Sample Chapter

Dependency Injection in .NET

Mark Seemann

Foreword by Glenn Block

MANNING

Dependency Injection in .NET

by Mark Seemann

Chapter 4

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

4 DI patterns

Menu

- CONSTRUCTOR INJECTION
- PROPERTY INJECTION
- METHOD INJECTION
- AMBIENT CONTEXT

Like all professionals, cooks have their own jargon that allow them to communicate about complex food preparation in a language that often sounds esoteric to the rest of us. It doesn't help that most of the terms they use are based on French (unless you already speak French, that is).

Sauces are a great example of the way cooks use their professional terminology. In chapter 1, I briefly discussed sauce béarnaise, but I didn't elaborate on the taxonomy that surrounds it (see figure 4.1).

A sauce béarnaise is really a *sauce hollandaise* where the lemon juice is replaced by a *reduction* of vinegar, shallots, chervil, and tarragon. Other sauces are based on sauce hollandaise—including my favorite, *sauce mousseline,* which is made by *folding* whipped cream into the hollandaise.

Did you notice all the jargon? Instead of saying, "carefully mixing the whipped cream into the sauce, taking care not to collapse it," I used the term *folding*. When you know what it means, it's a lot easier to say and understand.

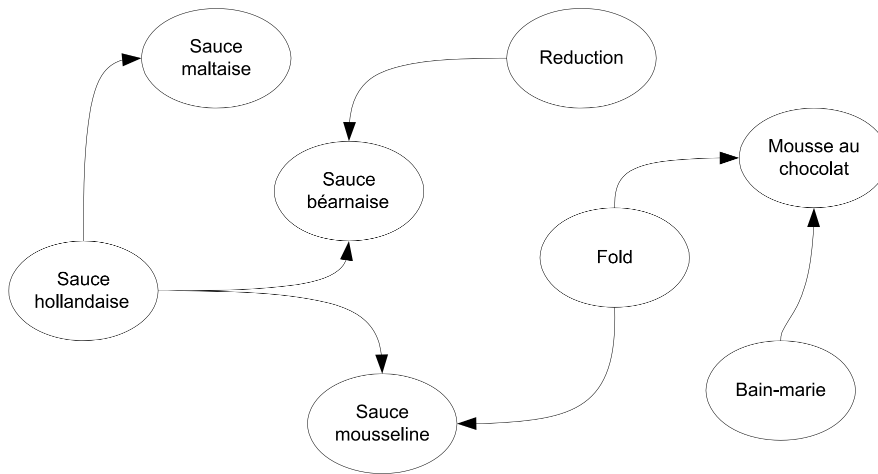


Figure 4.1 Several sauces are based on *sauce hollandaise*. In a *sauce béarnaise* the lemon is replaced with a *reduction* of vinegar and certain herbs, whereas the distinguishing feature of *sauce mousseline* is that whipped cream is *folded* into it—a technique also used to make *mousse au chocolat*.

The term *folding* isn't limited to sauces—it's a general way to combine something that's whipped with other ingredients. When making a classic *mousse au chocolat*, for example, I *fold* whipped egg whites into a mixture of whipped egg yolks and melted chocolate.

In software development, we have a complex and impenetrable jargon of our own. Although you may not know what the cooking term *bain-marie* refers to, I'm pretty sure most cooks would be utterly lost if you told them that "strings are immutable classes that represent sequences of Unicode characters."

When it comes to talking about how to structure code to solve particular types of problems, we have *Design Patterns* that give names to common solutions. In the same way that the terms *sauce hollandaise* and *fold* help us succinctly communicate how to make *sauce mousseline*, patterns help us talk about how code is structured. The eventing system in .NET is based on a design pattern called *Observer*, and `foreach` loops on *Iterator*.[1]

In this chapter, I'll describe the four basic DI patterns listed in figure 4.2. Because the chapter is structured to provide a catalog of patterns, each pattern is written so that it can be read independently. However, CONSTRUCTOR INJECTION is by far the most important of the four patterns.

Don't worry if you have only limited knowledge of design patterns in general. The main purpose of a design pattern is to provide a detailed and self-contained description of a particular way of attaining a goal—a recipe, if you will.

---

[1] Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (New York: Addison-Wesley, 1994), 293, 257.

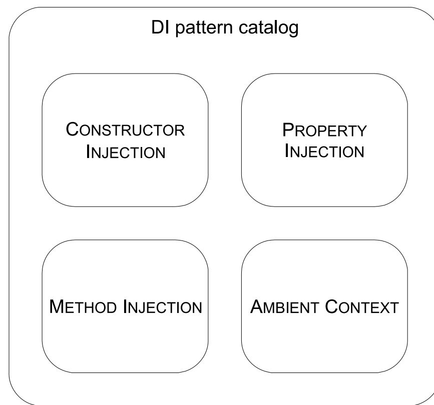


Figure 4.2 The structure of this chapter takes the form of a pattern catalog. Each pattern is written so it can be read independently of the other patterns.

For each pattern, I'll provide a short description, a code example, advantages and disadvantages, and so on. You can read about all four patterns in sequence or only read the ones that interest you. The most important pattern is CONSTRUCTOR INJECTION, which you should use in most situations; the other patterns become more specialized as the chapter progresses.

4.1 Constructor Injection

How do we guarantee that a necessary Dependency is always available to the class we're currently developing?

BY REQUIRING ALL CALLERS TO SUPPLY THE DEPENDENCY AS A PARAMETER TO THE CLASS'S CONSTRUCTOR.

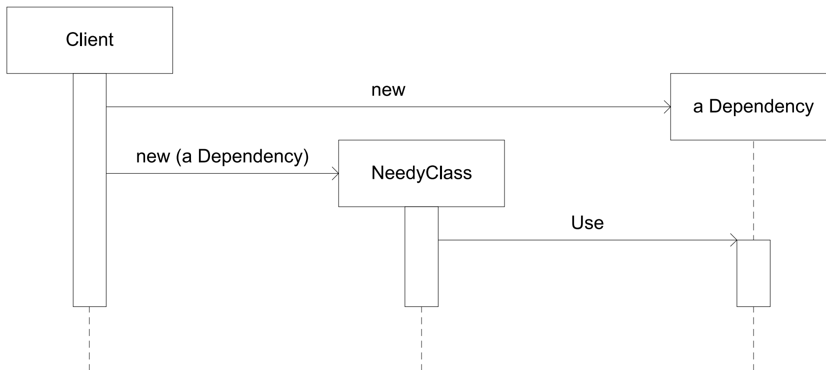


Figure 4.3 `NeedyClass` needs an instance of Dependency to work, so it requires any Client to supply an instance via its constructor. This guarantees that the instance is available to `NeedyClass` whenever it's needed.

When a class requires an instance of a DEPENDENCY to work at all, we can supply that DEPENDENCY through the class's constructor, enabling it to store the reference for future (or immediate) use.

4.1.1 How it works

The class that needs the DEPENDENCY must expose a public constructor that takes an instance of the required DEPENDENCY as a constructor argument. In most cases, this should be the only available constructor. If more than one DEPENDENCY is needed, additional constructor arguments can be used.

Listing 4.1 CONSTRUCTOR INJECTION

```
private readonly DiscountRepository repository;          // 4 Dependency field is read-only

public RepositoryBasketDiscountPolicy(                   // 1 Inject Dependency as constructor argument
    DiscountRepository repository)
{
    if (repository == null)                              // 2 Guard Clause
    {
        throw new ArgumentNullException("repository");
    }

    this.repository = repository;                        // 3 Save the Dependency for later
}
```

The DEPENDENCY (in the previous listing that would be the abstract `DiscountRepository` class) is a required constructor argument ❶. Any client code that doesn't supply an

instance of the DEPENDENCY can't compile. However, because both interfaces and abstract classes are reference types, a caller can pass in null as an argument to make the calling code compile; we need to protect the class against such misuse with a Guard Clause[2] ❷.

Because the combined efforts of the compiler and the Guard Clause guarantee that the constructor argument is valid if no exception is thrown, at this point, the constructor can save the DEPENDENCY for future use without knowing anything about the real implementation ❸.

It's good practice to mark the field holding the DEPENDENCY as `readonly`—this guarantees that once the initialization logic of the constructor has executed: the field can't be modified ❹. This isn't strictly required from a DI point of view, but it protects you from accidentally modifying the field (such as setting it to null) somewhere else in the depending class's code.

TIP Keep the constructor free of any other logic. The SINGLE RESPONSIBILITY PRINCIPLE implies that members should do only one thing, and now that we use the constructor to inject DEPENDENCIES, we should prefer to keep it free of other concerns.

TIP Think about CONSTRUCTOR INJECTION as *statically declaring a class's Dependencies*. The constructor signature is compiled with the type and is available for all to see. It clearly documents that the class requires the DEPENDENCIES it requests through its constructor.

When the constructor has returned, the new instance of the depending class is in a consistent state with a proper instance of its DEPENDENCY injected into it. Because it holds a reference to this DEPENDENCY, it can use it as often as necessary from any of its other members. It doesn't need to test for null, because the instance is guaranteed to be present.

4.1.2 When to use it

CONSTRUCTOR INJECTION should be your default choice for DI. It addresses the most common scenario where a class *requires* one or more DEPENDENCIES, and no reasonable LOCAL DEFAULTS are available.

CONSTRUCTOR INJECTION addresses that scenario well because it *guarantees* that the DEPENDENCY is present. If the depending class absolutely can't function without the DEPENDENCY, that guarantee is valuable.

TIP If at all possible, constrain the design to a single constructor. Overloaded constructors lead to ambiguity: which constructor should a DI CONTAINER use?

In cases where the local library can supply a good default implementation, PROPERTY INJECTION may be a better fit—but this is often not the case. In the earlier chapters, I

---

[2] Martin Fowler et al., *Refactoring: Improving the Design of Existing Code* (New York: Addison-Wesley, 1999), 50.

showed many examples of Repositories as DEPENDENCIES. These are good examples of DEPENDENCIES where the local library can supply no good default implementation, because the proper implementations belong in specialized Data Access libraries.

Table 4.1 CONSTRUCTOR INJECTION advantages and disadvantages

Advantages	Disadvantages
Injection guaranteed Easy to implement	Some frameworks make using CONSTRUCTOR INJECTION difficult.

Apart from the guaranteed injection already discussed, this pattern is also easy to implement using the four steps implied by listing 4.1.

The main disadvantage to CONSTRUCTOR INJECTION is that you need to modify your current application framework to support it. Most frameworks assume that your classes will have a default constructor and may need special help to create instances when the default constructor is missing. In chapter 7, I explain how to enable CONSTRUCTOR INJECTION for common application frameworks.

An apparent disadvantage of CONSTRUCTOR INJECTION is that it requires that the entire dependency graph is initialized immediately—often at application startup. However, although this sounds inefficient, it's rarely an issue. After all, even for a complex object graph, we're typically talking about creating a dozen new object instances, and creating an object instance is something the .NET Framework does extremely fast. Any performance bottleneck your application may have will appear in other places, so don't worry about it.

In extremely rare cases this may be a real issue, but in chapter 8, I'll describe the *Delayed* lifetime option that offers one possible remedy to this issue. For now, I'll merely observe that there may (in fringe cases) be a potential issue with initial load and move on.

4.1.3 Known use

Although CONSTRUCTOR INJECTION tends to be ubiquitous in applications employing DI, it's not very present in the .NET Base Class Library (BCL). This is mainly because the BCL is a set of libraries and not a full-fledged application.

Two related examples where we can see a sort of CONSTRUCTOR INJECTION in the BCL is with `System.IO.StreamReader` and `System.IO.StreamWriter`. Both take a `System.IO.Stream` instance in their constructors. They also have a lot of overloaded constructors that take a file path instead of a `Stream` instance, but these are convenience methods that internally create a `FileStream` based on the specified file path—here are all the `StreamWriter` constructors, but the `StreamReader` constructors are similar:

```
public StreamWriter(Stream stream);
public StreamWriter(string path);
public StreamWriter(Stream stream, Encoding encoding);
public StreamWriter(string path, bool append);
```

```
public StreamWriter(Stream stream, Encoding encoding,
    int bufferSize);
public StreamWriter(string path, bool append, Encoding encoding);
public StreamWriter(string path, bool append, Encoding encoding,
    int bufferSize);
```

The `Stream` class is an abstract class that serves as an ABSTRACTION upon which `StreamWriter` and `StreamReader` operate to perform their duties. You can supply any `Stream` implementation in their constructors and they will use it, but they will throw `ArgumentNullExceptions` if you try to slip them a null `Stream`.

Although the BCL can provide us with examples where we can see CONSTRUCTOR INJECTION in use, it's always more instructive to see an example. The next section walks you through a full implementation example.

4.1.4 Example: Adding a currency provider to the shopping basket

I'd like to add a new feature to the sample commerce application I expanded upon in chapter 2—namely, the ability to perform currency conversions. I'll spread the example throughout this chapter to demonstrate the different DI patterns in play, but when I'm done, the homepage should look like figure 4.4.

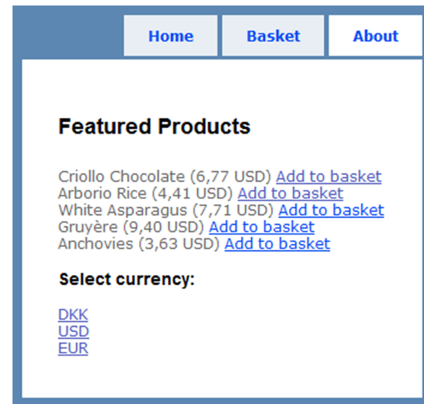


Figure 4.4 The sample commerce application with currency conversion implemented. The user can now select among three different currencies, and both product prices and basket totals (on the basket page) will be displayed in that currency.

One of the first things you need is a `CurrencyProvider`—a DEPENDENCY that can provide you with the currencies you request. You define it like this:

```
public abstract class CurrencyProvider
{
    public abstract Currency GetCurrency(string currencyCode);
}
```

The `Currency` class is another abstract class that provides conversion rates between itself and other currencies:

```
public abstract class Currency
{
    public abstract string Code { get; }

    public abstract decimal GetExchangeRateFor(
        string currencyCode);
}
```

You want the currency conversion feature on all pages that display prices, so you need it in both the `HomeController` and the `BasketController`. Because both implementations are quite similar, I'll only show you the `BasketController`.

A `CurrencyProvider` is likely to represent an out-of-process resource, such as a web service or a database that can supply conversion rates. This means that it would be most fitting to implement a concrete `CurrencyProvider` in a separate project (such as a Data Access library). Hence, there's no reasonable LOCAL DEFAULT. At the same time, the `BasketController` class will need a `CurrencyProvider` to be present; CONSTRUCTOR INJECTION is a good fit. The following listing shows how the `CurrencyProvider` DEPENDENCY is injected into the `BasketController`.

Listing 4.2 Injecting a `CurrencyProvider` into the `BasketController`

```
private readonly IBasketService basketService;
private readonly CurrencyProvider currencyProvider;

public BasketController(IBasketService basketService,
    CurrencyProvider currencyProvider)
{
    if (basketService == null)
    {
        throw new
            ArgumentNullException("basketService");
    }
    if (currencyProvider == null)
    {
        throw new
            ArgumentNullException("currencyProvider");
    }

    this.basketService = basketService;
    this.currencyProvider = currencyProvider;
}
```

4 Dependency fields are read-only

1 Inject Dependencies as constructor arguments

2 Guard Clauses

3 Save Dependencies for later

Because the `BasketController` class already had a DEPENDENCY on `IBasketService`, you add the new `CurrencyProvider` DEPENDENCY as a second constructor argument 1 and then follow the same sequence outlined in listing 4.1: Guard Clauses guarantee that the DEPENDENCIES aren't null 2, which means it's safe to store them for later use 3 in read-only fields 4.

Now that the `CurrencyProvider` is guaranteed to be present in the `BasketController`, it can be used from anywhere—for example, in the `Index` method:

```
public ViewResult Index()
{
    var currencyCode =
        this.CurrencyProfileService.GetCurrencyCode();
    var currency =
        this.currencyProvider.GetCurrency(currencyCode);

    // …
}
```

I haven't yet discussed the `CurrencyProfileService`, so for now, know that it provides the current user's preferred currency code. In section 4.2.4, I'll discuss the `CurrencyProfileService` in greater detail.

Given a currency code, the `CurrencyProvider` can be invoked to provide a `Currency` that represents that code. Notice that you can use the `currencyProvider` field without needing to check it in advance, because it's guaranteed to be present.

Now that you have the `Currency`, you can then proceed to perform the rest of the work in the `Index` method; note that I haven't yet shown that implementation. As we progress through this chapter, I'll build on this method and add more currency conversion functionality along the way.

4.1.5 Related patterns

CONSTRUCTOR INJECTION is the most generally applicable DI pattern available, and also the easiest to implement correctly. It applies when the DEPENDENCY is *required*.

If we need to make the DEPENDENCY optional, we can change to PROPERTY INJECTION if we have a proper LOCAL DEFAULT.

When the DEPENDENCY represents a CROSS-CUTTING CONCERN that should be potentially available to any module in the application, we can use an AMBIENT CONTEXT, instead.

The next pattern in this chapter is PROPERTY INJECTION, which is closely related to CONSTRUCTOR INJECTION; the only deciding parameter is whether the DEPENDENCY is optional or not.

4.2 Property Injection

How do we enable DI as an option in a class when we have a good Local Default?

BY EXPOSING A WRITABLE PROPERTY THAT LETS CALLERS SUPPLY A DEPENDENCY IF THEY WISH TO OVERRIDE THE DEFAULT BEHAVIOR.

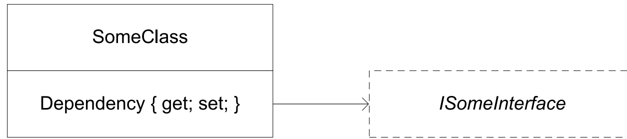


Figure 4.5 `SomeClass` **has an optional DEPENDENCY on** `ISomeInterface`**; instead of requiring callers to supply an instance, it's giving callers an option to define it via a property.**

When a class has a good LOCAL DEFAULT, but we still want to leave it open for extensibility, we can expose a writable property that allows a client to supply a different implementation of the class's DEPENDENCY than the default.

NOTE PROPERTY INJECTION is also known as SETTER INJECTION.

Referring to figure 4.5, clients wishing to use the `SomeClass` as-is can `new` up an instance of the class and use it without giving it a second thought, whereas clients wishing to modify the behavior of the class can do so by setting the `Dependency` property to a different implementation of `ISomeInterface`.

4.2.1 How it works

The class that uses the DEPENDENCY must expose a public writable property of the DEPENDENCY's type. In a bare-bones implementation, this may be as simple as the following listing.

Listing 4.3 PROPERTY INJECTION

```
public partial class SomeClass
{
    public ISomeInterface Dependency { get; set; }
}
```

`SomeClass` depends on `ISomeInterface`. Clients can supply implementations of `ISomeInterface` by setting the `Dependency` property. Notice that in contrast to CONSTRUCTOR INJECTION, you can't mark the `Dependency` property's backing field as `readonly` because you allow callers to modify the property at any given time of `SomeClass`'s lifetime.

Other members of the depending class can use the injected DEPENDENCY to perform their duties, like this:

```
public string DoSomething(string message)
{
    return this.Dependency.DoStuff(message);
}
```

However, such an implementation is fragile because the `Dependency` property isn't guaranteed to return an instance of `ISomeInterface`. Code like this would throw a `NullReferenceException` because the value of the `Dependency` property is null:

```
var mc = new SomeClass();
mc.DoSomething("Ploeh");
```

This issue can be solved by letting the constructor set a default instance on the property, combined with a proper Guard Clause in the property's setter.

Another complication arises if you allow clients to switch the DEPENDENCY in the middle of the class's lifetime. This can be addressed by introducing an internal flag that only allows a client to set the DEPENDENCY once.[3]

The example in section 4.2.4 shows how you can deal with these complications, but before I get to that, I'd like to explain when it's appropriate to use PROPERTY INJECTION.

4.2.2 When to use it

PROPERTY INJECTION should only be used when the class you're developing has a good LOCAL DEFAULT and you still want to enable callers to provide different implementations of the class's DEPENDENCY.

PROPERTY INJECTION is best used when the DEPENDENCY is *optional*.

NOTE There's some controversy around the issue of whether PROPERTY INJECTION indicates an optional DEPENDENCY. As a general API design principle, I consider properties to be optional because you can easily forget to assign them and the compiler doesn't complain. If you accept this principle in the general case, you must also accept it in the special case of DI.

Local Default

When you're developing a class that has a DEPENDENCY, you probably have a particular implementation of that DEPENDENCY in mind. If you're writing a Domain Service that accesses a Repository, you're most likely planning to develop an implementation of that Repository that uses a relational database.

It would be tempting to make that implementation the default used by the class under development. However, when such a prospective default is implemented in a different assembly, using it as a default would mean creating a hard reference to that other assembly, effectively violating many of the benefits of loose coupling described in chapter 1.

Conversely, if the intended default implementation is defined in the same library as the consuming class, you don't have that problem. This is unlikely to be the case with Repositories, but such LOCAL DEFAULTS are more likely as Strategies.[4]

The example in this section contains an example of a LOCAL DEFAULT.

---

[3] Eric Lippert calls this *popsicle immutability*. Eric Lippert, "Immutability in C# Part One: Kinds of Immutability," 2007, http://blogs.msdn.com/ericlippert/archive/2007/11/13/immutability-in-c-part-one-kinds-of-immutability.aspx

[4] Gamma, *Design Patterns*, 315.

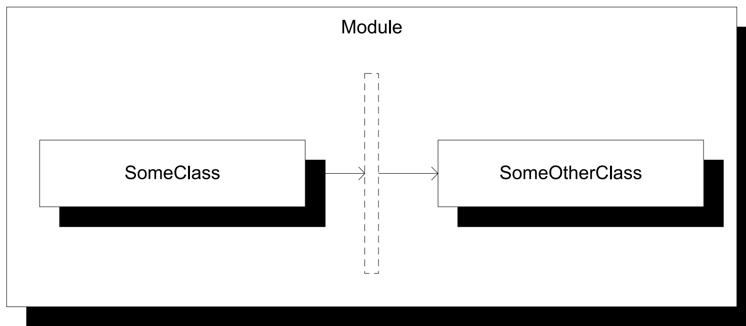


Figure 4.6 Even within a single module, we can introduce ABSTRACTIONS (represented by the vertical rectangle) that help reduce class coupling within that module. The main motivation for doing this is to enhance maintainability of the module by enabling classes to vary independently of each other.

In chapter 1, I discussed many good reasons for writing code with loose coupling, isolating modules from each other. However, loose coupling can also be applied to classes within a single module with great success. This is often done by introducing ABSTRACTIONS within a single module and letting classes communicate via ABSTRACTIONS, instead of being tightly coupled to each other.

Figure 4.6 illustrates that ABSTRACTIONS can be defined, implemented, and consumed within a single module with the main purpose of opening classes for extensibility.

NOTE The concept of opening a class for extensibility is captured by the OPEN/CLOSED PRINCIPLE[5] that, briefly put, states that a class should be open for extensibility, but closed for modification.

When we implement classes following the OPEN/CLOSED PRINCIPLE, we may have a LOCAL DEFAULT in mind, but we still provide clients with a way to extend the class by replacing the DEPENDENCY with something else.

NOTE PROPERTY INJECTION is only one among many different ways of applying the OPEN/CLOSED PRINCIPLE.

TIP Sometimes you only wish to provide an extensibility point, but leave the LOCAL DEFAULT as a no-op. In such cases, you can use the `Null Object`[6] pattern to implement the LOCAL DEFAULT.

TIP Sometimes you wish to leave the LOCAL DEFAULT in place, but have the ability to *add* more implementations. You can achieve this by modeling the DEPENDENCY around either the `Observer` or the `Composite` patterns.[7]

---

5 A good .NET-related introduction to the OPEN/CLOSED PRINCIPLE can be found in Jeremy Miller, "Patterns in Practice: The Open Closed Principle," (MSDN Magazine, June 2008). Also available online at http://msdn.microsoft.com/en-us/magazine/cc546578.aspx

6 Robert C. Martin et al., *Pattern Languages of Program Design 3* (New York: Addison-Wesley, 1998), 5.

7 Gamma, *Design Patterns*, 293, 163.

So far, I haven't shown you any examples of PROPERTY INJECTION, because the applicability of this pattern is more limited.

Table 4.2 PROPERTY INJECTION advantages and disadvantages

Advantages	Disadvantages
Easy to understand	Limited applicability Not entirely simple to implement robustly

The main advantage of PROPERTY INJECTION is that it's so easy to understand. I have often seen this pattern used as a first attempt when people decide to adopt DI.

Appearances can be deceptive, and PROPERTY INJECTION is fraught with difficulties. It's challenging to implement it in a robust manner. Clients may forget (or not want) to supply the DEPENDENCY, or mistakenly supply null as a value. Additionally: what should happen if a client tries to *change* the DEPENDENCY in the middle of the class's lifetime? This could lead to inconsistent or unexpected behavior, so you may want to protect yourself against that event.

With CONSTRUCTOR INJECTION, you could protect the class against such incidents by applying the `readonly` keyword to the backing field, but this isn't possible when you expose the DEPENDENCY as a writable property. In many cases, CONSTRUCTOR INJECTION is much simpler and more robust, but there are situations where PROPERTY INJECTION is the correct choice. This is the case when supplying a DEPENDENCY is optional, because you have a good LOCAL DEFAULT.

The existence of a good LOCAL DEFAULT depends in part on the granularity of modules. The .NET Base Class Library (BCL) ships as a rather large package; as long as the default stays within the BCL, it could be argued that it's also local. In the next section, I'll briefly touch upon that subject.

4.2.3 *Known use*

In the .NET BCL, PROPERTY INJECTION is a bit more common than CONSTRUCTOR INJECTION—probably because good LOCAL DEFAULTS are defined in many places.

`System.ComponentModel.IComponent` has a writable `Site` property that allows you to define an `ISite` instance. This is mostly used in design time scenarios (for example, by Visual Studio) to alter or enhance a component when it's hosted in a designer.

Another example that seems closer to how we're used to think about DI can be found in Windows Workflow Foundation (WF). The `WorkflowRuntime` class gives you the ability to add, get, and remove services. This isn't true PROPERTY INJECTION, because the API allows you to add zero or many untyped services through the same general-purpose API:

```
public void AddService(object service)
public T GetService<T>()
```

```
public object GetService(Type serviceType)
public void RemoveService(object service)
```

Although AddService will throw an ArgumentNullException if the service is null, there's no guarantee that you can retrieve a service with a given type because it may never have been added to the current WorkflowRuntime instance (in fact, this is because the GetService method is a SERVICE LOCATOR).

On the other hand, WorkflowRuntime comes with a lot of LOCAL DEFAULTS for each of the required services that it needs, and these are even named with the prefix *Default*, such as DefaultWorkflowSchedulerService and DefaultWorkflowLoaderService. If, for example, no alternative WorkflowSchedulerService is added either via the AddService method or the application configuration file, the DefaultWorkflowSchedulerService class is used.

With these BCL examples as hors d'œuvres, let's move on to a more substantial example of using and implementing PROPERTY INJECTION.

4.2.4 Example: Defining a currency profile service for the BasketController

In section 4.1.4, I started adding currency conversion functionality to the sample commerce application, and I briefly showed you some of the implementation of the BasketController's Index method—but glossed over the appearance of a CurrencyProfileService. Here's the deal:

The application needs to know which currency the user wishes to see. If you refer back to the screen shot in figure 4.4, you'll notice some currency links at the bottom of the screen. When the user clicks one of these links, you need to save the selected currency somewhere and associate that selection with the user. The CurrencyProfileService facilitates saving and retrieving the user's selected currency:

```
public abstract class CurrencyProfileService
{
    public abstract string GetCurrencyCode();

    public abstract void UpdateCurrencyCode(string currencyCode);
}
```

It's an ABSTRACTION that encodes the actions of applying and retrieving the current user's currency selection.

In ASP.NET MVC (and ASP.NET in general), you have a well-known piece of infrastructure that deals with such a scenario: the Profile service. An excellent LOCAL DEFAULT implementation of CurrencyProfileService is one that wraps around the ASP.NET Profile service and provides the necessary functionality defined by the GetCurrencyCode and UpdateCurrencyCode methods. The BasketController will use this DefaultCurrencyProfileService as the default while exposing a property that will allow the caller to substitute it by something else.

Listing 4.4 Exposing a `CurrencyProfileService` property

```
private CurrencyProfileService currencyProfileService;

public CurrencyProfileService CurrencyProfileService
{
    get
    {
        if (this.currencyProfileService == null)
        {
            this.CurrencyProfileService =
                new DefaultCurrencyProfileService(
                    this.HttpContext);
        }
        return this.currencyProfileService;
    }
    set
    {
        if (value == null)
        {
            throw new ArgumentNullException("value");
        }
        if (this.currencyProfileService != null)
        {
            throw new InvalidOperationException();
        }
        this.currencyProfileService = value;
    }
}
```

❶ **Lazy initialization of Local Default**

❷ **Only allow Dependency to be defined once**

The `DefaultCurrencyProfileService` itself uses CONSTRUCTOR INJECTION because it requires access to the `HttpContext`, and because the `HttpContext` isn't available to the `BasketController` at creation time, it has to defer creation of the `DefaultCurrencyProfileService` until the property is requested for the first time. In this case, lazy initialization ❶ is required, but in other cases, the LOCAL DEFAULT could have been assigned in the constructor. Notice that the LOCAL DEFAULT is assigned through the public setter, which ensures that all the Guard Clauses get evaluated.

The first Guard Clause guarantees that the DEPENDENCY isn't null. The next Guard Clause ❷ ensures that the DEPENDENCY can only be assigned once. In this case, I prefer that the `CurrencyProfileService` can't be changed once it's assigned, because otherwise it could lead to inconsistent behavior where a user's currency selection is first stored using one `CurrencyProfileService` and then subsequently retrieved from a different place, most likely yielding a different value.

You may also notice that, because you use the setter for lazy initialization ❶, the DEPENDENCY will also be locked once the property has been read. Once again, this is to protect clients from the case where the DEPENDENCY is subsequently changed without notification.

If you can get past all the Guard Clauses, you can save the instance for future use.

Compared to CONSTRUCTOR INJECTION, this is much more involved. PROPERTY INJECTION may look simple in its raw form as shown in listing 4.3, but properly implemented, it tends to be much more complex—and, in this example, I have even elected to ignore the issue of thread safety.

With the `CurrencyProfileService` in place, the start of the `BasketController`'s `Index` method can now use it to retrieve the user's preferred currency:

```
public ViewResult Index()
{
    var currencyCode =
        this.CurrencyProfileService.GetCurrencyCode();
    var currency =
        this.currencyProvider.GetCurrency(currencyCode);

    // …
}
```

This is the same code fragment shown in section 4.1.4. The `CurrencyProfileService` is used to get the user's selected currency, and the `CurrencyProvider` is subsequently used to retrieve that `Currency`.

In section 4.3.4, I'll return to the `Index` method to show what happens next.

4.2.5 Related patterns

You use PROPERTY INJECTION when the DEPENDENCY is optional because you have a good LOCAL DEFAULT. If you don't have a LOCAL DEFAULT, you should change the implementation to CONSTRUCTOR INJECTION.

When the DEPENDENCY represents a CROSS-CUTTING CONCERN that should be available to all modules in an application, you can implement it as an AMBIENT CONTEXT.

But before we get to that, METHOD INJECTION, in the next section, takes a slightly different approach, because it tends to apply more to the situation where we already have a DEPENDENCY that we wish to pass on to the collaborators we invoke.

4.3 Method Injection

How can we inject a Dependency into a class when it's different for each operation?
BY SUPPLYING IT AS A METHOD PARAMETER.

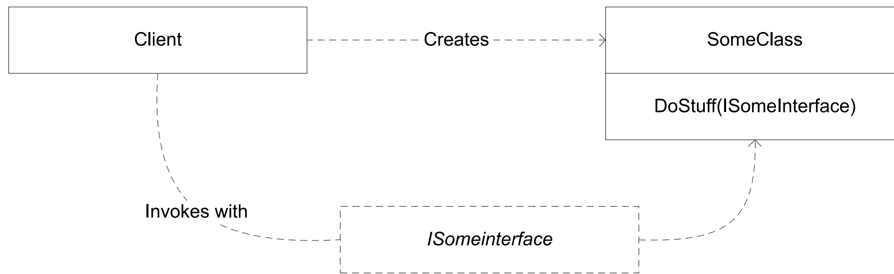


Figure 4.7 A Client creates an instance of `SomeClass`, but first injects an instance of the DEPENDENCY `ISomeInterface` with each method call.

When a DEPENDENCY can vary with each method call, you can supply it via a method parameter.

4.3.1 How it works

The caller supplies the DEPENDENCY as a method parameter in each method call. It can be as simple as this method signature:

```
public void DoStuff(ISomeInterface dependency)
```

Often, the DEPENDENCY will represent some sort of context for an operation that's supplied alongside a "proper" value:

```
public string DoStuff(SomeValue value, ISomeContext context)
```

In this case, the `value` parameter represents the value on which the method is supposed to operate, whereas the `context` contains information about the current context of the operation. The caller supplies the DEPENDENCY to the method, and the method uses or ignores the DEPENDENCY as it best suits it.

If the service uses the DEPENDENCY, it should be sure to test for null references first, as shown in the following listing.

Listing 4.5 Checking a method parameter for null before using it

```
public string DoStuff(SomeValue value, ISomeContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException("context");
    }

    return context.Name;
}
```

The Guard Clause guarantees that the context is available to the rest of the method body. In this example, the method uses the context's name to return a value, so ensuring that the context is available is important.

If a method doesn't use the supplied DEPENDENCY, it doesn't need to contain a Guard Clause. This sounds like a strange situation because, if the parameter isn't used, then why have it at all? However, you may need to keep it if the method is part of an interface implementation.

4.3.2 When to use it

METHOD INJECTION is best used when the DEPENDENCY can vary with each method call. This can be the case when the DEPENDENCY itself represents a value, but is often seen when the caller wishes to provide the consumer with information about the context in which the operation is being invoked.

This is often the case in add-in scenarios where an add-in is provided with information about the runtime context via a method parameter. In such cases, the add-in is required to implement an interface that defines the injecting method(s).

Imagine an add-in interface with this structure:

```
public interface IAddIn
{
    string DoStuff(SomeValue value, ISomeContext context);
}
```

Any class implementing this interface can be used as an add-in. Some classes may not care about the context at all, whereas other implementations will. A client may use a list of add-ins by calling each with a value and a context to return an aggregated result. This is shown in the following listing.

Listing 4.6 A sample add-in client

```
public SomeValue DoStuff(SomeValue value)
{
    if (value == null)
    {
        throw new ArgumentNullException("value");
    }

    var returnValue = new SomeValue();
    returnValue.Message = value.Message;

    foreach (var addIn in this.addIns)
    {
        returnValue.Message =
            addIn.DoStuff(returnValue, this.context);    // (1) Pass context to add-in
    }

    return returnValue;
}
```

The private `addIns` field is a list of `IAddIn` instances, which allows the client to loop through the list to invoke each add-in's `DoStuff` method. Each time the `DoStuff`

method is invoked on an add-in, the operation's context represented by the `context` field is passed as a method parameter ❶.

> **NOTE** METHOD INJECTION is closely related to the use of Abstract Factories described in section 6.1. Any Abstract Factory that takes an ABSTRACTION as input can be viewed as a variation of METHOD INJECTION.

At times, the value and the operational context are encapsulated in a single ABSTRACTION that works as a combination of both.

Table 4.3 METHOD INJECTION advantages and disadvantages

Advantages	Disadvantages
Allows the caller to provide operation-specific context	Limited applicability

METHOD INJECTION is different from other types of DI patterns we've seen so far in that the injection doesn't happen in a COMPOSITION ROOT, but, rather, dynamically at invocation time. This allows the caller to provide operation-specific context, which is a common extensibility mechanism used in the .NET BCL.

4.3.3 *Known use*

The .NET BCL provides many examples of METHOD INJECTION, particularly in the `System.ComponentModel` namespace.

`System.ComponentModel.Design.IDesigner` is used for implementing custom design-time functionality for components. It has an `Initialize` method that takes an `IComponent` instance so that it knows which component it's currently helping to design. Designers are created by `IDesignerHost` implementations that also take `IComponent` instances as parameters to create designers:

```
IDesigner GetDesigner(IComponent component);
```

This is a good example of a scenario where the parameter itself carries information: the component may carry information about which `IDesigner` to create, but at the same time, it's also the component upon which the designer must subsequently operate.

Another example in the `System.ComponentModel` namespace is provided by the `TypeConverter` class. Several of its methods take an instance of `ITypeDescriptorContext` that, as the name says, conveys information about the context of the current operation. Because there are many such methods, I don't want to list them all, but here is a representative example:

```
public virtual object ConvertTo(ITypeDescriptorContext context,
    CultureInfo culture, object value, Type destinationType)
```

In this method, the context of the operation is communicated explicitly by the `context` parameter while the value to be converted and the destination type are sent as separate parameters. Implementers can use or ignore the `context` parameter as they see fit.

ASP.NET MVC also contains several examples of METHOD INJECTION. The `IModelBinder` interface can be used to convert HTTP GET or POST data into strongly typed objects. Its only method is

```
object BindModel(ControllerContext controllerContext,
    ModelBindingContext bindingContext);
```

In the `BindModel` method, the `controllerContext` parameter contains information about the operation's context (among other things the `HttpContext`), whereas the `bindingContext` carries more explicit information about the values received from the browser.

When I recommend that CONSTRUCTOR INJECTION should be your preferred DI pattern, I'm assuming that you generally build applications based on frameworks. On the other hand, if you're building a framework, METHOD INJECTION can often be useful, because it allows you to pass information about the context to add-ins to the framework. That's one reason why we see METHOD INJECTION used so prolifically in the BCL.

4.3.4 Example: Converting baskets

In previous examples, we've seen how the `BasketController` in the sample commerce application retrieves the user's preferred currency (see sections 4.1.4 and 4.2.4). I'll now complete the currency conversion example by converting a `Basket` to the user's currency.

`Currency` is an ABSTRACTION that models a currency.

Listing 4.7 `Currency`

```
public abstract class Currency
{
    public abstract string Code { get; }

    public abstract decimal GetExchangeRateFor(
        string currencyCode);
}
```

The `Code` property returns the currency code for the `Currency` instance. `Currency` codes are expected to be international currency codes. For example, the currency code for Danish Kroner is DKK, whereas it's USD for US Dollars.

The `GetExchangeRateFor` method returns the exchange rate between the `Currency` instance and some other currency. Notice that this is an abstract method, which means that I'm making no assumptions about *how* that exchange rate is going to be found by the implementer.

In the next section, we'll examine how `Currency` instances are used to convert prices, and how this ABSTRACTION can be implemented and wired up so that you can convert some prices into such exotic currencies as US Dollars or Euros.

Injecting Currency

You'll use the `Currency` Abstraction as an information-carrying Dependency to perform currency conversions of Baskets, so you'll add a `ConvertTo` method to the `Basket` class:

```
public Basket ConvertTo(Currency currency)
```

This will loop through all the items in the basket and convert their calculated prices to the provided currency, returning a new `Basket` instance with the converted items. Through a series of delegated method calls, the implementation is finally provided by the `Money` class, as shown in the following listing.

Listing 4.8 Converting `Money` to another currency

```
public Money ConvertTo(Currency currency)        ◁ Inject Currency as method parameter ❶
{
    if (currency == null)
    {
        throw new ArgumentNullException("currency");
    }
    var exchangeRate =
        currency.GetExchangeRateFor(this.CurrencyCode);
    return new Money(this.Amount * exchangeRate,
        currency.Code);
}
```

The `Currency` is injected into the `ConvertTo` method via the `currency` parameter ❶ and checked by the ubiquitous Guard Clause that guarantees that the `currency` instance is available to the rest of the method body.

The exchange rate to the current currency (represented by `this.CurrencyCode`) is retrieved from the supplied `currency` and used to calculate and return the new `Money` instance.

With the implementation of the `ConvertTo` methods, you can finally implement the `Index` method on the `BasketController`, as shown in the following listing.

Listing 4.9 Converting a `Basket`'s currency

```
public ViewResult Index()
{
    var currencyCode =
        this.CurrencyProfileService.GetCurrencyCode();
    var currency =
        this.currencyProvider.GetCurrency(currencyCode);

    var basket = this.basketService              ❶ Convert the user's
        .GetBasketFor(this.User)                    basket to the
        .ConvertTo(currency);                       selected currency
    if (basket.Contents.Count == 0)
    {
        return this.View("Empty");
    }

    var vm = new BasketViewModel(basket);
    return this.View(vm);
}
```

The BasketController uses an IBasketService instance to retrieve the user's Basket. You may recall from chapter 2 that the IBasketService DEPENDENCY is provided to the BasketController via CONSTRUCTOR INJECTION. Once you have the Basket instance, you can convert it to the desired currency by using the ConvertTo method, passing in the currency instance ❶.

In this case, you're using METHOD INJECTION because the Currency ABSTRACTION is information-carrying, but will vary by context (depending on the user's selection). You could've implemented the Currency type as a concrete class, but that would've constrained your ability to define how exchange rates are retrieved.

Now that we've seen how the Currency class is used, it's time to change our viewpoint and examine how it might be implemented.

IMPLEMENTING CURRENCY

I haven't yet talked about how the Currency class is implemented because it's not that important from the point of view of METHOD INJECTION. As you may recall from section 4.1.4, and as you can see in listing 4.9, the Currency instance is served by the CurrencyProvider instance that was injected into the BasketController class by CONSTRUCTOR INJECTION.

To keep the example simple, I've shown what would happen if you decided to implement CurrencyProvider and Currency using a SQL Server database and LINQ to Entities. This assumes that the database has a table with exchange rates that has been populated in advance by some external mechanism. You could also have used a web service to request exchange rates from an external source.

The CurrencyProvider implementation passes a connection string on to the Currency implementation that uses this information to create an ObjectContext. The heart of the matter is the implementation of the GetExchangeRateFor method, shown in the following listing.

Listing 4.10 SQL Server–backed Currency implementation

```
public override decimal GetExchangeRateFor(string currencyCode)
{
    var rates = (from r in this.context.ExchangeRates
                 where r.CurrencyCode == currencyCode
                 || r.CurrencyCode == this.code
                 select r)
                 .ToDictionary(r => r.CurrencyCode);

    return rates[currencyCode].Rate
        / rates[this.code].Rate;
}
```

The first thing to do is get the rates from the database. The table contains rates as defined against a single, common currency (DKK), so you need both rates to be able to perform a proper conversion between two arbitrary currencies. You will index the retrieved currencies by currency code so that you can easily look them up in the final step of the calculation.

This implementation potentially performs a lot of out-of-process communication with the database. The `ConvertTo` method of `Basket` eventually calls this method in a tight loop, and hitting the database for each call is likely to be detrimental to performance. I'll return to this challenge in the next section.

4.3.5 Related patterns

Unlike the other DI patterns in this chapter, we mainly use METHOD INJECTION when we already have an instance of the DEPENDENCY we want to pass on to collaborators, but where we don't know the concrete types of the collaborators at design time (such as is the case with add-ins).

With METHOD INJECTION, we're on the other side of the fence compared to the other DI patterns: we don't consume the DEPENDENCY, but rather supply it. The types to which we supply the DEPENDENCY have no choice in how to model DI or whether they need the DEPENDENCY at all. They can consume it or ignore it as they see fit.

4.4 Ambient Context

How can we make a Dependency available to every module without polluting every API with Cross-Cutting Concerns?

BY MAKING IT AVAILABLE VIA A STATIC ACCESSOR.

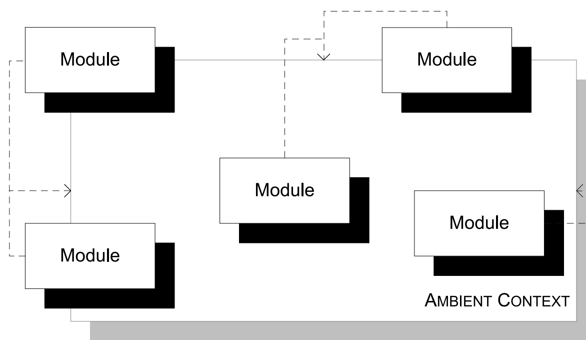


Figure 4.8 Every module can access an AMBIENT CONTEXT if it needs to.

A truly universal CROSS-CUTTING CONCERN can potentially pollute a large part of the API for an application if you have to pass an instance around to every collaborator. An alternative is to define a context that's available to anyone who needs it and that can be ignored by everyone else.

4.4.1 How it works

The AMBIENT CONTEXT is available to any consumer via a static property or method. A consuming class might use it like this:

```
public string GetMessage()
{
    return SomeContext.Current.SomeValue;
}
```

In this case, the context has a static `Current` property that a consumer can access. This property may be truly static, or may be associated with the currently executing thread.

To be useful in DI scenarios, the context itself must be an ABSTRACTION and it must be possible to modify the context from the outside—in the previous example, this means that the `Current` property must be writable. The context itself might be implemented as shown in the following listing.

Listing 4.11 AMBIENT CONTEXT

```
public abstract class SomeContext
{
    public static SomeContext Current
    {
        get
        {
            var ctx =
                Thread.GetData(
                Thread.GetNamedDataSlot("SomeContext"))
                as SomeContext;
```

❶ **Get current context from TLS**

```
            if (ctx == null)
            {
                ctx = SomeContext.Default;
                Thread.SetData(
                    Thread.GetNamedDataSlot("SomeContext"),
                    ctx);
            }
            return ctx;
        }
        set
        {
            Thread.SetData(
                Thread.GetNamedDataSlot("SomeContext"),     ❷ Save current
                value);                                        context in TLS
        }
    }

    public static SomeContext Default =
        new DefaultContext();
                                                           ❸ Value carried by
    public abstract string SomeValue { get; }                 the context
}
```

The context is an abstract class, which allows us to replace one context with another implementation at runtime.

In this example, the `Current` property stores the current context in Thread Local Storage (TLS) ❶, which means that every thread has its own context that's independent from the context of any other thread. In cases where no one has already assigned a context to TLS, a default implementation is returned. It's important to be able to guarantee that no consumer will ever get a `NullReferenceException` when they try to access the `Current` property, so there must be a good LOCAL DEFAULT. Note that in this case, the `Default` property is shared across all threads. This works because, in this example, `DefaultContext` (a class that derives from `SomeContext`) is immutable. If the default context was mutable, you would need to assign a separate instance to each thread to prevent cross-thread pollution.

External clients can assign a new context to TLS ❷. Notice that it's possible to assign null, but if this happens, the next read will automatically reassign the default context.

The whole point of having an AMBIENT CONTEXT is to interact with it. In this example, this interaction is represented by a solitary abstract string property ❸, but the context class can be as simple or complex as is necessary.

WARNING For simplicity's sake, I've skipped lightly over the thread-safety of the code in listing 4.11. If you decide to implement a TLS-based AMBIENT CONTEXT, be sure that you know what you're doing.

TIP The example in listing 4.11 uses TLS, but you can also use `CallContext` to similar effect.[8]

[8] See Mark Seemann, "Ambient Context," 2007, http://blogs.msdn.com/ploeh/archive/2007/07/23/AmbientContext.aspx for more information.

NOTE An AMBIENT CONTEXT doesn't need to be associated with a thread or call context. Sometimes, it makes more sense to make it apply to the entire AppDomain by making it `static`.

When you want to replace the default context with a custom context, you can create a custom implementation that derives from the context and assign it at the correct time:

```
SomeContext.Current = new MyContext();
```

For TLS-based contexts, you should assign the custom instance when you spawn the new thread, whereas for truly universal contexts, you can assign it in a COMPOSITION ROOT.

4.4.2 When to use it

AMBIENT CONTEXT should only be used in the rarest of cases. In most cases, CONSTRUCTOR INJECTION or PROPERTY INJECTION is far more suitable, but you may have a true CROSS-CUTTING CONCERN that would pollute every API in your application if you had to pass it along to all services.

WARNING AMBIENT CONTEXT is similar in structure to the SERVICE LOCATOR anti-pattern that I'll describe in chapter 5. The difference is that an AMBIENT CONTEXT only provides an instance of a single, strongly-typed DEPENDENCY, whereas a SERVICE LOCATOR is supposed to provide instances for every DEPENDENCY you might request. The differences are subtle, so be sure to fully understand when to apply AMBIENT CONTEXT before you do so. When in doubt, pick one of the other DI patterns.

In section 4.4.4, I'll implement a `TimeProvider` that can be used get the current time, and I'll also discuss why I prefer that to the static `DateTime` members. The current time is a true CROSS-CUTTING CONCERN because you can't predict which classes in which layers may need it. Most classes could conceivably use the current time, but only a small fraction are going to do so.

This could potentially force you to write a lot of code with an extra `TimeProvider` parameter, because you never know when you're going to need it:

```
public string GetSomething(SomeService service,
    TimeProvider timeProvider)
{
    return service.GetStuff("Foo", timeProvider);
}
```

The previous method passes the `TimeProvider` parameter on to the service. That may look innocuous, but when we then review the `GetStuff` method, we discover that it's never being used:

```
public string GetStuff(string s, TimeProvider timeProvider)
{
    return this.Stuff(s);
}
```

In this case, the `TimeProvider` parameter is being passed along as extra baggage only because it might be needed some day. This is polluting the API with irrelevant concerns and a big code smell.

AMBIENT CONTEXT *can* be the solution to this challenge, provided the conditions listed in table 4.4 are met.

Table 4.4 Conditions for implementing AMBIENT CONTEXT

Condition	Description
You need the context to be queryable.	If you only need to write some data (all methods on the context would return void), INTERCEPTION is a better solution. This may seem like a rare case to you, but it's quite common: log that something happened, record performance metrics, assert that the security context is uncompromised—all such actions are pure *Assertions*[9] that are better modeled with INTERCEPTION. You should only consider using an AMBIENT CONTEXT if you need to query it for some value (like the current time).
A proper LOCAL DEFAULT exists.	The existence of an AMBIENT CONTEXT is implicit (more on this to follow), so it's important that the context *just works*—even in the cases where it was never explicitly assigned.
It must be guaranteed available.	Even with a proper LOCAL DEFAULT, it's still important to ensure that it's impossible to assign null, which would make the context unavailable and all clients throw `NullReferenceExceptions`. Listing 4.11 shows some of the steps you can take to ensure this.

In most cases, the advantages of AMBIENT CONTEXT don't justify the disadvantages, so make sure that you can satisfy all of these conditions, and if you can't, consider other alternatives.

Table 4.5 AMBIENT CONTEXT advantages and disadvantages

Advantages	Disadvantages
Doesn't pollute APIs Is always available	Implicit Hard to implement correctly May not work well in certain runtimes

By far the greatest disadvantage of AMBIENT CONTEXT is its implicitness, but, as listing 4.11 suggests, it can also be hard to implement correctly, and there may even be issues with certain runtime environments (ASP.NET).

In the next sections, we'll take a more detailed look at each of the disadvantages in table 4.5.

[9] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (New York: Addison-Wesley, 2004), 255.

IMPLICITNESS

When an AMBIENT CONTEXT is in play, it's impossible to tell whether a given class uses it just by looking at its interface.

Consider the class shown in figure 4.9: it shows no outward sign of using an AMBIENT CONTEXT, yet the `GetMessage` method is implemented like this:

```
public string GetMessage()
{
    return SomeContext.Current.SomeValue;
}
```

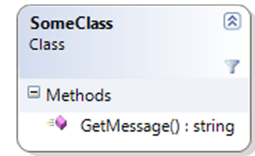


Figure 4.9 The class and its `GetMessage` method show no outward sign of using an AMBIENT CONTEXT, yet this may very well be the case.

When the AMBIENT CONTEXT is correctly implemented, you can at least expect that no exceptions will be thrown, but in this example, the context impacts the behavior of the method because it determines the return value. If the context changes, the behavior may change, and you may not initially understand why this is the case.

> **NOTE** In *Domain-Driven Design,* Eric Evans discusses *Intention-Revealing Interfaces,*[10] which captures the notion that an API should communicate what it does by its public interface alone. When a class uses an AMBIENT CONTEXT it does exactly the opposite: your only chances of knowing that this is the case are by reading the documentation or perusing the code itself.

Apart from the potential for subtle bugs, this implicitness also makes it hard to discover a class's extensibility points. An AMBIENT CONTEXT enables you to inject custom behavior into any class that uses it, but it's not apparent that this may be so. You can only discover this by reading the documentation or understanding the implementation in far more detail than you might have wanted.

IMPLEMENTATION IS TRICKY

Properly implementing an AMBIENT CONTEXT can be challenging. At the very least, you must guarantee that the context is always in a consistent state—that is, it must not throw any `NullReferenceExceptions` only because one context implementation was removed without replacing it with another.

To ensure that, you must have a suitable LOCAL DEFAULT which can be used if no other implementation was explicitly defined. In listing 4.11, I used lazy initialization of the `Current` property, because C# doesn't enable thread-static initializers.

When the AMBIENT CONTEXT represents a truly universal concept, such as time, you can get by with a simple writable Singleton[11]—a single instance that's shared across the entire AppDomain. I'll show you an example of this in section 4.4.4.

An AMBIENT CONTEXT can also represent a context that varies by the call stack's context, such as who initiated the request. We see that often in web application and web services, where the same code executes in context of many different users—each on

[10] Evans, *Domain-Driven Design,* 246.

[11] Gamma, *Design Patterns,* 127.

their own thread. In this case, the AMBIENT CONTEXT can have affinity with the currently executing thread and be stored in TLS, as we saw in listing 4.11, but this leads to other issues, particularly with ASP.NET.

CHALLENGES WITH ASP.NET

When an AMBIENT CONTEXT uses TLS, there can be issues with ASP.NET, because it may change threads at certain points in the page lifecycle, and there's no guarantee that anything stored in TLS will be copied from the old to the new thread.

When this is the case, you should use the current `HttpContext` to store request-specific data instead of TLS.

This thread-switching behavior isn't an issue when the AMBIENT CONTEXT is a universally shared instance, because a Singleton is shared across all threads in an `AppDomain`.

4.4.3 Known use

The .NET BCL contains a few AMBIENT CONTEXT implementations.

Security is addressed with the `System.Security.Principal.IPrincipal` interface that's associated with every thread. You can get or set the current principal for the thread with the `Thread.CurrentPrincipal` accessor.

Another AMBIENT CONTEXT based on TLS models the current culture of the thread. `Thread.CurrentCulture` and `Thread.CurrentUICulture` allows you to access and modify the cultural context of the current operation. Many formatting APIs, such as parsing and converting value types, implicitly use the current culture if one isn't explicitly provided.

Tracing provides an example of a universal AMBIENT CONTEXT. The `Trace` class isn't associated with a particular thread, but is truly shared across an entire AppDomain. You can write a trace message from anywhere with the `Trace.Write` method and have it written to any number of `TraceListeners` configured by the `Trace.Listeners` property.

4.4.4 Example: Caching Currency

The `Currency` ABSTRACTION in the sample commerce application from the previous sections is about as chatty an interface as it can be. Every time you want to convert a currency, you call the `GetExchangeRateFor` method that potentially looks up the exchange rate in some external system. This is a flexible API design because you can look up the rate with close to real-time precision if you need it, but in most cases, this won't be necessary and is more likely to become a performance bottleneck.

The SQL Server–based implementation I exhibited in listing 4.10 certainly performs a database query every single time you ask it about an exchange rate. When the application displays a shopping basket, each item in the basket is being converted, so this leads to a database query for every item in the basket even though the rate is unlikely to change from the first to the last item. It would be better to cache the exchange rate for a little while so that the application doesn't need to hit the database about the same rate several times within the same fraction of a second.

Depending on how important it is to have current currencies, the cache timeout can be short or long: cache for a single second or for hours. The timeout should be configurable.

To determine when to expire a cached currency, you need to know how much time went by since the currency was cached, so you need access to the current time. `DateTime.UtcNow` seems like a built-in AMBIENT CONTEXT, but it's not, because you can't assign the time—only query it.

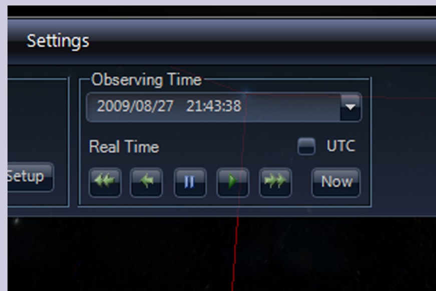
The inability to redefine the current time is rarely an issue in a production application, but can be an issue when unit testing.

Time simulations

Whereas the average web-based application is unlikely to need the ability to modify the current time, another type of application can benefit greatly from this ability.

I once wrote a rather complex simulation engine that depended on the current time. Because I always use Test-Driven Development (TDD), I had already used an ABSTRACTION of the current time so I could inject `DateTime` instances that were different from the actual machine time. This turned out to be a huge advantage when I later needed to accelerate time in the simulation by several orders of magnitude. All I had to do was to register a time provider that accelerated time, and the entire simulation immediately sped up.

If you want to see a similar feature in effect, you can take a look at the WorldWide Telescope[12] client application that allows you to simulate the night sky in accelerated time. The figure below shows a screen shot of the control that allows you to run time forward and backward at different speeds. I have no idea whether the developers behind that particular feature implemented it by using an ambient time provider, but that's what I would do.



WorldWide Telescope allows you to pause time or move forward or backward in time at different speeds. This simulates how the night sky looks at different times.

In the case of the sample commerce application, I want to be able to control time when I write unit tests so that I can verify that the cached currencies expire correctly.

[12] http://www.worldwidetelescope.org

TIMEPROVIDER

Time is a pretty universal concept (even if time moves at different speeds in different parts of the universe), so I can model it as a generally shared resource. Because there's no reason to have separate time providers per thread, the `TimeProvider` AMBIENT CONTEXT is a writable Singleton, as shown in the following listing.

Listing 4.12 `TimeProvider` AMBIENT CONTEXT

```
public abstract class TimeProvider
{
    private static TimeProvider current;

    static TimeProvider()
    {
        TimeProvider.current =
            new DefaultTimeProvider();
    }

    public static TimeProvider Current
    {
        get { return TimeProvider.current; }
        set
        {
            if (value == null)
            {
                throw new ArgumentNullException("value");
            }
            TimeProvider.current = value;
        }
    }

    public abstract DateTime UtcNow { get; }

    public static void ResetToDefault()
    {
        TimeProvider.current =
            new DefaultTimeProvider();
    }
}
```

❶ Initialize to default TimeProvider

❷ Guard Clause

❸ The important part

The purpose of the `TimeProvider` class is to enable you to control how time is communicated to clients. As described in table 4.4, a LOCAL DEFAULT is important, so you statically initialize the class to use the `DefaultTimeProvider` class (I'll show you that shortly) ❶.

Another condition from table 4.4 is that you must guarantee that the TimeProvider can never be in an inconsistent state. The `current` field must never be allowed to be null, so a Guard Clause guarantees that this isn't possible ❷.

All of this is scaffolding to make the `TimeProvider` easily accessible from anywhere. Its raison d'être is its ability to serve `DateTime` instances representing the current time ❸. I purposefully modeled the name and signature of the abstract property after `DateTime.UtcNow`. If necessary, I could also have added such abstract properties as `Now` and `Today`, but I don't need them for this example.

Having a proper and meaningful LOCAL DEFAULT is important, and luckily it's not hard to think of one in this example because it should simply return the current time. That means that, unless you explicitly go in and assign a different `TimeProvider`, any client using `TimeProvider.Current.UtcNow` will get the real current time.

The implementation of `DefaultTimeProvider` can be seen in the following listing.

Listing 4.13 Default time provider

```
public class DefaultTimeProvider : TimeProvider
{
    public override DateTime UtcNow
    {
        get { return DateTime.UtcNow; }
    }
}
```

The `DefaultTimeProvider` class derives from `TimeProvider` to provide the real time any time a client reads the `UtcNow` property.

When `CachingCurrency` uses the `TimeProvider` AMBIENT CONTEXT to get the current time, it will get the real current time unless you specifically assign a different `TimeProvider` to the application—and I only plan to do this in my unit tests.

CACHING CURRENCIES

To implement cached currencies, you're going to implement a Decorator that modifies a "proper" `Currency` implementation.

> **NOTE** The Decorator[13] design pattern is an important part of INTERCEPTION; I'll discuss it in greater detail in chapter 9.

Instead of modifying the existing SQL Server–backed `Currency` implementation shown in listing 4.10, you'll wrap the cache around it and only invoke the real implementation if the cache has expired or doesn't contain an entry.

As you may recall from section 4.1.4, a `CurrencyProvider` is an abstract class that returns `Currency` instances. A `CachingCurrencyProvider` implements the same base class and wraps the functionality of a contained `CurrencyProvider`. Whenever it's asked for a `Currency`, it returns a `Currency` created by the contained `CurrencyProvider`, but wrapped in a `CachingCurrency` (see figure 4.10).

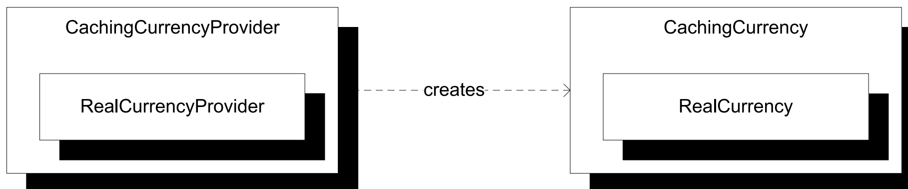


Figure 4.10 A `CachingCurrencyProvider` wraps a "real" `CurrencyProvider` and returns `CachingCurrency` instances that wrap "real" Currency instances.

[13] Gamma, *Design Patterns*, 175.

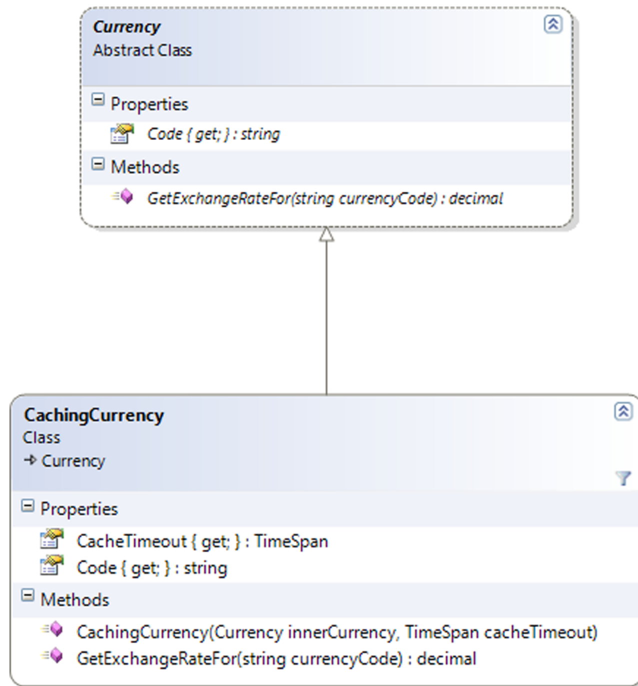


Figure 4.11 `CachingCurrency` **takes an inner currency and a cache timeout in its constructor and wraps the inner currency's functionality.**

> **TIP** The Decorator pattern is one of the best ways to ensure Separation of Concerns.

This design enables me to cache *any* currency implementation, and not only the SQL Server–based implementation I currently have. Figure 4.12 shows the outline of the `CachingCurrency` class.

`CachingCurrency` uses CONSTRUCTOR INJECTION to get the "real" instance whose exchange rates it should cache. For example, `CachingCurrency` delegates its `Code` property to the inner `Currency`'s `Code` property.

The interesting part of the `CachingCurrency` implementation is its `GetExchangeRateFor` method exhibited in the following listing.

Listing 4.14 Caching the exchange rate

```
private readonly Dictionary<string, CurrencyCacheEntry> cache;

public override decimal GetExchangeRateFor(string currencyCode)
{
    CurrencyCacheEntry cacheEntry;
    if ((this.cache.TryGetValue(currencyCode,
        out cacheEntry))
        && (!cacheEntry.IsExpired))
    {
        return cacheEntry.ExchangeRate;
    }
```

❶ **Return cached exchange rate if appropriate**

```
    var exchangeRate =
        this.innerCurrency
        .GetExchangeRateFor(currencyCode);

    var expiration =
        TimeProvider.Current.UtcNow + this.CacheTimeout;
    this.cache[currencyCode] =
        new CurrencyCacheEntry(exchangeRate, expiration);    ❷ Cache exchange rate

    return exchangeRate;
}
```

When a client asks for an exchange rate, you first intercept the call to look up the currency code in the cache. If there's an unexpired cache entry for the requested currency code, you return the cached exchange rate and the rest of the method is skipped ❶. I'll get back to the part about evaluating whether the entry has expired a bit later.

Only if there was no unexpired cached exchange rate do you invoke the inner `Currency` to get the exchange rate. Before you return it, you need to cache it. The first step is to calculate the expiration time, and this is where you use the `TimeProvider` AMBIENT CONTEXT, instead of the more traditional `DateTime.Now`. With the expiration time calculated, you can now cache the entry ❷ before returning the result.

Calculating whether a cache entry has expired is also done using the `TimeProvider` AMBIENT CONTEXT:

```
return TimeProvider.Current.UtcNow >= this.expiration;
```

The `CachingCurrency` class uses the `TimeProvider` AMBIENT CONTEXT in all places where it needs the current time, so writing a unit test that precisely controls time is possible.

MODIFYING TIME

When unit testing the `CachingCurrency` class, you can now accurately control how time seems to pass totally irrespective of the real system clock. That enables you to write deterministic unit tests even though the System Under Test (SUT) depends on the concept of the current time. The next listing shows a test that verifies that even though the SUT is asked for an exchange rate four times, only twice is the inner currency invoked: at the first call, and again when the cache expires.

Listing 4.15 Unit testing that a currency is correctly cached and expired

```
[Fact]
public void InnerCurrencyIsInvokedAgainWhenCacheExpires()
{
    // Fixture setup
    var currencyCode = "CHF";
    var cacheTimeout = TimeSpan.FromHours(1);

    var startTime = new DateTime(2009, 8, 29);
```

```
    var timeProviderStub = new Mock<TimeProvider>();
    timeProviderStub
        .SetupGet(tp => tp.UtcNow)
        .Returns(startTime);
    TimeProvider.Current = timeProviderStub.Object;     <-- ① Set TimeProvider Ambient Context

    var innerCurrencyMock = new Mock<Currency>();
    innerCurrencyMock
        .Setup(c => c.GetExchangeRateFor(currencyCode))
        .Returns(4.911m)
        .Verifiable();

    var sut =
        new CachingCurrency(innerCurrencyMock.Object,
            cacheTimeout);
    sut.GetExchangeRateFor(currencyCode);    <-- ② Should call inner currency
    sut.GetExchangeRateFor(currencyCode);    ③ Should be cached
    sut.GetExchangeRateFor(currencyCode);

    timeProviderStub
        .SetupGet(tp => tp.UtcNow)
        .Returns(startTime + cacheTimeout);   ④ Advance time past timeout
    // Exercise system
    sut.GetExchangeRateFor(currencyCode);    <-- ⑤ Should call inner currency
    // Verify outcome
    innerCurrencyMock.Verify(
        c => c.GetExchangeRateFor(currencyCode),
        Times.Exactly(2));                  ⑥ Verify that inner currency was invoked correctly
    // Teardown (implicit)
}
```

> **JARGON ALERT** The following text contains some unit testing terminology—I have emphasized it with italics, but because this isn't a book about unit testing, I'll refer you to the book *xUnit Test Patterns*[14] that is the source of all these pattern names.

One of the first things to do in this test is to set up a `TimeProvider` *Test Double* that will return `DateTime` instances as defined, instead of based on the system clock. In this test, I use a dynamic mock framework called Moq[15] to define that the `UtcNow` property should return the same `DateTime` until told otherwise. When defined, this *Stub* is injected into the AMBIENT CONTEXT ①.

The first call to `GetExchangeRateFor` should invoke the `CachingCurrency`'s inner `Currency`, because nothing has yet been cached ②, whereas the two next calls should return the cached value ③, because time is currently not passing at all according to the `TimeProvider` *Stub*.

With a couple of calls cached, it's now time to let time advance; you modify the `TimeProvider` *Stub* to return a `DateTime` instance that's exactly past the cache timeout ④ and invoke the `GetExchangeRateFor` method again ⑤, expecting it to invoke

14 Gerard Meszaros, *xUnit Test Patterns: Refactoring Test Code* (New York: Addison-Wesley, 2007).
15 http://code.google.com/p/moq/

the inner `Currency` for the second time because the original cache entry should now have expired.

Because you expect the inner `Currency` to have been invoked twice, you finally verify that this was the case by telling the inner `Currency` *Mock* that the `GetExchangeRateFor` method should have been invoked exactly twice ❻.

One of the many dangers of AMBIENT CONTEXT is that once it's assigned, it stays that way until modified again, but due to its implicit nature, this can be easy to forget. In the unit test, for example, the behavior defined by the test in listing 4.15 stays like that unless explicitly reset (which I do in a *Fixture Teardown*). This could lead to subtle bugs (this time in my test code) because that would spill over and pollute the tests that execute after that test.

AMBIENT CONTEXT looks deceptively simple to implement and use, but can lead to many difficult-to-locate bugs. There's a place for it, but use it only where no better alternative exists. It's like horseradish: great for certain things, but definitely not universally applicable.

4.4.5 Related patterns

AMBIENT CONTEXT can be used to model a CROSS-CUTTING CONCERN, although it requires that we have a proper LOCAL DEFAULT.

If it turns out that the DEPENDENCY isn't a CROSS-CUTTING CONCERN after all, you should change the DI strategy. If you still have a LOCAL DEFAULT you can switch to PROPERTY INJECTION, but otherwise, you must change to CONSTRUCTOR INJECTION.

4.5 Summary

The patterns presented in this chapter are a central part of DI. Armed with a COMPOSITION ROOT and an appropriate mix of the DI patterns, you can implement POOR MAN'S DI. When applying DI, there are many nuances and fine details to learn, but the patterns cover the core mechanics that answer the question, *how do I inject my Dependencies?*

These patterns aren't interchangeable. In most cases, your default choice should be CONSTRUCTOR INJECTION, but there are situations where one of the other patterns affords a better alternative. Figure 4.12 shows a decision process that can help you decide on a proper pattern, but if in doubt, choose CONSTRUCTOR INJECTION—you can never go horribly wrong with that choice.

The first thing to examine is whether the DEPENDENCY is something you need or something you already have but wish to communicate to another collaborator. In most cases, you probably need the DEPENDENCY, but in add-in scenarios, you may wish to convey the current context to an add-in. Every time the DEPENDENCY may vary from operation to operation, METHOD INJECTION is a good candidate for an implementation.

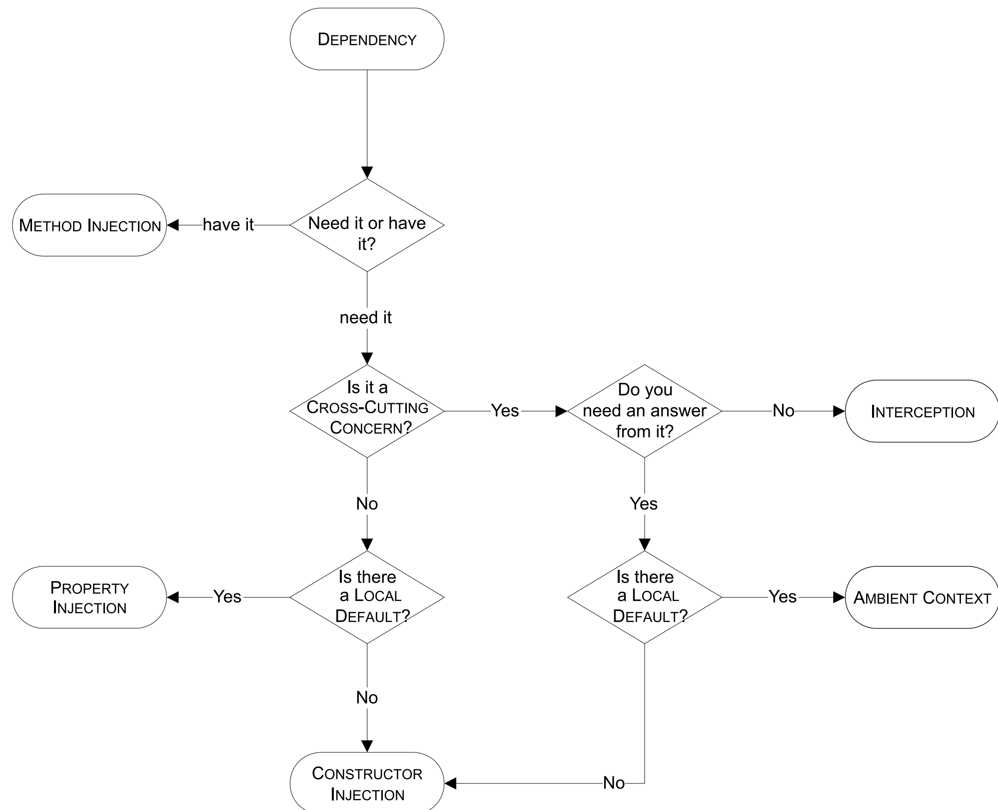


Figure 4.12 In most cases, you should end up choosing CONSTRUCTOR INJECTION, but there are situations where one of the other DI patterns is a better fit.

When the DEPENDENCY represents a CROSS-CUTTING CONCERN, the best pattern fit depends on the direction of communication. If you only need to record something (for example, the length of time an operation took, or what values were being passed in) INTERCEPTION (which I'll discuss in chapter 9) is the best fit. It also works well if the answer you need from it is already included in the interface definition. Caching is an excellent example of this latter use of INTERCEPTION.

If you need to query the CROSS-CUTTING DEPENDENCY for a response *not* included in the original interface, you can use AMBIENT CONTEXT only if you have a proper LOCAL DEFAULT that enables you to package the context itself with a reasonable default behavior that works for all clients without explicit configuration.

When the DEPENDENCY doesn't represent a CROSS-CUTTING CONCERN, a LOCAL DEFAULT is once more the deciding factor, as it can make explicitly assigning the DEPENDENCY optional—the default takes over if no overriding implementation is specified. This scenario can be effectively implemented with PROPERTY INJECTION.

In any other cases, the CONSTRUCTOR INJECTION pattern applies. As illustrated in figure 4.12, it looks as though CONSTRUCTOR INJECTION is a last-ditch pattern that only comes into play when all else fails. This is only partly true, because in most cases the specialized patterns don't apply, and by default CONSTRUCTOR INJECTION is the pattern left on the field. It's easy to understand and much simpler to implement robustly than any of the other DI patterns. You can build entire applications with CONSTRUCTOR INJECTION alone, but knowing about the other patterns can help you choose wisely in the few cases where it doesn't fit perfectly.

This chapter contained a systematic catalog that explained how you should inject DEPENDENCIES into your classes. The next chapter approaches DI from the opposite direction and takes a look at how not to go about it.

.NET/C#

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

ISBN 13: 978-1-935182-50-4
ISBN 10: 1-935182-50-1

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]