Dependency Injection in .NET
by Mark Seemann

Chapter 1

Copyright 2011 Manning Publications
brief contents

PART 1  PUTTING DEPENDENCY INJECTION ON THE MAP ..........1
  1  ■ A Dependency Injection tasting menu  3
  2  ■ A comprehensive example  29
  3  ■ DI Containers  58

PART 2  DI CATALOG ...............................................................93
  4  ■ DI patterns  95
  5  ■ DI anti-patterns  133
  6  ■ DI refactorings  162

PART 3  DIY DI .........................................................................197
  7  ■ Object Composition  199
  8  ■ Object Lifetime  236
  9  ■ Interception  275

PART 4  DI CONTAINERS ........................................................311
  10  ■ Castle Windsor  313
  11  ■ StructureMap  347
  12  ■ Spring.NET  385
  13  ■ Autofac  417
  14  ■ Unity  448
  15  ■ MEF  492
You may have heard that making a sauce béarnaise is difficult. Even many people who cook regularly 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.

**DEFINITION**  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 preparation can fail—the sauce may curdle or separate, and if that happens, you can’t resurrect it. It takes about 45 minutes to prepare, so a failed attempt means that you’ll have no 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 will tell you, it’s not difficult. You don’t have to be a professional cook to make it. Anyone learning to make it will fail at least once, but once you get the hang of it, you’ll succeed every time.

I think Dependency Injection (DI) is like sauce béarnaise. It’s assumed to be difficult and so few employ it. 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 us 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.

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

> 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-olds) should rely on the infrastructure (the parents) to provide the necessary services.

As figure 1.1 shows, this chapter is fairly linear in structure. First, I introduce DI, including its purpose and benefits. Although I include examples, overall, this chapter has less code than any other chapter in the book.

Before I introduce DI, I’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), I’ll discuss benefits and scope,

---

essentially 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. 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 write prototypes or applications that never make it past release 1, you’ll soon find yourself maintaining and extending existing code bases. To be able to work effectively with such a code base, it must be as maintainable as possible.

One of many ways to make code maintainable is through loose coupling. As far back as 1995, when the Gang of Four wrote Design Patterns, this was already common knowledge:

Program to an interface, not an implementation.

This important piece of advice isn’t the conclusion, but, rather, the premise, of Design Patterns; to wit: it appears on page 18. Loose coupling makes code extensible, and extensibility makes it maintainable.

DI is nothing more than a technique that enables loose coupling. However, 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 Unlearning DI

Like a Hollywood martial arts cliché, you must unlearn before you can learn. There are many misconceptions about DI, and if you carry those around, you’ll misinterpret what you read in this book. You must clear your mind to understand DI.

There are at least four common myths about DI:

- 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 we can start to learn about DI.

---

2 Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software (New York, Addison-Wesley, 1994), 18.
CHAPTER 1  A Dependency Injection tasting menu

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 standard software that supports different runtime environments. You may have an application that can run on more than one database engine: for example, one that supports both Oracle and SQL Server. To support this feature, the rest of the application can talk to the database through an interface. The code base can provide different implementations of this interface to provide access to Oracle and SQL Server, respectively. A configuration option can be used 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 *does* enable this scenario, but the fallacy is to think that the relationship is symmetric. Just because DI enables late binding doesn’t mean it’s only relevant in late binding scenarios. As figure 1.2 illustrates, late binding is only one of the many aspects of DI.

If you thought 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 to support unit testing. This isn’t true, either—although DI is certainly an important part of supporting unit testing.

To tell you the truth, my original introduction to DI came from struggling with certain aspects of Test-Driven Development (TDD). During that time I discovered DI and learned that other people had used it to support some of the same scenarios I was 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 to support unit testing is like claiming that it’s only relevant for supporting late binding. Figure 1.3 shows that although this is a different view, it’s a view as narrow as figure 1.2. In this book, I’ll do my best to show you the whole picture.

If you thought 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\textsuperscript{3} that we can use to create instances of the DEPENDENCIES that we need.

In the introduction to this chapter, I 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.

If you thought of DI as a SERVICE LOCATOR—that is, a general-purpose Factory—this is something you need to unlearn. DI is the opposite of a SERVICE LOCATOR; it’s a way to structure code so that we never have to imperatively ask for DEPENDENCIES. Rather, we force 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 we should use a DI CONTAINER.

A DI CONTAINER is an optional library that can make it easier for us to compose components when we wire up an application, but it’s in no way required. When we compose applications without a DI CONTAINER we call it POOR MAN’S DI; it takes a little more work, but other than that we don’t have to compromise on any DI principles.

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.

You may think that, although I’ve exposed four myths about DI, I have yet to make a compelling case against any of them. That’s true. In a sense, this whole book is one big argument against these common misconceptions.

In my experience, unlearning is vital because people tend to try to retrofit what I tell them about DI and align it with what they think they already know. When this happens, it takes a lot of time before it finally dawns on them that some of their most basic

\textsuperscript{3} Ibid., 87.
premises are wrong. I want to spare you that experience. So, if you can, try to read this book as though you know nothing about DI.

Let’s assume that you don’t know anything about DI or its purpose and begin by reviewing what DI does.

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 I could refer you to well-established authorities like the Gang of Four for details, I find it only fair to explain why this is true.

Software development is still a rather new profession, so in many ways we’re still figuring out how to implement good architecture. However, 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 1.4. 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. Although the cord’s long enough to give you a certain degree of movement, you can’t take the dryer with you. Apparently, the hotel management has decided that the cost of replacing stolen hair dryers is high enough to justify what’s otherwise an obviously inferior implementation.

What happens when the hair dryer stops working? The hotel has to call in a skilled professional who can deal with the issue. To fix the hardwired hair dryer, they will have to cut the power to the room, rendering it temporarily useless. Then, the technician will use special tools to painstakingly 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.

![Figure 1.4 In a cheap hotel room, you might find the hair dryer wired directly into the wall outlet. This is equivalent to using the common practice of writing tightly coupled code.](image)
Writing maintainable code

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 1.5, we use plugs and sockets. A socket defines a shape that the plug must match. In an analogy to software design, the socket is an interface.

In contrast to the hardwired hair dryer, plugs and sockets define a loosely coupled model for connecting electrical appliances. As long as the plug fits into the socket, 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, I would guess that you need power for a computer much more than you do for a hair dryer. That’s not a problem: we unplug the hair dryer and plug a computer into the same socket, as shown in figure 1.6.

It’s amazing that the concept of a socket predates computers by decades, and yet it provides an essential service to computers, too. 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 client or 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.

As figure 1.7 illustrates, we can unplug the computer if we don’t need to use it at the moment. Even though nothing is plugged in, the wall doesn’t explode.

If we unplug the computer from the wall, neither the wall outlet nor the computer breaks down (in fact, if it’s a laptop computer, it can even run on its batteries for a period of time). With software, however, a client often expects a service to be available. If the service was removed, we get a NullReferenceException. To deal with this type of situation, we can create an implementation of an interface that does “nothing.” This is a design pattern known as Null Object, and it corresponds roughly to unplugging the

---

Figure 1.6 Using sockets and plugs, we can replace the original hair dryer from figure 1.5 with a computer. This corresponds to the LISKOV SUBSTITUTION PRINCIPLE.

Figure 1.7 Unplugging the computer causes neither wall nor computer to explode. This can be roughly likened to the Null Object pattern.

---

Writing maintainable code

computer from the wall. Because we’re using loose coupling, we can replace a real implementation with something that does nothing without causing trouble.

There are many other things we can do. If we live in a neighborhood with intermittent power failures, we may wish to keep the computer running by plugging it into an Uninterrupted Power Supply (UPS), as shown in figure 1.8: we connect the UPS to the wall outlet and the computer to the UPS.

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 at different times. As figure 1.6 demonstrates, we can run the computer without a UPS, but we 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. It gives us the ability to incrementally introduce new features and CROSS-CUTTING CONCERNS without having to rewrite or change a lot of our existing code.

Another way to add new functionality to an existing code base is to compose an existing implementation of an interface with a new implementation. When we aggregate several implementations into one, we use the Composite design pattern. Figure 1.9 illustrates how this corresponds to plugging diverse appliances into a power strip.

The power strip has a single plug that we can insert into a single socket, while 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.

---

5 Gamma, Design Patterns, 175.
6 Ibid., 163.
Here’s a final example. We sometimes find ourselves 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 1.10, along when traveling, you need an adapter to charge it. Appropriately, there’s a design pattern with the same name.

The **Adapter** design pattern works like its physical namesake. It can be used to match two related, yet separate, interfaces to each other. This is particularly useful

---

Figure 1.9  A power strip makes it possible to plug several appliances into a single wall outlet. This corresponds to the Composite design pattern.

Figure 1.10  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.

---

\(^7\) Ibid., 139.
when you have an existing third-party API that you wish to expose as an instance of an interface your application consumes.

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 unpredicted 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.

Loose coupling can make a code base much more maintainable.

That’s the easy part. Programming to an interface instead of an implementation is easy. The question is, where do the instances come from? In a sense, this is what this entire book is about.

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 doesn’t compile:

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

An interface has no constructor, 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, I think you’re ready for an example.

### 1.2 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. In this section, I’ll show you what the code looks like and briefly outline some key benefits without going into details—in the rest of the book, I’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, and then I’ll show you the collaborating classes:

```csharp
private static void Main()
{
    IMessageWriter writer = new ConsoleMessageWriter();
    var salutation = new Salutation(writer);
    salutation.Exclaim();
}
```
The program needs to write to the console, so it creates a new instance of `ConsoleMessageWriter` that encapsulates exactly 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, which results in the message being written to the screen.

Figure 1.11 shows the relationship between the collaborators.

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

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

    public Salutation(IMessageWriter writer)
    {
        if (writer == null)
        {
            throw new ArgumentNullException("writer");
        }
        this.writer = writer;
    }

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

The `Salutation` class depends on a custom interface called `IMessageWriter`, and it requests an instance of it through its constructor 1. This is called **CONSTRUCTOR INJECTION** and is described in detail in chapter 4, which also contains a more detailed walkthrough of a similar code example.

The `IMessageWriter` instance is later used in the implementation of the `Exclaim` method 2, which writes the proper message to the DEPENDENCY.
IMessageWriter is a simple interface defined for the occasion:

```csharp
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:

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

The `ConsoleMessageWriter` class implements `IMessageWriter` by wrapping the 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.

You may be wondering about the benefit of replacing a single line of code with two classes and an interface with a total line count of 11, and rightly so. There are several benefits to be harvested from doing this.

### 1.2.2 Benefits of DI

How is the previous example better than the usual single line of code we normally use to implement Hello World in C#? In this example, DI adds an overhead of 1,100%, but, as complexity increases from one line of code to tens of thousands, this overhead diminishes and all but disappears. Chapter 2 provides a more complex example of applied DI, and although that example is still overly simplistic compared to real-life applications, you should notice that DI is far less intrusive.

I 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, we gain the benefits listed in table 1.1.

In table 1.1, I listed the late binding benefit first because, in my 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, this is most likely because they never consider the other benefits.

**LATE BINDING**

When I 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.
CHAPTER 1  A Dependency Injection tasting menu

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

<table>
<thead>
<tr>
<th>Benefit</th>
<th>Description</th>
<th>When is it valuable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Late binding</td>
<td>Services can be swapped with other services.</td>
<td>Valuable in standard software, but perhaps less so in enterprise applications where the runtime environment tends to be well-defined</td>
</tr>
<tr>
<td>Extensibility</td>
<td>Code can be extended and reused in ways not explicitly planned for.</td>
<td>Always valuable</td>
</tr>
<tr>
<td>Parallel development</td>
<td>Code can be developed in parallel.</td>
<td>Valuable in large, complex applications; not so much in small, simple applications</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Classes with clearly defined responsibilities are easier to maintain.</td>
<td>Always valuable</td>
</tr>
<tr>
<td>Testability</td>
<td>Classes can be unit tested.</td>
<td>Only valuable if you unit test (which you really, really should)</td>
</tr>
</tbody>
</table>

Remember when I asked you 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. However, requirements change.

NoSQL, Windows Azure, and the argument for composability

Years ago, I 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.

Windows 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 Sourcing8 are also becoming increasingly important.

In section 1.2.1, you didn’t use late binding because you explicitly created a new instance of IMessageWriter by hard-coding creation of a new ConsoleMessageWriter

---

instance. However, you can introduce late binding by changing only a single piece of the code. You only need to change this line of code:

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

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

```csharp
var typeName = ConfigurationManager.AppSettings["messageWriter"];
var type = Type.GetType(typeName, true);
IMessageWriter writer = (IMessageWriter)Activator.CreateInstance(type);
```

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:

```xml
<appSettings>
</appSettings>
```

**WARNING** 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.

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 enables us to efficiently recompose the application, similar to the way that we 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. The next listing shows how you can add that feature without changing any of the existing features: you add a new implementation of the IMessageWriter interface.

**Listing 1.2 Extending the Hello DI application with a security feature**

```csharp
public class SecureMessageWriter : IMessageWriter
{
    private readonly IMessageWriter writer;
```
public SecureMessageWriter(IMessageWriter writer)
{
    if (writer == null)
    {
        throw new ArgumentNullException("writer");
    }
    this.writer = writer;
}

public void Write(string message)
{
    if (Thread.CurrentPrincipal.Identity.IsAuthenticated)
    {
        this.writer.Write(message);
    }
}

The SecureMessageWriter class implements the IMessageWriter interface while also consuming it: it uses CONSTRUCTOR INJECTION to request an instance of IMessageWriter. This is a standard application of the Decorator design pattern that I mentioned in section 1.1.2. We’ll talk much more about Decorators in chapter 9.

The Write method is implemented by first checking whether the current user is authenticated 1. Only if this is the case does it allow the decorated writer field to Write 2 the message.

**NOTE** The Write method in listing 1.2 accesses the current user via an AMBIENT CONTEXT. A more flexible, but slightly more complex, option would’ve been to also supply the user via CONSTRUCTOR INJECTION.

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:

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

Notice that you decorate the old ConsoleMessageWriter instance with the new SecureMessageWriter class. Once more, the Salutation class is unmodified because it only consumes the IMessageWriter interface.

Loose coupling enables you to write code which is *open for extensibility, but closed for modification*. This is called the OPEN/CLOSED PRINCIPLE. The only place where you need to modify the code is at the application’s entry point; we call this the COMPOSITION ROOT.

The SecureMessageWriter implements the security features of the application, whereas the ConsoleMessageWriter addresses the user interface. This enables us to vary these aspects independently of each other and compose them as needed.
PARALLEL DEVELOPMENT
Separation of concerns makes it possible to develop code in parallel teams. When a software development project grows to a certain size, it becomes necessary to separate the development team into multiple teams of manageable sizes. Each team is assigned responsibility for an area of the overall application.

To demarcate responsibilities, each team will develop 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.

In the above 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’ve needed to agree upon 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 known benefit of the SINGLE RESPONSIBILITY PRINCIPLE, which states that each class should have only a single responsibility.

Adding new features to an application becomes simpler because it’s clear where changes should be applied. More often than not, we don’t even 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
For some, TESTABILITY is the least of their worries; for others, it’s an absolute requirement. Personally, I belong in the latter category: in my career, I’ve declined several job offers because they involved working with certain products that aren’t TESTABLE.

DEFINITION An application is considered TESTABLE when it can be unit tested.

The benefit of TESTABILITY is perhaps the most controversial of the benefits I’ve listed. Many developers and architects don’t practice unit testing, so they consider this benefit irrelevant at best. Others, like me, consider it essential. Michael Feathers even defines the term Legacy Application as any application that isn’t covered by unit tests.9

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 we can inject Test Doubles into the System Under Test (SUT), as we shall see in listing 1.3.

The ability to replace the intended DEPENDENCIES with test-specific replacements is a by-product of loose coupling, but I chose to list it as a separate benefit because the derived value is different.

---

CHAPTER 1  A Dependency Injection tasting menu

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 user interface or whatever other interface it provides. Such manual tests are time consuming and expensive to perform, so automated testing is much preferred.

There are different types of automated testing, such as unit testing, integration testing, performance testing, stress testing, and so on. Because unit testing has few 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 very important 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.

Depending on the type of application I’m developing, I may or may not care about the ability to do late binding, but I always care about Testability. Some developers don’t care about Testability, but find late binding important for the application they’re developing.

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 will never be used in the final application. Instead, they serve as placeholders for the real Dependencies, when these are unavailable or undesirable to use.

There’s a complete pattern language around Test Doubles, and many subtypes, such as Stubs, Mocks, and Fakes.10

---

10 Gerard Meszaros, xUnit Test Patterns: Refactoring Test Code (New York, Addison-Wesley, 2007), 522.
EXAMPLE: UNIT TESTING THE “HELLO” LOGIC
In section 1.21, you saw a Hello DI example. Although I showed you the final code first, I actually developed it using TDD. Listing 1.3 shows the most important unit test.

NOTE Don’t worry if you don’t have experience with unit testing or dynamic mocks. They will occasionally pop up throughout this book, but are in no way a prerequisite for reading it.11

Listing 1.3 Unit testing the Salutation class

```
[Fact]
public void ExclaimWillWriteCorrectMessageToMessageWriter()
{
    var writerMock = new Mock<IMessageWriter>();
    var sut = new Salutation(writerMock.Object);
    sut.Exclaim();
    writerMock.Verify(w => w.Write("Hello DI!"));
}
```

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 dynamic mock can be very useful—in this case, you use Moq.12 but you could’ve used other libraries or rolled your own instead. 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 any production implementation of IMessageWriter to pollute the test.

To create the Salutation class, you pass in the Mock instance of IMessageWriter. Because writerMock is an instance of Mock<IMessageWriter>, the Object property is a dynamically created instance of IMessageWriter. Injecting the required DEPENDENCY through the constructor is called CONSTRUCTOR INJECTION.

After exercising the System Under Test (SUT), you can use the Mock to verify that the Write method was invoked with the correct text. With Moq, you do this by calling the Verify method with an expression that defines what you expected. If the IMessageWriter.Write method was invoked with the "Hello DI!" string, the Verify method call completes, but if the Write method wasn’t called, or called with a different parameter, the Verify method would throw an exception and the test would fail.

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 answers this question by injecting the DEPENDENCIES from the outside. CONSTRUCTOR INJECTION is the preferred method of doing that.

---

11 You may, however, want to read The Art of Unit Testing followed by xUnit Test Patterns. See the bibliography for more details.
12 code.google.com/p/moq/
1.3 What to inject and what not to inject

In the previous section, I described the motivational forces that make us think about DI in the first place. If you’re convinced that loose coupling is a good idea, you may want to make everything loosely coupled. Overall, this is a good idea. When you must decide how to package modules, loose coupling provides especially useful guidance. You don’t have to abstract everything away and make everything pluggable. In this section, I’ll provide you with some decision tools that can 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 module. In the previous section, I 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 would like to use an XMLWriter, which is 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.

1.3.1 Seams

Everywhere we decide to program against an interface instead of a concrete type, we 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. It’s also a place where we can disassemble the application and work with the modules in isolation.

The Hello DI sample I built in section 1.2 contains a seam between Salutation and ConsoleMessageWriter, as illustrated in figure 1.12. The Salutation class doesn’t directly depend on the ConsoleMessageWriter class; rather, it uses the IMessageWriter interface to write messages. You can take the application apart at this seam and reassemble it with a different message writer.

As you learn DI, it can be helpful to categorize your dependencies into stable dependencies and volatile dependencies, but deciding where to put your seams will soon become second nature to you. The next sections discuss these concepts in more detail.

---

13 Feathers, Working Effectively with Legacy Code, 29-44.
1.3.2 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 leverage to make your own code more succinct.

The BCL modules are always available to your application, because it needs the .NET Framework to run. The concern about parallel development doesn’t apply to these modules because they already exist, and 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—I call them stable because they’re already there, tend to be backwards compatible, and invoking them has deterministic outcomes.

Most STABLE DEPENDENCIES are BCL types, but other dependencies can be stable as well. The important criteria for STABLE DEPENDENCIES are

- 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 the class or module with another.

Ironically, DI CONTAINERS themselves will tend to be STABLE DEPENDENCIES, because they fit all the criteria. 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 COMPOSITION ROOT.

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

In general, DEPENDENCIES can be considered stable by exclusion: they’re stable if they aren’t volatile.

1.3.3 Volatile Dependencies

Introducing SEAMS into an application is extra work, so you should only do it when it’s necessary. There may be more than one reason it’s necessary to isolate a DEPENDENCY behind a SEAM, but they’re closely related to the benefits of loose coupling, discussed in section 1.2.2.

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 I call them VOLATILE DEPENDENCIES for that reason. A DEPENDENCY should be considered Volatile if any of the following criteria is true:

- The DEPENDENCY introduces a requirement to set up and configure a runtime environment for the application. A relational database is the archetypical example: if we don’t hide the relational database behind a SEAM, we can never replace it by any other technology. It also makes it hard to set up and run automated unit tests.
Databases are a good example of BCL types that are volatile dependencies: even though LINQ to Entities is a technology contained in the BCL, its usage implies a relational database.

Other out-of-process resources such as message queues, web services, and even the file system fall into this category. Please note that it isn’t so much the concrete .NET types that are volatile, but rather what they imply about the runtime environment.

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, but is still in development. The obvious symptom of such dependencies is the inability to do parallel 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 should be deterministic. Typical sources of nondeterminism are random numbers and algorithms that depend on the current date or time.

Note that common sources of nondeterminism, such as System.Random, System.Security.Cryptography.RandomNumberGenerator, or System.DateTime.Now are defined in mscorlib, so 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.

Volatile dependencies are the focal point of DI. It’s for volatile dependencies, rather than stable dependencies, that we introduce seams into our application. Again, this obliges us to compose them using DI.

Now that you understand the differences between stable and volatile dependencies, you may 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, I’ll quickly describe what it covers.

### 1.4 DI scope

As we saw in section 1.2, 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.

As a class relinquishes control of dependencies, it gives up more than the decision to select particular implementations. However, as developers, we gain some advantages.

**NOTE** As developers, we gain control by removing that control from the classes that consume dependencies. This is an application of the single responsibility
**DI scope**

**PRINCIPLE:** these classes should only deal with their given area of responsibility, without concerning themselves with how DEPENDENCIES are created.

At first, it may seem like a disadvantage to let a class surrender control over which objects are created, but, as developers, we don’t lose that control—we only move it to another place.

However, **OBJECT COMPOSITION** isn’t the only dimension of control that we remove, because 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 will go out of scope. Many times, this is of no concern to the consumer, but in other cases, it may be.

DI gives us 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, which may be inconsistent with how other consumers do it. We have no way to centrally manage DEPENDENCIES, and no easy way to address CROSS-CUTTING CONCERNS. With DI, we gain the ability to **intercept** each DEPENDENCY instance and act upon it before it’s passed to the consumer.

With DI, we can **compose** applications while **intercepting** dependencies and controlling their **lifetimes**. **OBJECT COMPOSITION, INTERCEPTION, and LIFETIME MANAGEMENT** are three dimensions of DI. Next I’ll cover them 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**, we must be able to compose classes into applications (see figure 1.13). Such **OBJECT COMPOSITION** is often the foremost motivation for introducing DI into an application. Initially, DI was synonymous with **OBJECT COMPOSITION**; it’s the only aspect discussed in Martin Fowler’s original article on the subject.¹⁴

---


**Figure 1.13** **OBJECT COMPOSITION** signifies that modules can be composed into applications.
There are several ways we can compose classes into an application. When I discussed *late binding* I used a configuration file and a bit of dynamic object instantiation to manually compose the application from the available modules, but I could also have used CODE AS CONFIGURATION or a DI CONTAINER. We’ll return to these in chapter 7.

Although the original meaning of DI was closely tied to OBJECT COMPOSITION, other aspects have also turned out to be relevant.

### 1.4.2 Object Lifetime

A class that has surrendered control of its DEPENDENCIES gives up more than the power 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 a lot 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? Figure 1.14 illustrates that we can choose to inject a separate instance into each consumer, whereas figure 1.15 shows that we may alternatively choose to share a single instance across several consumers. However, 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.

Because DEPENDENCIES may 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 a whole, LIFETIME MANAGEMENT is a separate dimension of DI and important enough that I’ve set aside all of chapter 8 for it.

Giving up control of a DEPENDENCY also means giving up control of its lifetime; something else higher up in the call stack must manage the lifetime of the DEPENDENCY.

### 1.4.3 Interception

When we delegate control over DEPENDENCIES to a third party, as figure 1.16 shows, we also gain the power to modify them before we pass them on to the classes consuming them.
In the Hello DI example, I initially injected a `ConsoleMessageWriter` instance into a `Salutation` instance. Then, modifying the example, I added a security feature by creating a new `SecureMessageWriter` that only delegates further work to the `ConsoleMessageWriter` when the user is authenticated. This allows us to maintain the SINGLE RESPONSIBILITY PRINCIPLE.

This is possible to do because we 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—I’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, nonetheless, is outside the scope of this book. With INTERCEPTION, we can apply CROSS-CUTTING CONCERNS such as logging, auditing, access control, validation, and so forth in a well-structured manner that lets us 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, I think of DI as encompassing all three in a consistent way.

OBJECT COMPOSITION tends to dominate the picture because, without flexible OBJECT COMPOSITION, there would be no INTERCEPTION and no need to manage OBJECT LIFETIME. OBJECT COMPOSITION has dominated most of this chapter, and will continue to dominate the book, but we 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 we start to reap the benefits.

In part 3, I’ve devoted a chapter to each dimension, but I provided an overview here because it’s important to know that, in practice, DI is more than OBJECT COMPOSITION.
1.5 Summary

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 we can reap from loose coupling aren’t always immediately apparent, but they become visible over time, as the complexity of a code base grows. A tightly coupled code base will eventually deteriorate into Spaghetti Code, whereas a well-designed, loosely coupled code base can stay maintainable. It takes more than loose coupling to reach a truly Supple Design, but programming to interfaces is a prerequisite.

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 is about tools and techniques—an important point about loose coupling and DI is that, in order to be effective, it should be everywhere in your code base.

TIP DI must be pervasive. You can’t easily retrofit loose coupling onto an existing code base.

There are many misconceptions about DI. Some people think that it only addresses narrow problems, such as late binding or unit testing; although these aspects of software design certainly benefit from DI, the scope is much broader. The overall purpose is maintainability.

In the beginning of the chapter, I stated that you must unlearn to learn DI. This will remain true for the rest of the book: you must free your mind. In an excellent blog post, Nicholas Blumhardt writes:

The dictionary or associative array is one of the first constructs we learn about in software engineering. It’s easy to see the analogy between a dictionary and an IoC container that composes objects using dependency injection.18

The idea of DI as a service modeled along the lines of a dictionary leads directly to the service locator anti-pattern. This is why I put so much emphasis on the need to clear your mind of even the most basic assumptions. After all, when we’re talking about dictionaries, we’re talking about stuff that belongs in the “reptile brain of programming.”

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. I’ve dedicated the next chapter to a larger and more complex example to showcase these benefits.

---

17 However, see Feathers, Michael, Working Effectively with Legacy Code (New York, Prentice Hall, 2004).
Dependency Injection in .NET
Mark Seemann

Dependency Injection is a great way to reduce tight coupling between software components. Instead of hard-coding dependencies, such as specifying a database driver, you inject a list of services that a component may need. The services are then connected by a third party. This technique enables you to better manage future changes and other complexity in your software.

Dependency Injection in .NET introduces DI and provides a practical guide for applying it in .NET applications. The book presents the core patterns in plain C#, so you'll fully understand how DI works. Then you'll learn to integrate DI with standard Microsoft technologies like ASP.NET MVC, and to use DI frameworks like StructureMap, Castle Windsor, and Unity. By the end of the book, you'll be comfortable applying this powerful technique in your everyday .NET development.

What's Inside
• Many C#-based examples
• A catalog of DI patterns and anti-patterns
• Using both Microsoft and open source DI frameworks

This book is written for C# developers. No previous experience with DI or DI frameworks is required.

Mark Seemann is a software architect living in Copenhagen. Previously a developer and architect at Microsoft, Mark is now an independent consultant.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/DependencyInjectionin.NET

“Realistic examples keep the big picture in focus... A real treat.”
—From the Foreword by Glenn Block, Senior Program Manager, Microsoft

“Well-written, thoughtful, easy to follow, and ... timeless.”
—David Barkol, Neudesic, LLC

“Fills a huge need for .NET designers.”
—Paul Grebenc
PCA Services, Inc.

“Takes the mystery out of a mystifying topic.”
—Rama Krishna, 3C Software

“All you need to know ... and more!”
—Jonas Bandi, TechTalk