(IMessageWriter writer)           ❶
    {
        if (writer == null)                            ❷
            throw new ArgumentNullException("writer"); ❷

        this.writer = writer;
    }

    public void Exclaim()
    {
        this.writer.Write("Hello DI!");                  ❸
    }
}
```

❶ Provides the Salutation class with the IMessageWriter **Dependency** using **Constructor Injection**.
❷ Guard Clause verifies that the supplied IMessageWriter isn't null.
❸ Sends the Hello DI! message to the IMessageWriter **Dependency**.

The `Salutation` class depends on a custom interface called `IMessageWriter` (defined next). It requests an instance of it through its constructor. This practice is called **Constructor Injection**. A *Guard Clause* verifies that the supplied `IMessageWriter` isn't null by throwing an exception if it is.[9] And, finally, you use the previously injected `IMessageWriter` instance inside the implementation of the `Exclaim` method by calling its `Write` method. This sends the Hello DI! message to the `IMessageWriter` **Dependency**.

DEFINITION **Constructor Injection is the act of statically defining the list of required Dependencies by specifying them as parameters to the class's constructor. (Constructor Injection is described in detail in chapter 4, which also contains a more detailed walk-through of a similar code example.)**

To speak in DI terminology, we say that the `IMessageWriter` **Dependency** is injected into the `Salutation` class using a constructor argument. Note that `Salutation` has no awareness of the `ConsoleMessageWriter`. It interacts with it exclusively through the `IMessageWriter` interface. `IMessageWriter` is a simple interface defined for the occasion:

```
public interface IMessageWriter
{
    void Write(string message);
}
```

It could have had other members, but in this simple example, you only need the `Write` method. It's implemented by the `ConsoleMessageWriter` class that the `Main` method passes to the `Salutation`class:

```
public class ConsoleMessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine(message);
    }
}
```

The `ConsoleMessageWriter` class implements `IMessageWriter` by wrapping the .NET Base Class Library's `Console` class. This is a simple application of the Adapter design pattern that we talked about in section 1.1.2.

[9] See page 250, Martin Fowler et al., *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999).

1.2.2 Benefits of DI

You may be wondering about the benefit of replacing a single line of code with two classes and an interface, resulting in a total 11 lines. You could easily solve the same problem as shown here:

```
private static void Main()
{
    Console.WriteLine("Hello DI!");
}
```

DI might seem like overkill, but there are several benefits to be harvested from doing this. How is the previous example better than the usual single line of code you normally use to implement Hello World in C#? In this example, DI adds an overhead of 1100%, but, as complexity increases from one line of code to tens of thousands, this overhead diminishes and all but disappears. Chapter 3 provides a more complex example of applied DI. Although that example is still overly simplistic compared to real-life applications, you should notice that DI is far less intrusive.

We don't blame you if you find the previous DI example to be over-engineered, but consider this: by its nature, the classic Hello World example is a simple problem with well-specified and constrained requirements. In the real world, software development is never like this. Requirements change and are often fuzzy. The features you must implement also tend to be much more complex. DI helps address such issues by enabling loose coupling. Specifically, you gain the benefits listed in table 1.1.

Table 1.1. Benefits gained from loose coupling. Each benefit is always available but will be valued differently depending on circumstances.

Benefit	Description	When is it valuable?
Late binding	Services can be swapped with other services without recompiling code.	Valuable in standard software, but perhaps less so in enterprise applications where the runtime environment tends to be well-defined
Extensibility	Code can be extended and reused in ways not explicitly planned for.	Always valuable
Parallel development	Code can be developed in parallel.	Valuable in large, complex applications; not so much in small, simple applications
Maintainability	Classes with clearly defined responsibilities are easier to maintain.	Always valuable
Testability	Classes can be unit tested.	Always valuable

In table 1.1, we listed the late binding benefit first because, in our experience, this is the one that's foremost in most people's minds. When architects and developers fail to understand the benefits of loose coupling, it's most likely because they never consider the other benefits.

Late binding

When we explain the benefits of programming to interfaces and DI, the ability to swap out one service with another is the most prevalent benefit for most people, so they tend to weigh the advantages against the disadvantages with only this benefit in mind. Remember when we suggested that you may need to unlearn before you can learn? You may say that you know your requirements so well that you know you'll never have to replace, say, your SQL Server database with anything else. But requirements change.

NoSQL, Microsoft Azure, and the argument for composability

Years ago, I (Mark) was often met with a blank expression when I tried to convince developers and architects of the benefits of DI. "Okay, so you can swap out your relational data access component for something else. For what? Is there any alternative to relational databases?"

XML files never seemed like a convincing alternative in highly scalable enterprise scenarios. This has changed significantly in the last couple of years.

Azure was announced at PDC 2008 and has done much to convince even die-hard Microsoft-only organizations to reevaluate their position when it comes to data storage. There's now a real alternative to relational databases, and I only have to ask if people want their application to be cloud-ready. The replacement argument has now become much stronger.

A related movement can be found in the whole NoSQL concept that models applications around denormalized data—often document databases. But concepts such as Event Sourcing are also becoming increasingly important.[10]

In section 1.2.1, you didn't use late binding because you explicitly created a new instance of `IMessageWriter` by hard-coding the creation of a new `ConsoleMessageWriter` instance. You can, however, introduce late binding by changing this single line of code:

```
IMessageWriter writer = new ConsoleMessageWriter();
```

To enable late binding, you might replace that line of code with something like this:

Listing 1.2. Late binding an `IMessageWriter` implementation

```
IConfigurationRoot configuration = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json")
    .Build();

string typeName = configuration["messageWriter"];
Type type = Type.GetType(typeName, throwOnError: true);

IMessageWriter writer = (IMessageWriter)Activator.CreateInstance(type);
```

[10] Martin Fowler, *"Event Sourcing,"* 2005, martinfowler.com/eaaDev/EventSourcing.html.

NOTE This example takes some shortcuts to make a point. In fact, it suffers from the **Constrained Construction** anti-pattern covered in detail in chapter 5.

By pulling the type name from the application configuration file and creating a `Type` instance from it, you can use reflection to create an instance of `IMessageWriter` without knowing the concrete type at compile time. To make this work, you specify the type name in the `messageWriter` application setting in the application configuration file:

```
{
  "messageWriter":
    "Ploeh.Samples.HelloDI.Console.ConsoleMessageWriter, HelloDI.Console"
}
```

Loose coupling enables late binding because there's only a single place where you create the instance of the `IMessageWriter`. Because the `Salutation` class works exclusively against the `IMessageWriter` interface, it never notices the difference. In the Hello DI! example, late binding would enable you to write the message to a different destination than the console; for example, a database or a file. It's possible to add such features—even though you didn't explicitly plan ahead for them.

Extensibility

Successful software must be able to change. You'll need to add new features and extend existing features. Loose coupling lets you efficiently recompose the application, similar to the way that you can rewire electrical appliances using plugs and sockets.

Let's say that you want to make the Hello DI! example more secure by only allowing authenticated users to write the message. Listing 1.3 shows how you can add that feature without changing any of the existing features—you simply add a new implementation of the `IMessageWriter` interface.

Listing 1.3. Extending the Hello DI! application with a security feature

```
public class SecureMessageWriter : IMessageWriter         ❶
{
    private readonly IMessageWriter writer;
    private readonly IIdentity identity;

    public SecureMessageWriter(
        IMessageWriter writer,                              ❷
        IIdentity identity)
    {
        if (writer == null)
            throw new ArgumentNullException("writer");
        if (identity == null)
            throw new ArgumentNullException("identity");

        this.writer = writer;
        this.identity = identity;
    }

    public void Write(string message)
    {
        if (this.identity.IsAuthenticated)            ❸
        {
            this.writer.Write(message);               ❹
        }
    }
}
```

❶ Implements the IMessageWriter interface while also consuming it
❷ **Constructor Injection** that requests an instance of IMessageWriter
❸ Verifies whether the user is authenticated
❹ If authenticated, writes the message using the injected message writer

NOTE **This is a standard application of the Decorator design pattern that we mentioned in section 1.1.2. We'll talk much more about Decorators in chapter 9.**

Besides an instance of `IMessageWriter`, the `SecureMessageWriter` constructor requires an instance of `IIdentity`. The `Write` method is implemented by first checking whether the current user is authenticated using the injected `IIdentity`. If this is the case, it allows the decorated writer field to `Write` the message. The only place where you need to change existing code is in the `Main`method because you need to compose the available classes differently than before:

```
IMessageWriter writer =
    new SecureMessageWriter(                    ❶
        new ConsoleMessageWriter(),
        WindowsIdentity.GetCurrent());
```

❶ The ConsoleMessageWriter is intercepted with the SecureMessageWriter Decorator.

NOTE **Compared to listing 1.2, you now use a hard-coded** `ConsoleMessageWriter`**.**

Notice that you wrap or *decorate* the old `ConsoleMessageWriter` instance with the new `SecureMessageWriter` class. Once more, the `Salutation` class is unmodified because it only consumes the `IMessageWriter` interface. Similarly, there's no need to either modify or duplicate the functionality in the `ConsoleWriter` class either. You use the `System.Security.Principal.WindowsIdentity`class to retrieve the identity of the user on whose behalf this code is being executed.[11]

As we've stated before, loose coupling enables you to write code that is *open for extensibility, but closed for modification*. The only place where you need to modify the code is at the application's entry point. The `SecureMessageWriter` implements the

[11] The `System.Security.Principal.WindowsIdentity` class is located in the System.Security.Principal.Windows NuGet package, which is part of .NET Core.

security features of the application, whereas the `ConsoleMessageWriter` addresses the user interface. This enables you to vary these aspects independently of each other and compose them as needed. Each class has its own **Single Responsibility**.

Parallel development

Separation of concerns makes it possible to develop code in parallel. When a software development project grows to a certain size, it becomes necessary to have multiple developers work in parallel on the same code base. At a larger scale, it'll even become necessary to separate the development team into multiple teams of manageable sizes. Each team is often assigned responsibility for an area of the overall application. To demarcate responsibilities, each team develops one or more modules that will need to be integrated into the finished application. Unless the areas of each team are truly independent, some teams are likely to depend on the functionality developed by other teams.

DEFINITION In object-oriented software design, a *module* is a group of logically related classes (or components), where a module is independent and interchangeable to other modules. Typically, you'll see that a *layer* consists of one or more modules.

In the previous example, because the `SecureMessageWriter` and `ConsoleMessageWriter` classes don't depend directly on each other, they could've been developed by parallel teams. All they would have needed to agree on was the shared interface `IMessageWriter`.

Maintainability

As the responsibility of each class becomes clearly defined and constrained, maintenance of the overall application becomes easier. This is a consequence of the **Single Responsibility Principle**, which states that each class should have only a single responsibility. We'll discuss the **Single Responsibility Principle** in more detail in chapter 2.

Adding new features to an application becomes simpler because it's clear where changes should be applied. More often than not, you don't need to change existing code, but can instead add new classes and recompose the application. This is the **Open/Closed Principle** in action again.

Troubleshooting also tends to become less grueling, because the scope of likely culprits narrows. With clearly defined responsibilities, you'll often have a good idea of where to start looking for the root cause of a problem.

Testability

An application is considered **Testable** when it can be unit tested. For some, **Testability** is the least of their worries; for others, it's an absolute requirement. Personally, we belong in the latter category. In Mark's career, he's declined several job offers because they involved working with certain products that weren't **Testable**.

Testability

The term **Testable** is horribly imprecise, yet it's widely used in the software development community, chiefly by those who practice unit testing. In principle, any application can be tested by trying it out. Tests can be performed by people using the application via its UI or whatever other interface it provides. Such manual tests are time-consuming and expensive to perform, so automated testing is preferred.

You'll find different types of automated testing—unit testing, integration testing, performance testing, stress testing, and so on. Because unit testing has fewer requirements on runtime environments, it tends to be the most efficient and robust type of test. It's often in this context that **Testability** is evaluated.

Unit tests provide rapid feedback on the state of an application, but it's only possible to write unit tests when the unit in question can be properly isolated from its **Dependencies**. There's some ambiguity about how granular a unit really is, but everyone agrees that it's certainly not something that spans multiple modules. The ability to test modules in isolation is crucial in unit testing.

It's only when an application is susceptible to unit testing that it's considered **Testable**. The safest way to ensure **Testability** is to develop it using Test-Driven Development (TDD).

It should be noted that unit tests alone don't ensure a working application. Full system tests or other in-between types of tests are still necessary to validate whether an application works as intended.

The benefit of **Testability** is perhaps the most controversial of those we've listed. Some developers and architects still don't practice unit testing, so they consider this benefit irrelevant at best. We, however, see it has an essential part of software development, which is why we marked it as *"Always valuable"* in table 1.1. Michael Feathers even defines the term *legacy application* as any application that isn't covered by unit tests.[12]

Almost by accident, loose coupling enables unit testing because consumers follow the **Liskov Substitution Principle**: they don't care about the concrete types of their **Dependencies**. This means that you can inject Test Doubles into the System Under Test (SUT), as you'll see in listing 1.4.

Test Doubles

It's a common technique to create implementations of **Dependencies** that act as stand-ins for the real or intended implementations. Such implementations are called *Test Doubles*, and they'll never be used in the final application. Instead, they serve as placeholders for the real **Dependencies** when these are unavailable or undesirable to use.

Test Doubles are useful when the real **Dependency** is slow, expensive, destructive, or simply outside the scope of the current test. There's a complete pattern language around Test Doubles and many subtypes, such as Stubs, Mocks, and Fakes.[13]

The ability to replace the intended **Dependencies** with test-specific replacements is a by-product of loose coupling, but we chose to list it as a separate benefit because the derived value is different. Our personal experience is that DI is beneficial even during integration testing. Although integration tests typically communicate with real external

[12] Page xvi, Michael C. Feathers, *Working Effectively with Legacy Code* (Prentice Hall, 2004).
[13] Page 522, Gerard Meszaros, *xUnit Test Patterns: Refactoring Test Code* (Addison-Wesley, 2007).

systems (like a database), you'll still find the need to have a certain degree of isolation. In other words, there are still reasons to replace, **Intercept** or mock certain **Dependencies** in the application being tested.

Intercepting text messages

I (Steven) worked on multiple applications that sent SMS messages through a third-party service. I didn't want our test environment to send those text messages to the real gateway because there was a per-message cost, and I certainly didn't want to accidentally spam mobile phones with those test messages.

During manual testing, on the other hand, text messages were sent to mobile phones. But, in this case, a Decorator was applied that changed the phone number sent to the gateway to one that the tester could supply. This way the tester was able to get all messages on his own phone and verify the system under test.

Depending on the type of application you're developing, you may or may not care about the ability to do late binding, but we always care about **Testability**. Some developers don't care about **Testability** but find late binding important for the application they're developing. Regardless, DI provides options in the future with minimal additional overhead today.

Example: unit testing the "HelloDI" logic

In section 1.2.1, you saw the Hello DI! example. Although we showed you the final code first, we developed it using TDD. Listing 1.4 shows the most important unit test.

NOTE **Don't worry if you don't have experience with unit tests. They'll occasionally pop up throughout this book but are in no way a prerequisite for reading it.**[14]

Listing 1.4. Unit testing the Salutation class

```
[Fact]
public void ExclaimWillWriteCorrectMessageToMessageWriter()
{
    var writer = new SpyMessageWriter();
    var sut = new Salutation(writer);                    ❶
    sut.Exclaim();
    Assert.Equal(
        expected: "Hello DI!",
        actual: writer.WrittenMessage);
}

public class SpyMessageWriter : IMessageWriter
{
    public string WrittenMessage { get; private set; }

    public void Write(string message)
```

[14] You may, however, want to read Roy Osherove's *The Art of Unit Testing*, 2nd Ed. (Manning, 2013), followed by Gerard Meszaros' *xUnit Test Patterns* (Addison-Wesley, 2007).

```
    {
        this.WrittenMessage += message;
    }
}
```

(1) The IMessageWriter **Dependency** is stubbed using the SpyMessageWriter Test Spy.

The `Salutation` class needs an instance of the `IMessageWriter` interface, so you need to create one. You could use any implementation, but in unit tests, a Test Double can be useful—in this case, we roll our own Test Spy implementation.[15]

In this case, the Test Double is as involved as the production implementation. This is an artifact of how simple our example is. In most applications, a Test Double is significantly simpler than the concrete, production implementations it stands in for. The important part is to supply a test-specific implementation of `IMessageWriter` to ensure that you test only one thing at a time. Right now, you're testing the `Exclaim` method of the `Salutation` class, so you don't want a production implementation of `IMessageWriter` to pollute the test. To create the `Salutation` class, you pass in the Spy instance of `IMessageWriter` using **Constructor Injection**.

After exercising the SUT, you can call `Assert.Equal` to verify whether the expected outcome equals the actual outcome. If the `IMessageWriter.Write` method was invoked with the "Hello DI!" string, the `SpyMessageWriter` would have stored this in its `WrittenMessage` property, and the `Equal` method completes. But if the `Write` method wasn't called, or called with a different parameter, the `Equal` method would throw an exception and the test fails.

Loose coupling provides many benefits: code becomes easier to develop, maintain, and extend, and it becomes more **Testable**. It's not even particularly difficult. We program against interfaces, not concrete implementations. The only major obstacle is to figure out how to get hold of instances of those interfaces. DI surmounts this obstacle by injecting the **Dependencies** from the outside. **Constructor Injection** is the preferred method of doing that, though we'll also explore a few additional options in chapter 4.

1.3 What to inject and what not to inject

In the previous section, we described the motivational forces that makes one think about DI in the first place. If you're convinced that loose coupling is a benefit, you may want to make everything loosely coupled. Overall, that's a good idea. When you need to decide how to package modules, loose coupling proves especially useful. But you don't have to abstract everything away and make it pluggable. In this section, we'll provide some decision tools to help you decide how to model your **Dependencies**.

The .NET Base Class Library (BCL) consists of many assemblies. Every time you write code that uses a type from a BCL assembly, you add a dependency to your

[15] A Test Spy is *"a Test Double that captures the indirect output calls made to another component by the SUT for later verification by the test."* See Gerard Meszaros, *xUnit Test Patterns*, (Addison-Wesley, 2007), page 538.

module. In the previous section, we discussed how loose coupling is important and how programming to an interface is the cornerstone. Does this imply that you can't reference any BCL assemblies and use their types directly in your application? What if you'd like to use an `XmlWriter` that's defined in the System.Xml assembly?

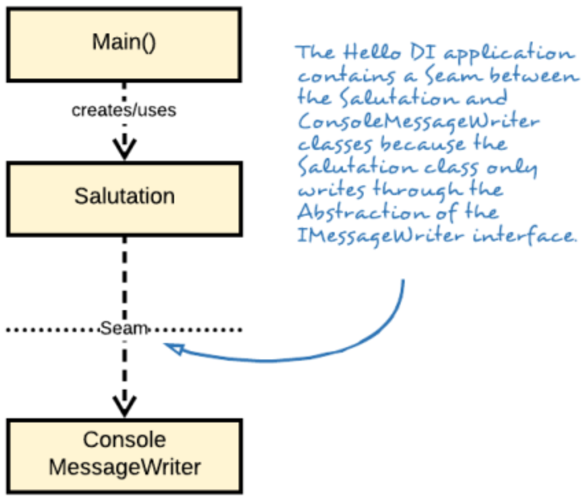
You don't have to treat all **Dependencies** equally. Many types in the BCL can be used without jeopardizing an application's degree of coupling—but not all of them. It's important to know how to distinguish between types that pose no danger and types that may tighten an application's degree of coupling. Focus mainly on the latter.

Seams

Everywhere you decide to program against an **Abstraction** instead of a concrete type, you introduce a **Seam** into the application. A **Seam** is a place where an application is assembled from its constituent parts, similar to the way a piece of clothing is sewn together at its seams.[16] It's also a place where you can disassemble the application and work with the modules in isolation.

The Hello DI! example we built in section 1.2 contains a **Seam** between `Salutation` and `ConsoleMessageWriter` as illustrated in figure 1.11. The `Salutation` class doesn't directly depend on the `ConsoleMessageWriter` class; rather, it uses the `IMessageWriter` interface to write the message. You can take the application apart at this **Seam** and reassemble it with a different message writer.

Figure 1.11. The Seam in the Hello DI! application from section 1.2.


Main()
creates/uses
Salutation
Seam
Console MessageWriter
The Hello DI application contains a Seam between the Salutation and ConsoleMessageWriter classes because the Salutation class only writes through the Abstraction of the IMessageWriter interface.

As you learn DI, it can be helpful to categorize your **Dependencies** into **Stable Dependencies** and **Volatile Dependencies**. Deciding where to put your **Seams** will soon become second nature to you. The next sections discuss these concepts in more detail.

[16] See pages 29-44, Michael C. Feathers, *Working Effectively with Legacy Code* (Prentice Hall, 2004).

1.3.1 *Stable Dependencies*

Many of the modules in the BCL and beyond pose no threat to an application's degree of modularity. They contain reusable functionality that you can use to make your own code more succinct. The BCL modules are always available to your application, because it needs the .NET Framework to run, and, because they already exist, the concern about parallel development doesn't apply to these modules. You can always reuse a BCL library in another application.

By default, you can consider most (but not all) types defined in the BCL as safe, or **Stable Dependencies**. We call them *stable* because they're already there, tend to be backward compatible, and invoking them has deterministic outcomes. Most **Stable Dependencies** are BCL types, but other **Dependencies** can be stable too. The important criteria for **Stable Dependencies** include the following:

- The class or module already exists.
- You expect that new versions won't contain breaking changes.
- The types in question contain deterministic algorithms.
- You never expect to have to replace, wrap, decorate, or intercept the class or module with another.

Other examples may include specialized libraries that encapsulate algorithms relevant to your application. For example, if you're developing an application that deals with chemistry, you can reference a third-party library that contains chemistry-specific functionality.

Referencing the DI Container

DI Containers themselves might be considered either **Stable Dependencies** or **Volatile Dependencies**, depending on whether you want to replace them. When you decide to base your application on a particular **DI Container**, you risk being stuck with this choice for the entire lifetime of the application. That's yet another reason why you should limit the use of the container to the application's entry point. Only the entry point should reference the **DI Container**.

In general, **Dependencies** can be considered stable by exclusion. They're stable if they aren't volatile.

1.3.2 *Volatile Dependencies*

Introducing **Seams** into an application is extra work, so you should only do it when it's necessary. There can be more than one reason it's necessary to isolate a **Dependency** behind a **Seam**, but those reasons are closely related to the benefits of loose coupling (discussed in section 1.2.1).

Such **Dependencies** can be recognized by their tendency to interfere with one or more of these benefits. They aren't stable because they don't provide a sufficient foundation for applications, and we call them **Volatile Dependencies** for that reason. A **Dependency** should be considered volatile if any of the following criteria are true:

- *The* **Dependency** *introduces a requirement to set up and configure a runtime environment for the application.* It isn't so much the concrete .NET types that are volatile, but rather what they imply about the runtime environment.

  Databases are good examples of BCL types that are **Volatile Dependencies**, and relational databases are the archetypical example. If you don't hide the relational database behind a **Seam**, you can never replace it by any other technology. It also makes it hard to set up and run automated unit tests. (Even though the Microsoft SQL Server client library is a technology contained in the BCL, its usage implies a relational database.) Other out-of-process resources like message queues, web services, and even the filesystem fall into this category. The symptoms of this type of **Dependency** are lack of late binding and extensibility, as well as disabled **Testability**.
- *The* **Dependency** *doesn't yet exist, or is still in development.*
- *The* **Dependency** *isn't installed on all machines in the development organization.* This may be the case for expensive third-party libraries or **Dependencies** that can't be installed on all operating systems. The most common symptom is disabled **Testability**.
- *The* **Dependency** *contains nondeterministic behavior.* This is particularly important in unit tests because all tests must be deterministic. Typical sources of nondeterminism are random numbers and algorithms that depend on the current date or time.

  Because the BCL defines common sources of nondeterminism, such as `System.Random`, `System.Security.Cryptography.RandomNumberGenerator`, or `System.DateTime.Now`, you can't avoid having a reference to the assembly in which they're defined. Nevertheless, you should treat them as **Volatile Dependencies** because they tend to destroy **Testability**.

IMPORTANT **Volatile Dependencies** are the focal point of DI. It's for **Volatile Dependencies** rather than **Stable Dependencies** that you introduce **Seams** into your application. Again, this obligates you to compose them using DI.

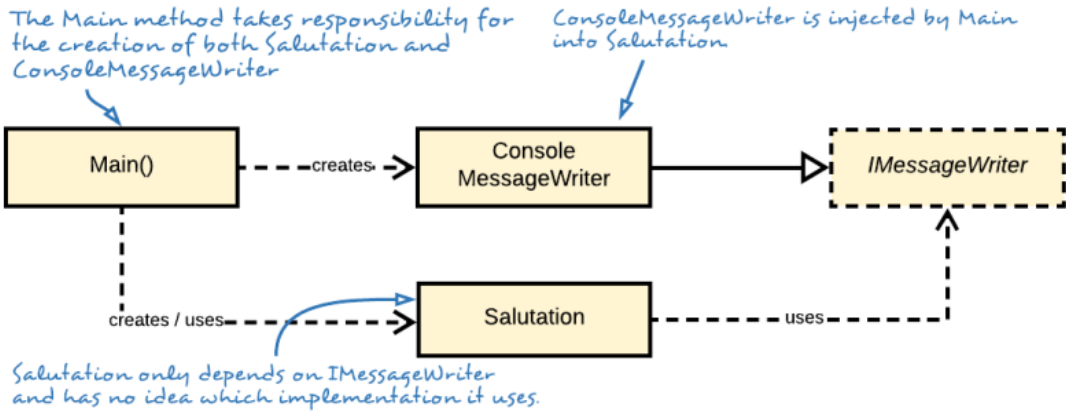
Now that you understand the differences between **Stable** and **Volatile Dependencies**, you can begin to see the contours of the scope of DI. Loose coupling is a pervasive design principle, so DI (as an enabler) should be everywhere in your code base. There's no hard line between the topic of DI and good software design, but to define the scope of the rest of the book, we'll quickly describe what it covers.

1.4 DI scope

As we discussed before, an important element of DI is to break up various responsibilities into separate classes. One responsibility that we take away from classes is the task of creating instances of **Dependencies**. The task of creating instances of **Dependencies** is referred to as **Object Composition**.

We discussed this in our Hello DI! example where our `Salutation` class was released of the responsibility of creating its **Dependency**. Instead this responsibility was moved to the application's `Main` method. Here's the UML diagram again:

Figure 1.12. Relationship between the collaborators of the Hello DI! application (repeated)


The Main method takes responsibility for the creation of both Salutation and ConsoleMessageWriter
ConsoleMessageWriter is injected by Main into Salutation
Main()
creates
Console MessageWriter
IMessageWriter
creates / uses
Salutation
uses
Salutation only depends on IMessageWriter and has no idea which implementation it uses.

As a class relinquishes control of **Dependencies**, it gives up more than the decision to select particular implementations. By doing this, we, as developers, gain some advantages. At first, it may seem like a disadvantage to let a class surrender control over which objects are created, but, we don't lose that control—we only move it to another place.

NOTE As developers, we gain control by removing a class's control over its **Dependencies**. This is an application of the **Single Responsibility Principle**. Classes should not have to deal with the creation of their **Dependencies**.

Object Composition isn't the only dimension of control that we remove: a class also loses the ability to control the *lifetime* of the object. When a **Dependency** instance is injected into a class, the consumer doesn't know when it was created, or when it'll go out of scope. This should be of no concern to the consumer. Making the consumer oblivious to the lifetime of its **Dependencies**simplifies the consumer.

DI gives you an opportunity to manage **Dependencies** in a uniform way. When consumers directly create and set up instances of **Dependencies**, each may do so in its own way. This can be inconsistent with how other consumers do it. You've no way to centrally manage **Dependencies** and no easy way to address **Cross-Cutting Concerns**. With DI, you gain the ability to **Intercept**each **Dependency** instance and act on it before it's passed to the consumer. This provides extensibility in applications.

With DI, you can compose applications while intercepting **Dependencies** and controlling their lifetimes. **Object Composition**, **Interception**, and **Lifetime Management** are three dimensions of DI. Next we'll cover each of these briefly; a more detailed treatment follows in part 3 of the book.

1.4.1 Object Composition

To harvest the benefits of extensibility, late binding, and parallel development, you must be able to compose classes into applications. This means that you'll want to create an application out of individual classes by putting them together, much like plugging electrical appliances together. And, as with electrical appliances, you'd want to easily rearrange those classes when new requirements are introduced, ideally without having to make changes to existing classes.

Object Composition is often the primary motivation for introducing DI into an application. In fact, initially, DI was synonymous with **Object Composition**; it's the only aspect discussed in Martin Fowler's original article on the subject.[17]

You can compose classes into an application in several ways. When we discussed late binding, we used a configuration file and a bit of dynamic object instantiation to manually compose the application from the available modules. We could also have used **Configuration as Code** using a **DI Container**. We'll return to these in chapter 12.

Many people refer to DI as *Inversion of Control (IoC)*. These two terms are sometimes used interchangeably, but DI is a subset of IoC. Throughout the book, we'll consistently use the most specific term—DI. If we mean IoC, we'll refer to it specifically.

Dependency Injection or *Inversion of Control*?

The term **Inversion of Control** originally meant any sort of programming style where an overall framework or runtime controlled the program flow.[18] According to that definition, most software developed on the .NET Framework uses IoC. When you write an ASP.NET Core MVC application, for instance, you create controller classes with action methods, but it's ASP.NET Core that will be calling your action methods. This means you aren't in control—the framework is.

These days, we're so used to working with frameworks that we don't consider this to be special, but it's a different model from being in full control of your code. This can still happen for a .NET application, most notably for command-line executables. As soon as `Main` is invoked, your code is in full control. It controls program flow, lifetime —everything. No special events are being raised and no overridden members are being invoked.

Before DI had a name, people started to refer to libraries that manage **Dependencies** as **Inversion of Control Containers**, and soon, the meaning of IoC gradually drifted towards that particular meaning: **Inversion of Control** over **Dependencies**. Always the taxonomist, Martin Fowler introduced the term *Dependency Injection* to specifically refer to IoC in the context of dependency management. Dependency Injection has since been widely accepted as the most correct terminology. In short, IoC is a much broader term that includes, but isn't limited to, DI.

1.4.2 Object Lifetime

A class that has surrendered control of its **Dependencies** gives up more than the power

[17] See Martin Fowler's *"Inversion of Control Containers and the Dependency Injection pattern,"* 2004,martinfowler.com/articles/injection.html.
[18] See Martin Fowler's *"InversionOfControl,"* 2005, martinfowler.com/bliki/InversionOfControl.html.

to select particular implementations of an **Abstraction**. It also gives up the power to control when instances are created and when they go out of scope.

In .NET, the garbage collector takes care of these things for us. A consumer can have its **Dependencies** injected into it and use them for as long as it wants. When it's done, the **Dependencies** go out of scope. If no other classes reference them, they're eligible for garbage collection.

What if two consumers share the same type of **Dependency**? Listing 1.5 illustrates that you can choose to inject a separate instance into each consumer, whereas listing 1.6 shows that you can alternatively choose to share a single instance across several consumers. But, from the perspective of the consumer, there's no difference. According to the **Liskov Substitution Principle**, the consumer must treat all instances of a given interface equally.

Listing 1.5. Consumers get their own instance of the same type of Dependency.

```
IMessageWriter writer1 = new ConsoleMessageWriter();      ❶
IMessageWriter writer2 = new ConsoleMessageWriter();      ❶

var salutation = new Salutation(writer1);                 ❷
var valediction = new Valediction(writer2);               ❷
```

❶ Two instances of the same IMessageWriter **Dependency** are created.
❷ Each consumer gets its own private instance.

Listing 1.6. Consumers sharing an instance of the same type of Dependency

```
IMessageWriter writer = new ConsoleMessageWriter();       ❶

var salutation = new Salutation(writer);                  ❷
var valediction = new Valediction(writer);                ❷
```

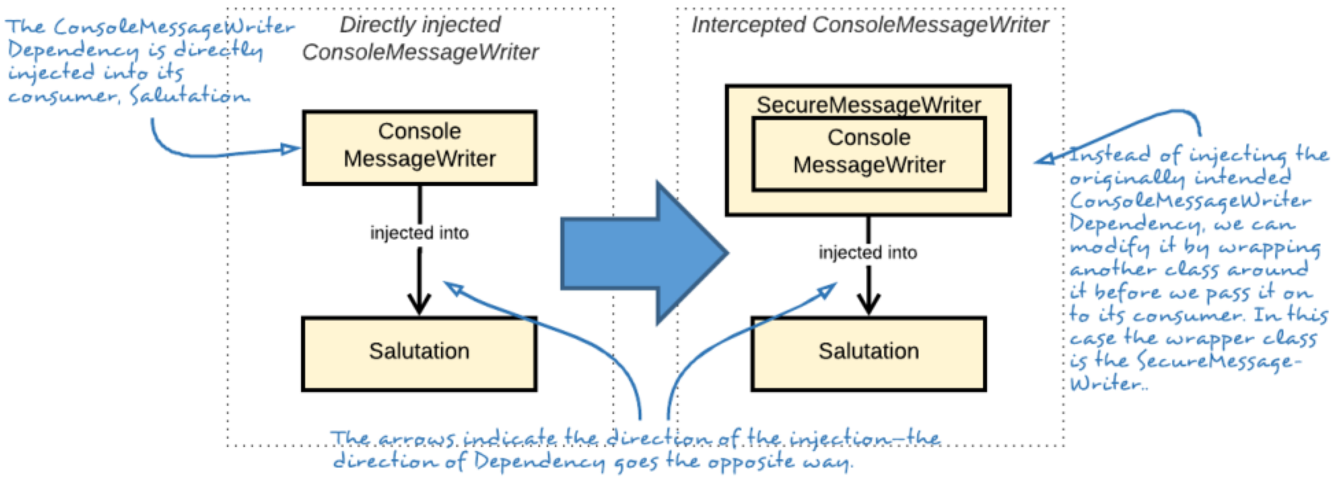
❶ One instance is created.
❷ That same instance is injected into two consumers.

Because **Dependencies** can be shared, a single consumer can't possibly control its lifetime. As long as a managed object can go out of scope and be garbage collected, this isn't much of an issue. But when **Dependencies** implement the `IDisposable` interface, things become much more complicated as we'll discuss in section 8.2. As a whole, **Lifetime Management** is a separate dimension of DI and important enough that we've set aside all of chapter 8 for it.

1.4.3 Interception

When we delegate control over **Dependencies** to a third party, as figure 1.13 shows, we also provide the power to modify them before we pass them on to the classes consuming them.

Figure 1.13. Intercepting a `ConsoleMessageWriter`


The ConsoleMessageWriter Dependency is directly injected into its consumer, Salutation.
Directly injected ConsoleMessageWriter
Console MessageWriter
injected into
Salutation
Intercepted ConsoleMessageWriter
SecureMessageWriter
Console MessageWriter
injected into
Salutation
Instead of injecting the originally intended ConsoleMessageWriter Dependency, we can modify it by wrapping another class around it before we pass it on to its consumer. In this case the wrapper class is the SecureMessage-Writer.
The arrows indicate the direction of the injection—the direction of Dependency goes the opposite way.

In the Hello DI! example, we initially injected a `ConsoleMessageWriter` instance into a `Salutation` instance. Then, modifying the example, we added a security feature by creating a new `SecureMessageWriter` that only delegates further work to the `ConsoleMessageWriter` when the user is authenticated. This allows you to maintain the **Single Responsibility Principle**. It's possible to do this because you always program to interfaces; recall that **Dependencies** must always be **Abstractions**. In the case of the `Salutation`, it doesn't care whether the supplied `IMessageWriter` is a `ConsoleMessageWriter` or a `SecureMessageWriter`. The `SecureMessageWriter` can wrap a `ConsoleMessageWriter` that still performs the real work.

NOTE **Interception is an application of the Decorator design pattern. Don't worry if you aren't familiar with the Decorator design pattern. We'll provide a refresher in chapter 9, which is entirely devoted to Interception.**

Such abilities of **Interception** move us along the path towards **Aspect-Oriented Programming**, a closely related topic that we'll cover in chapters 10 and 11. With **Interception** and **Aspect-Oriented Programming**, you can apply **Cross-Cutting Concerns** such as logging, auditing, access control, validation, and so forth in a well-structured manner that lets you maintain Separation of Concerns.

1.4.4 DI in three dimensions

Although DI started out as a series of patterns aimed at solving the problem of **Object Composition**, the term has subsequently expanded to also cover **Object Lifetime** and **Interception**. Today, we think of DI as encompassing all three in a consistent way.

Object Composition tends to dominate the picture because, without flexible **Object Composition**, there'd be no **Interception** and no need to manage **Object Lifetime**. **Object Composition**has dominated most of this chapter and will continue to dominate this book, but you shouldn't forget the other aspects. **Object Composition** provides the foundation, and **Lifetime Management**addresses some important side effects. But it's mainly when it comes to **Interception** that you start to reap the benefits.

In part 3, we've devoted a chapter to each dimension briefly mentioned here. But it's important to know that, in practice, DI is more than **Object Composition**.

1.5 Conclusion

Dependency Injection is a means to an end, not a goal in itself. It's the best way to enable loose coupling, an important part of maintainable code. The benefits you can reap from loose coupling aren't always immediately apparent, but they'll become visible over time, as the complexity of a code base grows. An important point about loose coupling and DI is that, in order to be effective, it should be everywhere in your code base.

A tightly coupled code base will eventually deteriorate into Spaghetti Code; whereas, a well-designed, loosely coupled code base can stay maintainable.[19] It takes more than loose coupling to reach a truly supple design, but programming to interfaces is a prerequisite.[20]

TIP **DI must be pervasive. You can't easily retrofit loose coupling onto an existing code base.**[21]

DI is nothing more than a collection of design principles and patterns. It's more about a way of thinking and designing code than it's about tools and techniques. The purpose of DI is to make code maintainable. Small code bases, like a classic Hello World example, are inherently maintainable because of their size. This is why DI tends to look like over-engineering in simple examples. The larger the code base becomes, the more visible the benefits. We've dedicated the next two chapters to a larger and more complex example to showcase these benefits.

1.6 Summary

- Dependency Injection is a set of software design principles and patterns that enable you to develop loosely coupled code. Loose coupling makes code more maintainable.
- When you've a loosely coupled infrastructure in place, it can be used by anyone and adapted to changing needs and unanticipated requirements without having to make large changes to the application's code base and its infrastructure.
- Troubleshooting tends to become less taxing because the scope of likely culprits narrows.
- DI enables *late binding*, which is the ability to replace classes or modules with different ones without the need for the original code to be recompiled.

[19] See page 119, William J. Brown et al., *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (Wiley Computer Publishing, 1998).
[20] See page 243, Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2004).
[21] Michael C. Feathers, *Working Effectively with Legacy Code* (Prentice Hall, 2004)

- DI makes it easier for code to be extended and reused in ways not explicitly planned for, similar to the way that you can rewire electrical appliances using plugs and sockets.
- DI simplifies parallel development on the same code base because the separation of concerns allows each team member or even entire teams to work more easily on isolated parts.
- DI makes software more **Testable** because you can replace **Dependencies** with test implementations when writing unit tests.
- When you practice DI, collaborating classes should rely on the infrastructure to provide the necessary services. You do this by letting your classes depend on interfaces, instead of concrete implementations.
- Classes shouldn't ask a third-party for their **Dependencies**. This is an anti-pattern called **Service Locator**. Instead, classes should specify their required **Dependencies** statically using constructor parameters, a practice called **Constructor Injection**.
- Many developers think that DI requires specialized tooling, a so-called **DI Container**. This is a myth. A **DI Container** is a useful, but optional tool.
- One of the most important software design principles that enables DI is the **Liskov Substitution Principle**. It allows replacing one implementation of an interface with another without breaking either the client or implementation.
- **Dependencies** are considered **Stable** in the case that they're already available, have deterministic behavior, don't require a setup runtime environment (such as a relational database), and don't need to be replaced, wrapped, or intercepted.
- **Dependencies** are considered **Volatile** when they are under development, aren't always available on all development machines, contain nondeterministic behavior, or need to be replaced, wrapped, or intercepted.
- **Volatile Dependencies** are the focal point of DI. We inject **Volatile Dependencies** into a class's constructor.
- By removing control over **Dependencies** from their consumers, and moving that control into the application's entry point, you gain the ability to apply **Cross-Cutting Concerns** more easily and can manage the lifetime of **Dependencies** more effectively.
- To succeed, you need to apply DI pervasively. All *components* should get their required **Volatile Dependencies** using **Constructor Injection**. It's hard to retrofit loose coupling and DI onto an existing code base.