

Dependency Injection

With examples in
Java, Ruby, and C#

Dhanji R. Prasanna



Unedited Draft



 MANNING



**MEAP Edition
Manning Early Access Program**

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Contents

- Chapter 1 Dependency Injection: What's all the hype?
- Chapter 2 Time for some Injection
- Chapter 3 Investigating Dependency Injection
- Chapter 4 Building modular applications
- Chapter 5 Scope: objects and their state
- Chapter 6 Advanced scoping and pitfalls
- Chapter 7 From birth to death: object lifecycle
- Chapter 8 Managing your object's behavior
- Chapter 9 Designing managed objects
- Chapter 10 DI Integration with other frameworks
- Chapter 11 Dependency Injection in Action!

Dependency Injection: what's all the hype?

"We all agree that your theory is crazy, but is it crazy enough?"

-- Niels Bohr

In this chapter you will

- See an object as a service
- Learn about building and assembling services
- Take a tour of various pre-existing solutions
- Investigate a technique called the Hollywood Principle
- Briefly survey available frameworks

So you're an expert on Dependency Injection, you know it and use it every day; it's like your morning commute to work--you sleepwalk through it, making all the right left turns (and the occasional wrong right turns, but quickly correcting) until you're comfortably sitting behind your desk at work. Or you've heard of DI and Inversion of Control and read the occasional article, in which case this is your first commute to work on a new job and you're waiting at the station, with a strong suspicion about which train to get on, and with an even stronger suspicion that you're on the wrong platform.

Or you're somewhere in-between, you're feeling your way through the idea, not fully convinced about DI yet: planning out that morning commute and looking for the best route to work, *map-questing* it... Or you've got your own home-brew setup that works just fine thank you very much; you've no need of a DI technology: you bike to work, get a lot of exercise on the way and are carbon-efficient...

STOP!

Take a good, long breath. Dependency Injection is the art of making work come home to you.

1.1 Every solution needs a problem

Most software written today is written to automate some real world process: whether it be the writing of a letter, purchasing the new album from your favorite band; or placing an order to sell some stock. In Object-Oriented programming, these are *objects* and their interactions are *methods*.

Objects represent their real-world counterpart. An `Airplane` represents a 747 and a `Car` represents a Toyota; a `PurchaseOrder` represents you buying this book; and so on.

Of particular interest, is the interaction between objects: An airplane flies, while a car can be driven and a book can be opened and read. This is really where the value of the automation is realized; and where it is valuable in simplifying our lives.

Take the familiar activity of writing an email; you compose the message using an email application (like *Mozilla Thunderbird* or *GMail*). And send it across the internet to one or more recipients as in figure 1.1. This entire activity can be modeled as the interaction of various objects.



Figure 1.1 Email is composed locally, delivered across an Internet Relay and received by an Inbox

This highlights an important precept in this book: the idea of an *object acting as a service*. In the above case, email acts as a message composition service, internet relays are delivery agents and my correspondent's inbox, is a receiving service.

1.1.1 Seeing Objects as Services

The process of emailing a correspondent can be reduced to composing, delivering and receiving of email by each responsible object, namely the `Emailer`, `InternetRelay` and `RecipientInbox`. Each object is a *client* of the next.

`Emailer` uses the `InternetRelay` as a service to *send* email, and in turn, the `InternetRelay` uses the `RecipientInbox` as a service to *deliver* sent mail.

Furthermore, the act of composing an email can itself be reduced into more granular tasks:

- Writing the message
- Checking spelling
- Looking up a recipient's address
- ...and so on. Each of these is a fairly specialized task, and are modeled as specific services. For example, "writing the message" falls into the domain of editing text, so choosing a `TextEditor` is appropriate. Modeling the `TextEditor` in this fashion has many advantages over extending `Emailer` to write text messages itself: We know exactly where to look if we want to find out what the logic looks like for editing text
- Our `Emailer` is not cluttered with distracting code meant for text manipulation
- We can reuse the `TextEditor` component in other scenarios (say, a calendar or note-taking application) without much additional coding
- Conversely, if someone else has written a general-purpose text editing component, we can make use of it rather than writing one from scratch

Similarly, "checking spelling" is done by a `SpellChecker`. If we wanted to check spelling in a different language, it would not be difficult to swap out the English `SpellChecker` in favor of a French one. `Emailer` itself would not need to worry about checking spelling, French, English or otherwise.

Right, so now we've seen the value of decomposing our services into objects. This principle is important because it highlights the relationship between one object and others it uses to perform a service: an object depends on its services to perform a function.

In our example, the `Emailer` depends on a `SpellChecker`, a `TextEditor`, and an `AddressBook`. This relationship is called a *dependency*. In other words, `Emailer` is a client of its dependencies.

Composition also applies transitively; so an object may depend on other objects who themselves have dependencies, and so on. In our case, `SpellChecker` may depend on a `Parser` to recognize words and a `Dictionary` of valid words. `Parser` and `Dictionary` may themselves depend on other objects.

This composite system of dependencies is commonly called an *object graph*. This object graph, though composed of many dependencies, is functionally a single *unit*.

Let's sum up what we have so far:

- **Service** - An object that performs a *well-defined* function when called upon.
- **Client** - Any consumer of a service; an object that calls upon a service to perform a *well-understood*

function.

The service and client relationship implies a clear *contract* between the objects in the role of performing a specific function that is formally understood by both entities. You will also hear them referred to as:

- **Dependency** - A specific service that is *required* by another object to fulfill its function.
- **Dependent** - A client object that needs a dependency (or dependencies) in order to perform its function.

Not only can you describe an object graph as a system of discrete services and clients, but you also begin to see that a client cannot function *without* its services. In other words, an object cannot function properly without its dependencies.

NOTE

.Dependency Injection as a subject is primarily concerned with reliably and efficiently building such object graphs; and the strategies, patterns and best practices therein.

Let's look at some ways of building object graphs before we take on dependency injection.

1.2 Pre-DI Solutions

Here is a simple relationship between an object and its dependency (see figure 1.2):

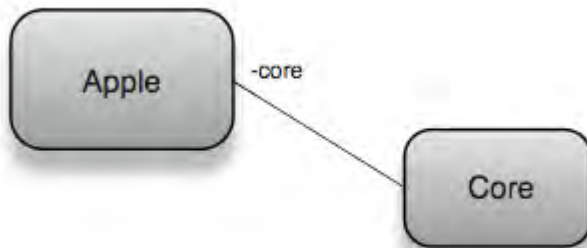


Figure 1.2 Object `Emailer` is composed of another object, `SpellChecker`

Were you asked to code such a class without any other restrictions you might attempt something like this:

```
public class Emailer {
    private SpellChecker spellChecker;
    public Emailer() {
        this.spellchecker = new SpellChecker();
    }

    public void send(String text) { .. }
}
```

Then constructing a working `Emailer` (one with a `SpellChecker`) is as simple as constructing an `Emailer` itself:

```
Emailer emailer = new Emailer();
```

No doubt you have written code like this at some point. I certainly have. Now, let's say I want to write a unit test for the `send()` method to ensure that it is checking spelling before sending any message. How would you do it? You might create a *mock* `SpellChecker` and give that to the `Emailer`. Something like:

```
public class MockSpellChecker {
```

```

private boolean didCheckSpelling = false;
public boolean checkSpelling(String text) {
    didCheckSpelling = true;
}

public boolean verifyDidCheckSpelling() { return didCheckSpelling; } #1
}

```

(Annotation) <#1 Called by test to verify behavior>

Of course, you can't use this mock because we are unable to substitute the internal spelling checker that an `Emailer` has. This effectively makes your class *untestable*. Which is a show-stopper for this approach. Furthermore, this approach also prevents us from creating objects of the same class with different behaviors. See listing 1.1 and figure 1.3.

Listing 1.1; An email service that checks spelling in English.

```

public class Emailer {
    private SpellChecker spellChecker;
    public Emailer() {
        this.spellChecker = new EnglishSpellChecker();
    }
}

```

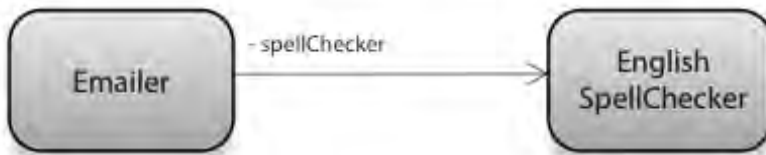


Figure 1.3 An email service that checks spelling in English

In this example, the `Emailer` has an `English` spelling checker. Can we create an `Emailer` with a `FrenchSpellChecker`? We can't! Because `Emailer` encapsulates the *creation* of its dependencies. What we need is a more flexible solution: *construction by-hand* (sometimes called "manual" Dependency Injection); where instead of a dependent creating its own dependencies, it has them provided externally.

1.2.1 Construction by-hand

Naturally, the solution is not to encapsulate the creation of dependencies; but what does this mean and to whom do we offload this burden? There are several techniques that answer this problem. In the earlier section, we used the object's constructor to create its dependencies. With a slight modification, we can keep the structure of the object graph, but offload the burden of creating dependencies. Here is such a modification:

```

public class Emailer {
    private SpellChecker spellChecker;
    public void setSpellChecker(SpellChecker spellChecker) {
        this.spellChecker = spellChecker
    }
}

```



Figure 1.4 An email service which checks spelling using an *abstract* spelling service

Notice that I've replaced the constructor that created its own `SpellChecker` with a method that *accepts* a `SpellChecker`. Now, we can construct an `Emailer` and substitute a mock `SpellChecker`:

```

@Test
public void ensureEmailerChecksSpelling() {
    MockSpellChecker mock = new MockSpellChecker();
    Emailer emailer = new Emailer();
    emailer.setSpellChecker(mock);           #1
    emailer.send("Hello there!");

    assert mock.verifyDidCheckSpelling(); #2
}
  
```

(Annotation) <#1 Pass in mock dependency for testing>

(Annotation) <#2 Verify that the dependency was used properly>

Similarly, it is trivial to construct `Emailers` with various behaviors. Here is one for French spelling:

```

Emailer service = new Emailer();
service.setSpellChecker(new FrenchSpellChecker());
  
```

And even one for Japanese,

```

Emailer service = new Emailer();
service.setSpellChecker(new JapaneseSpellChecker());
  
```

Cool! At the time of creating the `Emailer`, it is up to you to provide its dependencies. This allows you to choose particular flavors of its services that suit your needs (French, Japanese, and so on).

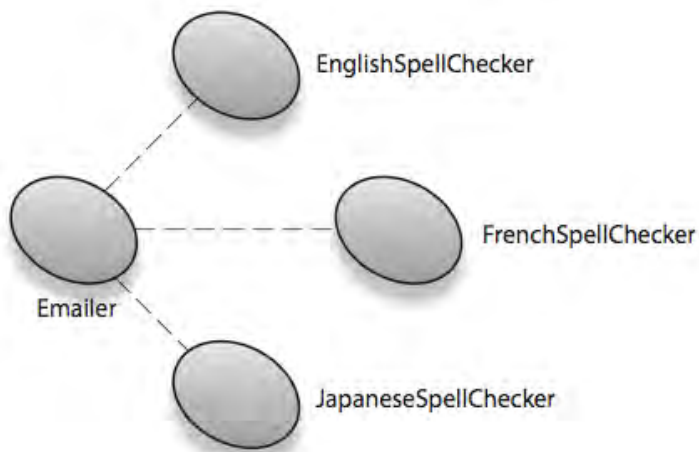


Figure 1.5 The same `Emailer` class can now check spelling in a variety of languages

Since you end up connecting the pipes yourself at the time of construction this technique is referred to as *construction by-hand*. In the previous example we used a setter method (a method that accepts a value and sets it as a dependency). You can also pass in the dependency via a constructor as per the following example:

```
public class Emailer {
    private SpellChecker spellChecker;
    public Emailer(SpellChecker spellChecker) {
        this.spellChecker = spellChecker
    }
}
```

Then creating the Emailer is even more concise:

```
Emailer service = new Emailer(new JapaneseSpellChecker());
```

This technique is called constructor injection and has the advantage of being explicit about its contract--you can never create an Emailer and forget to set its dependencies. As is possible in the earlier example if you forgot to call the `setSpellChecker()` method.. This concept is obviously very useful in object oriented programming. And we'll study it in greater detail in the coming chapters.

NOTE

The idea of connecting the pipes together, or giving a client its dependency is sometimes also referred to as "injecting" objects into one another, and other times as "wiring" the objects together. .

While construction by-hand definitely helps with testing, it also has some problems. The most obvious one being the burden of knowing how to construct object graphs being placed on the client of a service. If I use the same object in many places, I must repeat code for wiring objects in all of those places. Construction by-hand, as the name suggests, is really tedious! If you alter the dependency graph or any of its parts then you may be forced to go through and alter all of its clients as well. Consequently, fixing even a small bug can mean changing vast amounts of code.

Another grave problem is the fact that users need to know how object graphs are wired internally. This violates the principle of encapsulation and becomes problematic when dealing with code that is used by many clients who really shouldn't have to care about the internals of their dependencies in order to use them. There are also potent arguments against construction by-hand that we will encounter in other forms in the coming chapters. So how can we offload the burden of dependency creation and not shoot ourselves in the foot doing so? One answer is the *factory pattern*.

1.2.2 The Factory Pattern

Another time-honored method of constructing object graphs is the Factory design pattern (also known as the Abstract Factory¹ pattern). The idea behind the factory pattern is to offload the burden of creating dependencies to a third party object called a factory (shown in figure 1.3). Just as an automotive factory creates and assembles cars, so too does an abstract factory create and assemble object graphs.

¹ Eric Gamma et al. GoF

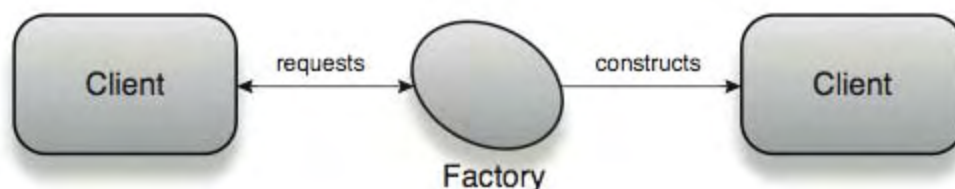


Figure 1.6 A client requests its dependencies from a factory

Let's apply the factory pattern to our `Emailer`. The `Emailer`'s code remains the same (as shown in listing 1.2).

Listing 1.2 An email service whose spell checker is set via constructor

```

public class Emailer {
    private SpellChecker spellChecker;
    public Emailer(SpellChecker spellChecker) {
        this.spellChecker = spellChecker
    }
}
  
```

However, instead of constructing the object graph by hand, we do it inside another class called a factory (as per listing 1.3).

Listing 1.3 A "French" email service factory

```

public class EmailerFactory {
    public Emailer newFrenchEmailer() {
        return new Emailer(new FrenchSpellChecker());
    }
}
  
```

Notice that the factory is very explicit about what kind of `Emailer` it is going to produce; `newFrenchEmailer()` creates one with a French spelling checker. Any code that uses French email services is now fairly straightforward:

```

Emailer service = new EmailerFactory().newFrenchEmailer();
  
```

The most important thing to notice here is that the client code has no reference to spell checking, address books, or any of the other *internals* of `Emailer`. By adding a level of abstraction (the factory), we have separated the code using the `Emailer` from the code that *creates* the `Emailer`. This leaves client code clean and concise.

The value of this becomes more apparent as we deal with richer and more complex object graphs. Listing 1.5 shows a factory that constructs our `Emailer` with many more dependencies.

Listing 1.5 A factory that constructs an email service with a Japanese spell checker, text editor

```

public class EmailerFactory
{
    public Emailer newJapaneseEmailer() {
        Emailer service = new Emailer();
        service.setSpellChecker(new JapaneseSpellChecker());
        service.setAddressBook(new EmailBook());
        service.setTextEditor(new SimpleJapaneseTextEditor());
    }
}
  
```

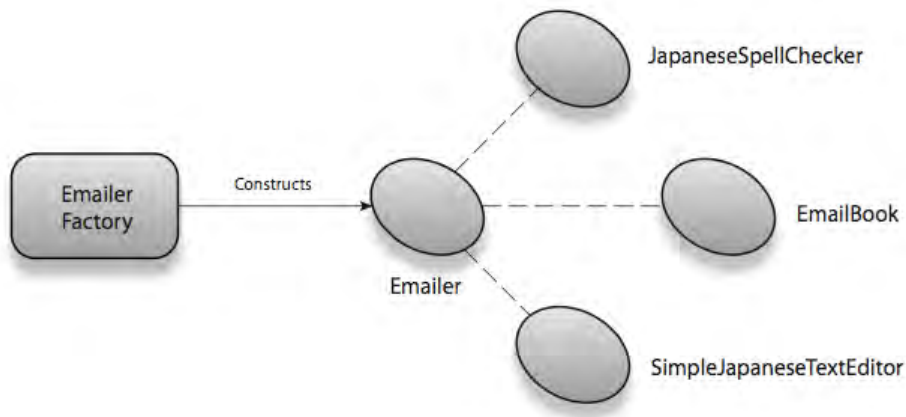


Figure 1.7 EmailerFactory constructs and assembles a *Japanese* emailer with various dependencies

And code that *uses* such an Emailer is as simple and readable as we could wish:

```
Emailer emailer = new EmailerFactory().newJapaneseEmailer();
```

The beauty of this approach is that client code only needs to know which factory to use to obtain a dependency (and none of its internals).

Now how about testing this code? Are we able to mock Emailer's dependencies? Sure, our test can simply ignore the factory and pass in mocks:

```

@Test
public void testEmailer() {
    MockSpellChecker spellChecker = new MockSpellChecker(); |#1
    ... |#1

    Emailer emailer = new Emailer();
    emailer.setSpellChecker(spellChecker); |#2
    ... |#2

    emailer.send("Hello there!");

    //verify emailer's behavior
    assert ...; #3
}
  
```

(Annotation) <#1 Create mocks for each dependency>
 (Annotation) <#2 Set mocked dependencies on emailer>
 (Annotation) <#3 Ensure everything worked>

So far so good. Now let's look at a slightly different angle: how can we test *clients* of Emailer? Listing 1.6 for instance.

Listing 1.6 A client uses Emailer to send messages typed by a user

```

public class EmailClient {
    private Emailer emailer = new EmailerFactory().newEnglishEmailer(); #1

    public void run() {
        emailer.send(someMessage()); #2

        confirm("Sent!");
    }
}
  
```

(Annotation) <#1 Get ourselves an Emailer from its factory>
 (Annotation) <#2 Send message and confirm to user>

This client does not know anything about `Emailer`'s internals, instead depending on a factory. If we want to test that it correctly calls `Emailer.send()`, we need to use a mock. Rather than set the dependency directly, must pass in the mock via the factory:

Listing 1.7 Test sets up a mock instance for `EmailClient` using `Emailer`'s factory

```
@Test
public void testEmailClient() {
    MockEmailer mock = new MockEmailer();
    EmailerFactory.set(mock); #1

    new EmailClient().run();

    assert mock.correctlySent(); #2
}
(Annotation) <#1 Set mock instance on factory using a static method>
(Annotation) <#2 Verify that the mock was used correct by EmailClient>
```

In this test, we pass in `MockEmailer` using a static method `EmailerFactory.set()`, which stores and uses the provided object rather than creating new ones:

Listing 1.8 A client uses `Emailer` to send messages typed by a user

```
public class EmailerFactory {
    private static Emailer instance; #1

    public Emailer newEmailer() {
        if (null == instance) #2
            return new Emailer(..);

        return instance;
    }

    static void set(Emailer mock) { #3
        instance = mock;
    }
}
(Annotation) <#1 static holder stores mock instance for later use>
(Annotation) <#2 If no mock is present, create a new Emailer with dependencies>
(Annotation) <#3 Save mock instance to static holder>
```

In Listing 1.8, `EmailerFactory` has been heavily modified to support testing. A test can set up a mock instance via the static `set()` method first, then verify the behavior of any clients (as shown in listing 1.7).

Unfortunately, this is not the complete picture since forgetting to clean up the mock can interfere with other `Emailer`-related tests that run later. So, we must *reset* the factory at the end of every test:

```
@Test
public void testEmailClient() {
    MockEmailer mock = new MockEmailer();
    EmailerFactory.set(mock);

    new EmailClient().run();

    assert mock.correctlySent();
    EmailerFactory.set(null); #1
}
(Annotation) <#1 Reset factory's state to null>
```

We're not quite out of the woods yet. If there is an exception thrown before the factory is reset, it can still leave things in an erroneous state. So, a safer cleanup is required:

```
@Test
public void testEmailClient() {
    MockEmailer mock = new MockEmailer();
    EmailerFactory.set(mock);
    try {
        new EmailClient().run();
    }
```

```

        assert mock.correctlySent();
    } finally {
        EmailerFactory.set(null);
    }
}

```

(Annotation) <#1 Reset factory's state inside a finally block>

Much better. A lot of work to write a simple assertion, but worth it! Or is it? It turns out even this careful approach is insufficient in broader cases where you may want to run tests in parallel. The static mock instance inside `EmailerFactory` can cause these tests to clobber each other from concurrent threads and renders them useless.

NOTE

This is an essential problem with shared state, often portended by the *singleton pattern*. Its effects and solutions are examined more closely in chapter 5.

While the factory pattern solves many of the problems with construction by-hand, it obviously still leaves us with some significant hurdles to overcome. Apart from the testability problem, the fact that a factory must accompany every service is troubling. Not only this, a factory must accompany *every* variation of *every* service. This is a sizable amount of distracting clutter and adds a lot of peripheral code to be tested and maintained. Take a look at a concrete example in listing 1.9 and you will see what I mean.

Listing 1.9 Factories that create either French or Japanese email services

```

public class EmailerFactory {
    public Emailer newJapaneseEmailer() {
        Emailer service = new Emailer();
        service.setSpellChecker(new JapaneseSpellChecker());
        service.setAddressBook(new EmailBook());
        service.setTextEditor(new SimpleJapaneseTextEditor());
    }
    public Emailer newFrenchEmailer() {
        service = new Emailer();
        service.setSpellChecker(new FrenchSpellChecker());
        service.setAddressBook(new EmailBook());
        service.setTextEditor(new SimpleFrenchTextEditor());
    }
}

```

(Annotation) <#A Specific dependencies of a Japanese email service>

(Annotation) <#B Specific dependencies of a French email service >

Now, if you wanted an English version of the `Emailer` you would have to add yet *another* method to the factory. And consider what happens when if we replace `EmailBook` with a `PhoneAndEmailBook`. You are forced to make the following changes:

```

public class EmailerFactory {
    public Emailer newJapaneseEmailer() { ...
        emailer.setAddressBook(new PhoneAndEmailBook());
        ...
    }
    public Emailer newFrenchEmailer() { ...
        emailer.setAddressBook(new PhoneAndEmailBook());
        ...
    }
    public Emailer newEnglishEmailer() { ...
        emailer.setAddressBook(new PhoneAndEmailBook());
        ...
    }
}

```

All three of the changes are identical! It is clearly not a desirable scenario. Furthermore, any client code is at the mercy of available factories: you must create new factories for each additional object graph variation. All this spells reams of additional code to write, test and maintain.

In the case of `Emailer`, following the idea to its inevitable extreme yields:

```

Emailer service = new EmailerFactoryFactory()           #1
                  .newAdvancedEmailerFactory()         #2
                  .newJapaneseEmailerWithPhoneAndEmail(); #3

```

(Annotation) <#1 "General" factory produces simple or advanced factories>
(Annotation) <#2 Factory for advanced Emailers>
(Annotation) <#3 Factory producing specific Emailer configurations>

It is very difficult to test code like this. I have seen it ruin several projects. Clearly, the factory technique's drawbacks are serious, especially in larger and more complex code. What we need is a broader mitigation of the core problem.

1.2.3 The Service Locator Pattern

Very simply, a Service Locator is a kind of factory. It is a third party object responsible for producing a fully constructed object graph.

In the typical case, service locators are used to find services published by external sources; it may be an API offered by a bank to transfer funds or a search interface from Google.

These external sources may reside in the same application, machine or local area network; or they may not. For example: an interface to a NASA Mars Rover several millions of miles away. Or a simple library for text processing, bundled within an application. Take a look at figure 1.8 for a visual.

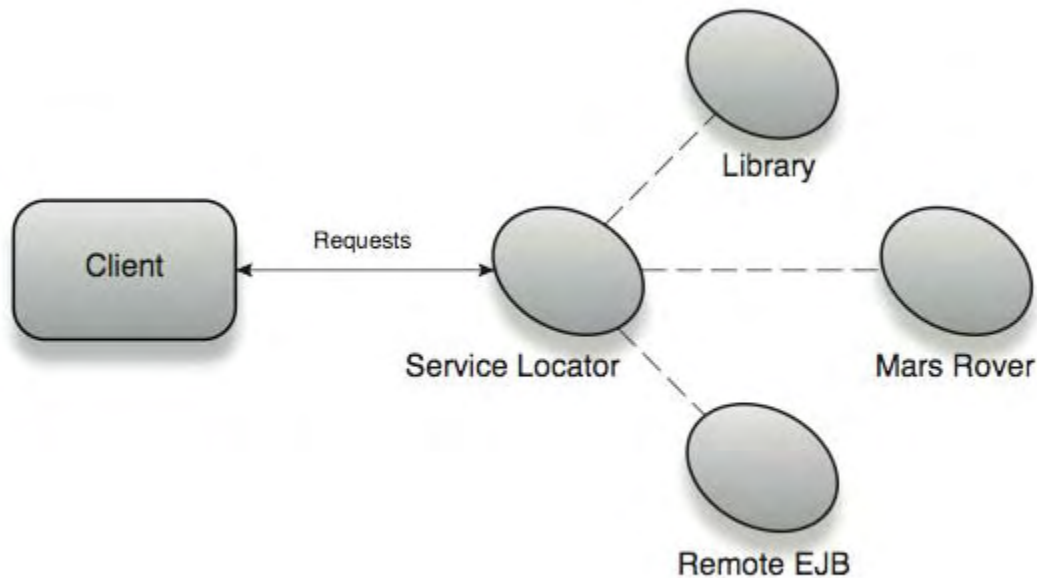


Figure 1.8 Service Locators find specific services requested by clients

Let's look at what a service locator does:

```

Emailer emailer = new ServiceLocator().Get("Emailer");

```

Notice that we pass the service locator a *key*, in this case the word "Emailer". This tells our locator that we want an emailer. This is significantly different from a factory that produced only one kind of service. A service locator is therefore, a factory that can produce *any* kind of service.

Right away this helps reduce a huge amount of repetitive factory code, in favor of the single service locator.

JNDI: A Service Locator

The Java Naming and Directory Interface (JNDI) is a good example of a service locator. It is often used by application servers to register resources at start time and later by deployed applications to look them up. Web applications typically use JNDI to look up data sources in this manner.

Let's apply the service locator pattern to the earlier example:

```
Mailer emailer = new ServiceLocator().Get("JapaneseEmailerWithPhoneAndEmail"); #1  
(Annotation) <#1 Key precisely describes the desired service>
```

This code is simple and readable. The identity of a service (its key) is sufficient to obtain exactly the right service and configuration. Now, altering the behavior of a service identified by a particular key, or fixing bugs within it by changing its object graph will have no effect on dependent code, and can be done transparently.

Unfortunately, being a kind of factory, service locators suffer from the same problems of testability and shared state. The keys used to identify a service are opaque and can be confusing to work with, as anyone who has used JNDI can attest. If a key is bound improperly, then the wrong type of object may be created and this error is found out only at runtime. The practice of embedding information about the service within its identifier (viz., "JapaneseEmailerWithPhoneAndEmail") is also verbose and places too much emphasis on arbitrary conventions.

With dependency injection, we take a completely different approach. One that emphasizes testability and concise code that is easy to read and maintain. However, that is only the beginning; as you will see soon, with dependency injection we can do a great deal more than that.

1.3 Embracing Dependency Injection

With dependency injection, we take the best parts of the aforesaid pre-DI solutions and leave behind their drawbacks: DI enables testability in the same way as construction by-hand, via setter method or constructor injection.

It removes the need for clients to know about their dependencies and how to create them, just as factories do

And leaves behind the problems of shared state and repetitive clutter, by moving construction responsibility to a library

With Dependency Injection clients need know nothing about French or English `Mailers`, let alone French or English `SpellCheckers` and `TextEditors`, in order to use them. This idea of not *explicitly* knowing is central to Dependency Injection. More accurately: not *asking* for dependencies, and instead having them provided to you is an idea called the Hollywood Principle.

1.3.1 The Hollywood Principle

Very simply, the Hollywood Principle is "don't call us, we'll call you!" Just as Hollywood talent agents use this principle to arrange auditions for actors, so do Dependency Injection libraries use this principle to provide objects with what they depend on.

This is similar to what we saw in construction by-hand (sometimes referred to as *manual* dependency injection). However, with one important difference: the task of creating, assembling and wiring the dependencies into an object graph is performed by an external framework (or library) known as a *Dependency Injection framework*, or simply a *Dependency Injector*. Figure 1.9 illustrates this arrangement.

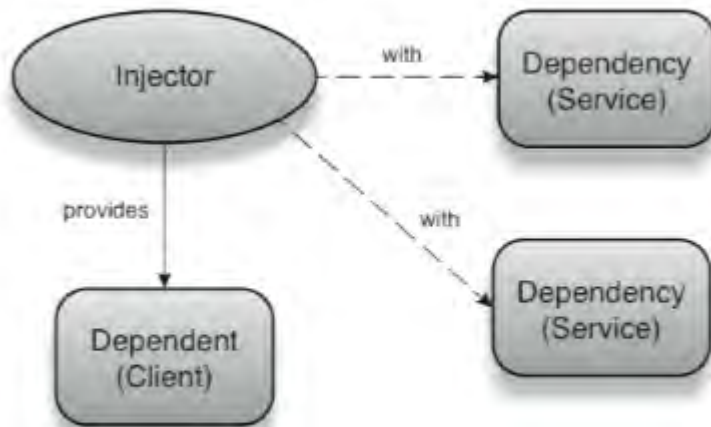


Figure 1.9 The Injector provides an object with its dependencies

Control over the construction, wiring and assembly of an object graph no longer resides with the clients or services themselves. The Hollywood Principle's reversal of responsibilities is sometimes also called *Inversion of Control*.

NOTE

Dependency Injection frameworks are sometimes referred to as Inversion of Control or IoC **Containers**. Examples of such frameworks include: *PicoContainer* (for Java), *StructureMap* (for C#) and *Spring* (for Java and C#).

Listing 1.8 shows the Hollywood Principle in action.

Listing 1.8 An Emailer and a client

```

public class Emailer {
    ...
    public void send(String text) { .. }
}

public class SimpleEmailClient { #1
    private Emailer emailer;
    public SimpleEmailClient(Emailer emailer) {
        this.emailer = emailer;
    }
    public void sendEmail() { #2
        emailer.send(readMessage());
    }
}
  
```

(Annotation) <#1 SimpleEmailClient depends on Emailer>

(Annotation) <#2 Sends an email read from input>

In this example, our dependent is `SimpleEmailClient` ([#1]) and the service it uses ([#2]) is `Emailer`. Notice that neither class is aware of how to construct its graph; nor do they explicitly ask for a service.

Note that in order send mail, `SimpleEmailClient` does not need to expose anything about `Emailer` or how it works. Put another way, `SimpleEmailClient` encapsulates `Emailer` and sending email is completely opaque to the user. Constructing and connecting dependencies is now performed by a dependency injector (see figure 1.6).

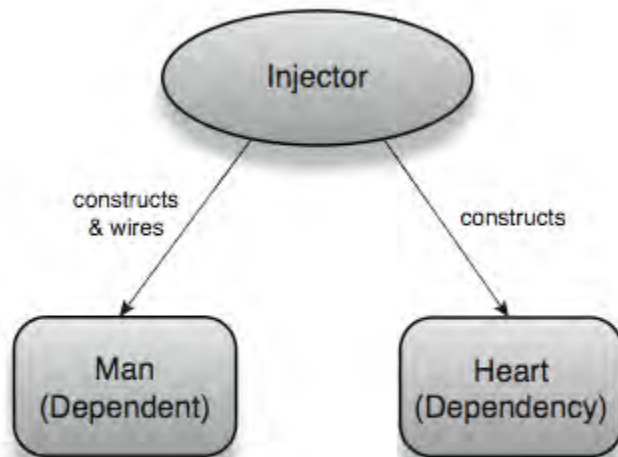


Figure 1.10 The Injector constructs and wires SimpleEmailClient with a dependency (Emailer)

The dependency is shown as a class diagram in figure 1.7, below.



Figure 1.11 Client code that uses an email service provided by dependency injection

Notice that SimpleEmailClient knows nothing about what kind of Emailer it needs or is using to send a message. All it knows is that it accepts *some kind* of Emailer, and this dependency is simply used ([#2]) when needed. You will also notice that the client code is now starting to resemble service code; both are free of logic to create or locate dependencies. Dependency injection facilitates this streamlining, stripping code of distracting clutter and infrastructure logic, leaving purposed, *elementary logic* behind.

Of course, we have not yet seen how the wiring is actually done; as this differs vastly from injector to injector, but what we've seen is still very instructive as it highlights the separation of *infrastructure* code (meant for wiring and construction) from *application* code (the core purpose of a service). The next couple of sections explore this idea, before we jump into the specifics of working with dependency injectors.

1.3.2 Inversion of Control vs. Dependency Injection

Worthy as they are of a heavyweight bout, these two terms are not really opposed to one another as the heading suggests. You will come across the term *Inversion of Control* (or *IoC*) quite often, both in the context of dependency injection and outside it. The phrase Inversion of Control is itself rather vague and

connotes a general reversal of responsibilities which is non-specific and could equally mean any of:

- a module inside a Java EE application server,
- an object wired by a Dependency Injector,
- a test method automatically invoked by a framework, or even
- an event handler called on clicking a mouse button.

Pedantic users of the term suggest that all of these cases are consistent with its definition and that dependency injection itself is simply one instance of IoC.

In common use, dependency injectors are frequently referred to as IoC Containers.

In the interest of clarity, for the rest of this book I will abandon the term IoC (and its evil cousin *IoC Container*) in favor of the following, more precise terms:

- **Hollywood Principle** - the idea that a dependent is *contacted* with its dependencies
- **Dependency Injector** - a framework or library that embodies the Hollywood Principle
- **Dependency Injection** - the range of concerns with designing applications built on these principles.

"Dependency Injection"

As near as I can determine, it wasn't until Martin Fowler wrote, "Inversion of Control Containers and the Dependency Injection Pattern"² that the term *Dependency Injection* came into popular use. The term came out of many discussions on the subject between Martin Fowler, the authors of PicoContainer (including Paul Hammant, Aslak Hellesøy), Jon Tirsén and Rod Johnson among others.

Early frameworks differed by the various forms of wiring that they proffered and promoted. Over the years, the efficacy of setter- and constructor- wiring overtook other forms of wiring; and more flexible solutions that emphasized the safety of contracts and the reduction of *repetitive* code emerged. With the rise of related paradigms such as *Aspect Oriented Programming* (AOP), these features continued to improve. Applications built with DI became streamlined and tolerant to rapid structural and behavioral change.

The modes of configuring a dependency injector also evolved from verbose sets of contracts and configuration files to more concise forms, using new language constructs such as *Annotations* and *Generics*. Dependency injectors also took better advantage of class manipulation tools like *reflection* and *proxying*. And began to exploit design patterns such as *Decorators*³, *Builders*⁴ and *Domain-Specific Languages*.

Domain-specific language (DSL)

A Domain-Specific Language or DSL is a language used to solve a specific problem as opposed to a language like C++ that is intended to solve general (or any) problems. DSLs are sometimes used in Dependency Injection as a precise way of expressing the structural relationship between objects. They are also used to capture other concerns (such as scope) in the *language* of DI. (For more information, see the forthcoming book *Building Domain Specific Languages in Boo* from Manning Publications.)

A growing emphasis on unit-testing was (and continues to be) a natural catalyst to the growth of DI popularity. This made for agile programs, which are easily broken down into discrete, modular *units* of that are simple to test and swap out with alternative behaviors. Consequently, *loose coupling* is a core driver in the evolution of dependency injection.

Loose Coupling

Loose Coupling describes an adaptable relationship between a client and service, where significant changes to the internals of the service have minimal impact on the client. It is generally achieved by placing a rigid contract

² The article has been updated many times over the years. You can read the latest version at <http://martinfowler.com/articles/injection.html>

³ The Decorator Pattern. Design Patterns: Erich Gamma et al.

⁴ The Builder Pattern. Design Patterns: Erich Gamma et al.

between client and service so that either may evolve independently, so long as they fulfill the obligations laid down by the contract. Loose-coupling is explored in greater detail in chapter 4.

We will explore the best approaches to solving problems with DI, and look in detail at bad practices, pitfalls and corner-cases to watch out for and indeed the safety, rigor and power of dependency injection, properly applied. And most of all we'll see how DI can make your code lean, clean and something mean.

1.4 Dependency Injection in the real world

We have now looked at several possible solutions and hinted at a cool alternative called dependency injection. We've also taken an evening stroll down history lane, turning at the corner of terminology boulevard. (And somehow visited Hollywood on the way!) Before we proceed to the nuts and bolts of dependency injection, let's take a quick survey of the landscape to learn what libraries are available; learn a little bit about them and their origins.

This is by no means a comparison or evaluation of frameworks, simply a brief introduction. Neither is it meant to be a comprehensive list of options. I will only touch on relatively well-known and widely-used DI libraries in Java. And only those that are open-source and freely available. Not all of them are purely DI-focused, and very few support the full gamut of dependency injection design patterns described in this book. Nevertheless each is worth a look.

1.4.1 Java

Java is the birthplace of dependency injection and easily sports the broadest and most mature set of DI libraries among all platforms. Since many of the problems DI evolved to address are fundamental to Java, DI's effectiveness and prevalence is especially clear in this platform. We'll look at five such libraries ranging from the earliest incarnations of DI in Apache Avalon, through mature widely-adopted libraries like Spring and PicoContainer and also the cutting-edge, modern incarnation in Google Guice.

APACHE AVALON

Apache Avalon is possibly the earliest DI library in the Java world, and perhaps the first library that really focused on dependency injection as a core competency. Avalon styled itself as a complete application container, in the days before Java EE and application servers were predominant. It's primary mode of wiring was via custom interfaces, not setter methods or constructors. Avalon also supported myriad lifecycle and event management interfaces and was popular with many Apache projects. The *Apache JAMES* mail server (a pure Java SMTP, POP3 and IMAP email server) is built on Avalon.

Avalon is completely defunct now, though Apache James is still going strong. Some of the ideas from Avalon have passed on to another project, Apache Excalibur, whose DI container is named *Fortress*. Neither Avalon nor Excalibur/Fortress are in very common use today.

SPRING FRAMEWORK

Spring Framework is a ground-breaking and cornerstone dependency injection library of the Java world. It is largely responsible for the popularity and evolution of the DI idiom and for a long time was almost synonymous with dependency injection. Spring was created by Rod Johnson and others and was initially meant to solve specific pains in enterprise project. It was established as an open-source project in 2003 and grew rapidly in scope and adoption. Spring provides a vast set of abstractions, modules and points of integration for enterprise, open-source and commercial libraries. It consists of much more than dependency injection, though DI is its core competency. It primarily supports setter and constructor injection and has a variety of options for managing objects created by its dependency injector. For example, it provides support for both the *AspectJ* and *AopAlliance Aspect Oriented Programming* paradigms.

Spring adds functionality, features and abstractions for popular third-party libraries at alarming rates. It is extremely well documented in terms of published reference books as well as online documentation and continues to enjoy widespread growth and adoption.

PICOCONTAINER & NANOCONTAINER

PicoContainer was possibly the first DI library to offer constructor wiring. It was envisioned and built around certain philosophical principles and many of the discussions found on its website are of a theoretical nature. It was built as a lightweight "engine" for wiring components together. In this sense it is well-suited to extensions and customization. This makes PicoContainer useful under-the-covers. PicoContainer supports both setter and

constructor injection, but clearly demonstrates a bias toward the latter. Due to its nuts-and-bolts nature, many people find PicoContainer difficult to use in applications directly. NanoContainer is a sister-project of PicoContainer that attempts to bridge this gap by providing a simple configuration layer, with PicoContainer doing the DI work underneath. NanoContainer also provides other useful extensions.

APACHE HIVEMIND

Apache HiveMind was started by Howard Lewis Ship, the creator of *Apache Tapestry*, a popular *component-oriented* Java web framework. In fact, for a long time Tapestry used HiveMind as its internal dependency injector. Much of the development and popularity of HiveMind has stemmed from this link, though both HiveMind and Tapestry have parted company in later releases. HiveMind originally began at a consulting project of Howard's that involved the development and interaction of several hundreds of services. It was primarily used in managing and organizing these services into common, reusable patterns.

HiveMind offers both setter and constructor injection, and provides some unusual and innovative DI features that are missing in other popular libraries. For example, HiveMind is able to create and manage *pools* of a service for multiple concurrent clients. Apache HiveMind is not as widely used as some other DI libraries discussed here, however it has a staunch and loyal following.

GOOGLE GUICE

Guice (pronounced "Guice!") is a hot new lightweight Dependency Injector created by Bob Lee and others at Google. Guice unabashedly embraces Java 5 language features and strongly emphasizes type and contract safety. It is lightweight and decries verbosity in favor of concise, type-safe and rigorous configuration. Guice is used heavily at Google, particularly in the vast AdWords application and a version of it is at the heart of the *Struts2* web framework. It supports both setter and constructor injection along with some interesting alternatives. Guice's approach to AOP and construction of object graphs is intuitive and simple.

It is a new kid on the block, but its popularity is rising and its innovations have already prompted several followers and even some mainstays to emulate its ideas. Guice has a vibrant and thoughtful user community.

1.4.2 DI in other Languages & Libraries

There are several other DI libraries that I have not mentioned above. You could write a whole book on the subject! Some provide extra features, orthogonal to dependency injection or are simply focused on a different problem space. *JBoss Seam* is one such framework--it offers complex state management for web applications and integration points for standardized services such as EJBs. It also offers a reasonably sophisticated subset of Dependency Injection.

As a language, C# (and .NET) is structurally similar to Java. They are both statically typed, object-oriented languages that need to be compiled. It is no surprise then, that C# is found in the same problem space as Java and that dependency injectors are applied to equal effect in applications written in C#. However, the prevalence of DI is much lower in the C# world in general. While C# has ports of some Java libraries like Spring and PicoContainer, it also has some innovative DI of its own, particularly in Castle MicroKernel, which does a lot more than just DI. StructureMap, on the other hand is a mainstay, and a more traditional DI library.

Some platforms (or languages) make it harder to design dependency injectors because they lack features such as *garbage collection* and *reflection*. C++ is a prime example of such a language. However, it is still possible to write dependency injectors in these languages with other methods that substitute for these tools. For instance, some C++ libraries use *precompilation* and *source code generation* to enable dependency injection.

Still other languages place different (and sometimes less) restrictions on types and expressions, allowing objects to take on a more dynamic role in constructing and using services. Python and Ruby are good examples of such *duck typed* languages. Neither Python nor Ruby programs need to be compiled; their source code is directly *interpreted* and type-checking is done on the fly. Copland is a DI library for Ruby which was inspired by (and is analogous) to Apache HiveMind for Java. Copland uses many of the same terms and principles as HiveMind but is more flexible in certain regards due to the dynamic nature of Ruby.

1.5 Summary

This chapter was an exposition to the subject of building programs in units and should have laid the groundwork for Dependency Injection. Most software is about automating some process in the real world, like sending a friend a message by email or processing a purchase order for some stock. Components in these processes are modeled as a system of objects and methods. When we have vast swathes of components and interactions to manage,

constructing and connecting them together becomes a tedious and complex task. Development time spent on maintenance, refactoring and testing can explode without carefully designed architectures. We can reduce all of this to the rather simple-sounding problem of finding and constructing the dependencies (requisite services) of a component (client). However, as we've seen this is non-trivial when dealing with objects that require different behaviors and does not often scale with typical solutions.

Prior to the use of Dependency Injectors, there were several solutions of varying effectiveness:

- **Construction by-hand:** which involves the client creating and wiring together all of its dependencies. This is a workable solution and certainly conducive to testing, but it scales very poorly as we've merely offloaded the burden of creating and wiring dependencies to clients. It also means clients must know about their dependencies and about *their* implementation details.
- **The Factory Pattern:** clients request a service from an intermediary component known as a factory. This offloads the burden to factories and leaves clients relatively lean. Factories have been very successful over the years, however they pose some very real problems for testing and can introduce more with shared state. in very large projects they can cause an explosion of infrastructure code. Code using factories is difficult to test and can be a problem to maintain.
- **The Service Locator Pattern:** essentially a type of factory, but using keys to identify services and their specific configurations. The Service Locator solves the explosion-of-factories problem by hiding the details of the service and its configuration from client code. Since it is a factory, it is also prone to testability and shared state problems. . Finally, it requires that a client explicitly request each of its dependencies by an arbitrary key, which can be unclear and abstruse.

Dependency Injection offers an innovative alternative via the *Hollywood Principle*: "don't call us, we'll call you!" meaning that a component need know nothing about its dependencies nor explicitly ask for them. A third party library or framework known as the *Dependency Injector* is responsible for constructing, assembling and wiring together clients with services.

DI solves the problem of object graph construction and assembly by applying the Hollywood Principle across components, however this is just a small window into what is possible when an application is designed and built around Dependency Injection. We will explore managing state, modifying behavior and lifecycle of objects in broader contexts, we'll look at the nuances, pitfalls and corner cases of various configurations and the subtle consequences of particular design choices. This book will equip you with the knowledge and careful understanding you need to design robust, testable and easily maintainable applications for any purpose or platform. In the next chapter, you'll get a chance to get your hands dirty. We'll dive right in to using some of the DI libraries surveyed in this chapter and contrast their various approaches. Keep a bar of soap handy!