

Dependency Injection

With examples in
Java, Ruby, and C#

Dhanji R. Prasanna



Unedited Draft

 MANNING





**MEAP Edition
Manning Early Access Program**

Copyright 2007 Manning Publications

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

Table of Contents

Introduction

Chapter 1 Dependency Injection: What's all the hype?

Chapter 2 Inject me already!

Chapter 3 Investigating Dependency Injection

Chapter 4 Building modular applications

Chapter 5 Scope: Objects and their state

Chapter 6 From birth to death: Object Lifecycle

Chapter 7 Managing your object's behavior

Chapter 8 Designing Enterprise Applications with Dependency Injection

Chapter 9 Designing Managed Objects

Chapter 10 DI Integration with other Frameworks

Chapter 11 Choose your poison: Comparing DI frameworks

Appendix A A list of open-source DI frameworks

Appendix B References

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 the problem of building and assembling objects
- Take a tour of various pre-existing solutions, and reason about why they fall short
- Get introduced to an innovative technique called the Hollywood Principle
- Briefly survey available frameworks
- Believe the hype about Dependency Injection!

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

Before we investigate Dependency Injection, it is worth asking what the point of DI is? *Exactly what problem does it solve?*

Most software written today is written to automate (or model) 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. As such, it is often useful to model things in the real world as virtual entities that talk, interact and live among one another in a way that mimics reality (the parts of it that we're interested in anyway). In the Object-Oriented world, we call these virtual entities “objects” and the interactions between them “methods.”

Objects represent and act as their real-world counterpart would. An Airplane object represents a 747 and a Car object represents a Toyota; a PurchaseOrder represents the process of you buying this book; and so on.

Of particular interest, is the act part of an object's responsibility: 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 model is realized; and where it plays its part 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 representative components (i.e. objects).



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

This highlights an important precept that will become very important to us for the rest of this book: the idea of an *object acting as a service*. In the above case, my email service acts as a message composition service, internet relays act as delivery agents and my correspondent's inbox, as a receiver.

1.1.1 Seeing Objects as Services

Another way to see objects as services is to think of them as intermediary agents that perform well-understood functions to fulfill a larger purpose. The process of emailing a correspondent can be reduced to composing, delivering and receiving of email by component agents, namely the **Emailer**, **InternetRelay** and **RecipientInbox**. Each such component 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, so you would model them as specific services. For example, "writing the message" falls into the domain of editing text, so choosing a **TextEditor** as the agent of this service is appropriate. Modeling the **TextEditor** in this fashion has many advantages over extending **Emailer** to "write text messages" itself:

- The **TextEditor** is the central port of call for all things text editing (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 (the logic in **Emailer** is thus streamlined and focused on its purpose)
- 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) without much difficulty

Similarly, "checking spelling" may be performed by a **SpellChecker** component. 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 **SpellChecker**. **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 of composition is important because it highlights the relationship between a component and the services it uses in a formal sense: A component depends on its services to perform a function.

In our example, an **Emailer** depends on a **SpellChecker**, a **TextEditor**, and an **AddressBook**. This service relationship is called a *dependency*. More specifically, the service component is a dependency of the client component.

Composition also applies transitively; so service components are themselves composed of dependencies, who have their own dependencies and so on ad infinitum (and sometimes ad nauseum). In our case, a **SpellChecker** may depend on a **Parser** to recognize string tokens and a **Dictionary** of valid words to match these tokens against. The **Parser** may itself depend on a **CharacterReader**, the **Dictionary** on a **Database** and so on.

This idea of a component that depends on other components transitively and the assembly of all these dependencies into a single unit (the whole "system of pipes") is known as a graph of dependencies or simply: an *object graph*. This object graph, though composed of many dependencies, is functionally a single unit exposing the functionality of its *topmost* object

Pause for a second. We've used a lot of terms such as service, dependency, client, and so forth; before proceeding let's define them a bit more formally.

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

Fairly vague definitions! And, as you are probably thinking, can be applied rather generously--to almost any scenario. However, the important terms that you should have noticed are "well-defined" and "well-understood". They say a lot more than simply describing services and clients as endpoints of an activity; rather, they imply a clear *contract* between service and client in the role of performing a specific function; one that is formally understood by both entities.

In the particular world of objects and methods, these abstract concepts take on a more concrete form:

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

Right. Now we have broadened the idea of a contractual relationship to a more compositional one. Not only can I describe a system of objects as discrete services and their clients, but I also begin to see that a client cannot function without its services. In other words, a client (dependent) is composed of its services (dependencies)¹.

NOTE

Thus, an object graph is the contractual assembly of a single component and its various dependency components (including *their* dependencies) toward a specific purpose.

¹ For the rest of this book, I will use the words *dependent* and *dependency* as interchangeable with *client* and *service*, respectively.

Dependency Injection as a subject is primarily concerned with reliably and efficiently building such object graphs; and the strategies, patterns and best practices therein. It behooves us then, to explore ways in which object graphs can be constructed to form coherent, single units of functionality; ostensibly to create objects that are complete object graphs.

1.2 Pre-DI Solutions

In this vein, let us describe such a class of object and look at specific ways to build it. First, our class (the trivial example of an Apple and its Core) in figure 1.2.

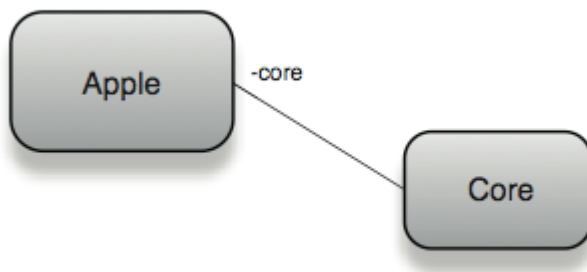


Figure 1.2 Object **Apple** is composed of another object, **Core**

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

```
public class Apple {
    private Core core;
    public Apple() {
        this.core = new Core();
    }
}
```

Then constructing a complete **Apple** (one with a **Core**) is as simple as constructing an **Apple** itself:

```
Apple a = new Apple();
```

Although fairly naive, this sort of construction is sufficient for the purpose and does illustrate a common and widely-used method for constructing object graphs. It is also quite common in older applications and is often taught in introductory language courses. Of course, this method has many drawbacks and does not scale well when dealing with very large object graphs.

Let us examine the most serious drawback—the one that prevents us from creating objects of the same class with different behaviors. To get a better idea of what I mean, let’s resurrect the *Emailer*. 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

How would we create an Emailer with a French SpellChecker? We can't! This class not only encapsulates its dependencies, it also encapsulates the *creation* of its dependencies.

What we need is a better solution to this problem. Which leads to the next topic: *construction by-hand* (sometimes, “manual” Dependency Injection); where instead of a dependent creating its own dependencies, it has them set 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 and many architectural styles to boot. In the earlier section, we used the object's constructor to create its dependencies. With a slight modification, we can still keep the structure of the object graph, but offload the burden of creating dependencies to an external actor (see figure 1.4). 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 **Emailers** with various behaviors quite easily. Here is one for French spelling:

```

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

```

And even one for Japanese spellers,

```
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 these services that suit your needs (French, Japanese, and so on), as shown in figure 1.5.

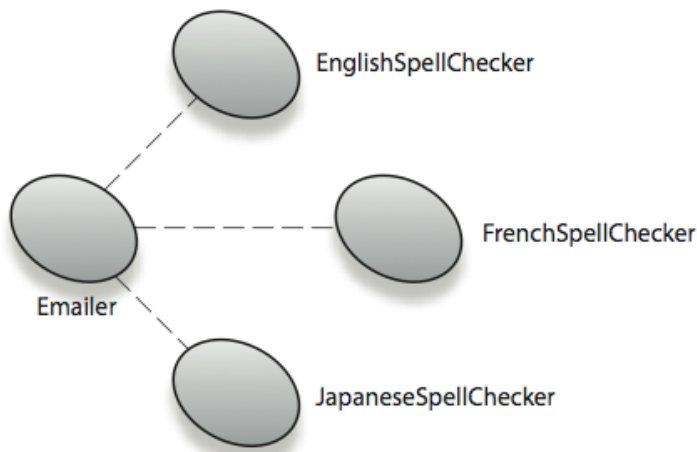


Figure 1.5 Emailer can now check spelling in a variety of languages

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 simply accepts a value and sets it as the dependency). You can also pass in the dependency via a constructor as in this 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 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 previous example if you forgot to call the setter). This concept is very important in object oriented programming, and is referred to as contract-safety. We will study this concept in much greater detail in the coming chapters.

The idea of “connecting the pipes” together, or placing a reference to a dependency into a client object is central to constructing object graphs. It is sometimes also referred to as “injecting” objects into one another, or “wiring” the objects together. To avoid confusion, I will use the term wiring when we talk about connecting a dependency to its client.

There are many problems with Construction by-hand, the most obvious one being the burden of knowing how to construct object graphs being placed on the end-user of the service. If I use the same object graph 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.

As you can see, this quickly becomes nightmarish and unworkable for any decent sized code base. 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. There are also potent arguments against Construction by-hand that we will encounter in other forms (such as the problem of tight-coupling) 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 Factory² pattern). The idea behind the factory pattern is to offload the burden of creating dependencies to a third party called a factory (shown in figure 1.6). Just as an automotive factory creates and assembles cars, so too does an abstract factory create and assemble object graphs.

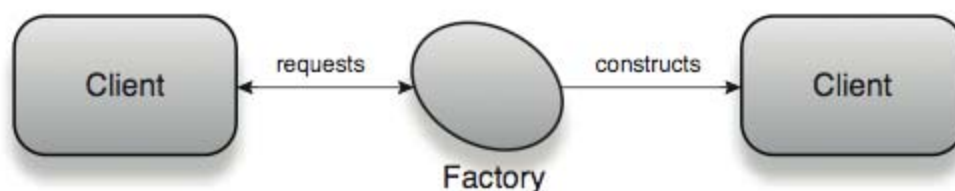


Figure 1.6 A client requests its dependencies from a factory

Let's apply the factory pattern to our **Emailer**. The **Emailer**'s code remains the same as we ended up with in the last section as in listing 1.2.

Listing 1.2 An email service (in Java) whose spell checker is set externally

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

However, instead of constructing the object graph in our client code we do it inside a factory as per listing 1.3.

Listing 1.3 A "French" email service factory (in Java)

```
public class EmailerFactory {
```

² Eric Gamma et al. GoF

```

    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; one with a French spelling checker. Any code that uses French email services is now fairly straightforward (see listing 1.4, below)

Listing 1.4 Code that obtains a French email service from a factory (in Java)

```

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

```

The most important thing to notice here is that the 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), I have separated the code using the **Emailer** from the code that creates and configures the **Emailer**.

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 (in Ruby) that constructs an email service with a Japanese spell checker, text editor

```

class EmailerFactory
  def japaneseEmailer
    service = Emailer.new
    service.spellChecker = JapaneseSpellChecker.new
    service.addressBook = EmailBook.new
    service.textEditor = SimpleJapaneseTextEditor.new
  end
end

```

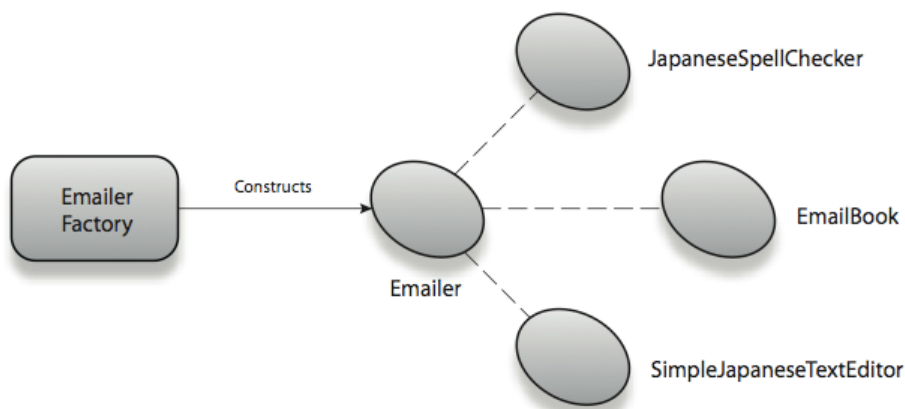


Figure 1.7 EmailerFactory constructs and assembles a Japanese emailer with various services

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

```

service = EmailerFactory.new.japaneseEmailer

```

The beauty of this approach is that client code only needs to know which factory to use to obtain a dependency.

While the factory pattern solves many of the problems raised by the Construction by-hand method (and solves them quite nicely), it still leaves us with some significant hurdles to overcome. The fact that a factory must accompany every service (or object graph) should be enough to give you pause. Not only this, a factory must accompany every variation of every service. Apart from being a sizable amount of distracting clutter it adds a large amount of peripheral code to be tested and maintained; and must be changed accordingly should the structure of your object graphs change. Take a look at a concrete example in listing 1.6 and you will see what I mean.

Listing 1.6 Factories (in Ruby) that create either French or Japanese email services

```
class EmailerFactory
  def japaneseEmailer
    service = Emailer.new
    service.spellChecker = JapaneseSpellChecker.new      |#A
    service.addressBook = EmailBook.new
    service.textEditor = SimpleJapaneseTextEditor.new    |#A
  end
  def frenchEmailer
    service = Emailer.new
    service.spellChecker = FrenchSpellChecker.new        |#B
    service.addressBook = EmailBook.new
    service.textEditor = SimpleFrenchTextEditor.new      |#B
  end
end
(Annotation) <#A Specific dependencies of a Japanese email service>
(Annotation) <#B Specific dependencies of a French email service >
```

Now, if I wanted an English version of the **Emailer** I would have to add yet another method to the factory. You may be tolerant of this at face value: different factories for different purposes. However, consider what happens when I want to make a basic change in the **Emailer**: replace the **EmailBook** component with one that can store both phone numbers and email addresses (call it a **PhoneAndEmailBook**). I am forced to make the following changes, as seen in listing 1.7.

Listing 1.7 The relevant changes to a factory to add phone & email addressing functionality

```
class EmailerFactory
  def japaneseEmailer ...
    emailer.addressBook = PhoneAndEmailBook.new
    ...
  def frenchEmailer    ...
    emailer.addressBook = PhoneAndEmailBook.new
    ...
  def englishEmailer   ...
    emailer.addressBook = PhoneAndEmailBook.new
    ...
end
```

All three of the changes are identical! If this doesn't bother you, it should. It is clearly not a desirable scenario. You can somewhat mitigate the problem by creating another factory that builds a half-constructed **Emailer** with only the language-specific components (Japanese or French spell checkers) remaining to be set by the original factories but this is not exactly an ideal remedy; and it is especially troublesome if you want to wire the dependencies via a constructor. It could also mean a lot of refactoring if you have got repeating code (as we did in the previous example).

Furthermore, any client code is at the mercy of available factories: you must either create new factories for each additional object graph variation. Additionally, any client of a factory must also know which factory to use. This is a derivative of the early problem we fought of a user having to know the *how* part of object graph construction, though certainly not as foul. The idea of a factory that uses another factory to obtain parts of its dependency graph is also less than appealing. All this spells reams of additional code to write, test and maintain.

In the case of **Mailer**, following the idea to its inevitable extreme yields:

```
Mailer service = new MailerFactoryFactory()           #1
                  .NewAdvancedMailerFactory()         #2
                  .NewJapaneseMailerWithPhoneAndEmail(); #3
(Annotation) <#1 "General" factory produces simple or advanced factories>
(Annotation) <#2 Factory for advanced Mailers>
(Annotation) <#3 Factory producing specific Mailer configurations>
```

It is not unusual to come across code like this. I have seen it even in some very carefully engineered projects. When dealing with objects graphs that are constructed from disparate libraries (or packages) the situation becomes even more complex. You often end up with an explosion of factory methods for every purpose. Clearly, a more robust solution is warranted.

While the factory technique has been very successful over the years (many core libraries such as the Java platform SDK still make heavy use of this pattern), its drawbacks are too many in number to dismiss as we scale to larger and more complex and frequently changing code bases. 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 on whose shoulders we place the burden of obtaining a fully constructed service (object graph) for our use. More specifically, a service locator is a kind of factory that has a specialized purpose, namely obtaining a service. Thus, service locators can be used on top of various other means, such as regular factories, or construction by-hand to obtain fully constructed services.

In the typical case, service locators are used to locate services published by external sources such as a web-service that transforms images between formats or an Enterprise JavaBean³ that places purchase orders for books.

These external sources may or may not reside in the same application, machine or local area network. Put another way, they are external in concept rather than physical location. They may be as simple as a library that obtains instances of a service that it ships with (and not at all “external” in any physical capacity). On the other hand they may be as complex as the interface to a Mars Rover several millions of miles away and directed via radio signals using NASA’s communications protocols. However, the basic pattern of obtaining a service via a locator is the same in all cases. Take a look at figure 1.8 for a visual idea.

³ Very simply, an Enterprise JavaBean (or EJB) is a service built according to a particular standard; namely the EJB standard. You can learn more about EJBs at <http://java.sun.com/products/ejb>

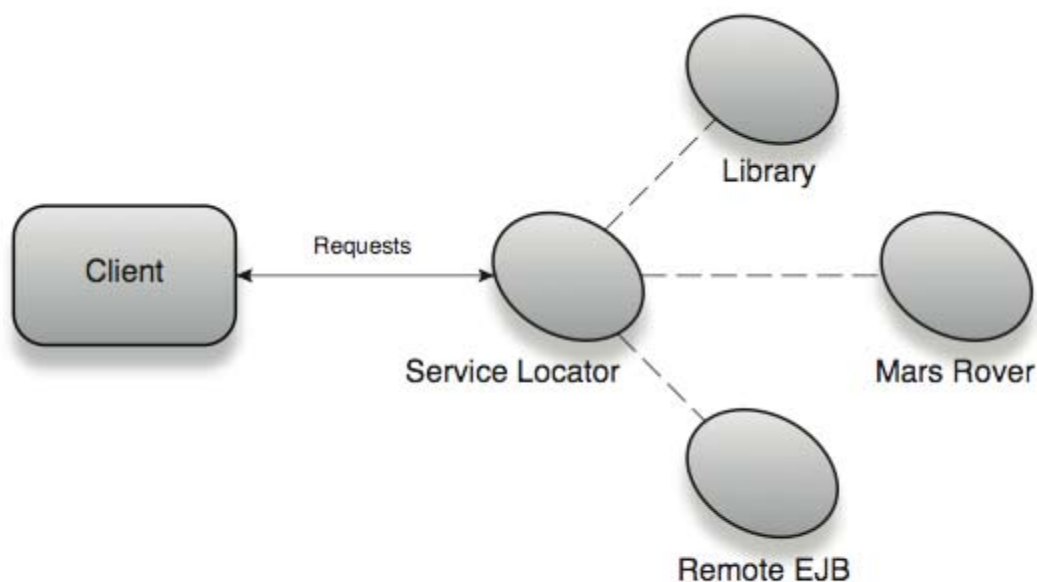


Figure 1.8 Service Locators find or construct specific services requested by clients

Let's look at what a service locator does:

```
emailer = ServiceLocator.new.get("Emailer")
```

Notice that we pass the service locator a key, in this case a string with the word "Emailer". This tells our locator that we want a particular kind of service, namely an email service. This is significantly different from a factory that produced only one kind of service. Looked at another way, a service locator is a factory that can produce (or "locate") *any* kind of service.

Instantly we have reduced a client's dependence on a potential horde of factories. While you may still end up coding these factories underneath the service locator, it is important to recognize that clients do not have to know about them directly anymore. They simply request a service by its key. More importantly, if you choose to use some simpler method than a slew of factories beneath a service locator, this is possible without impacting client code.

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:

```
emailer = ServiceLocator.new.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.

Service locators mitigate almost all problems except one: clients of services must still go out and explicitly ask for each of their dependencies. They must also know the names of those dependencies in the service locator's language (namely, the key associated with a service). With Dependency Injection, we will remove even this need. 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 Believe the hype: Dependency Injection

When looking at the various pre-DI solutions to constructing object graphs what should have struck you is the criticism we applied to each one: particularly the principle of separating the need to obtain a dependency from the dependent object. This principle is at the heart of Dependency Injection and forms much of its foundation and philosophy. Not only does it release client code from having to know how to construct and find its dependencies, but it also helps make code concise and easy to maintain. Think back to our example of the French and English email services; with pre-DI methods, a client of a French or English email service would have to know which one it wanted and explicitly construct or request it via a factory.

With Dependency Injection clients need know nothing about French or English **Emailers**, let alone French or English **SpellCheckers** and **TextEditors** (or their factories), in order to use them. This idea of "not knowing" is central to Dependency Injection. "Not knowing", or more accurately: not *asking* for dependencies, and instead having them provided to you is known as 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 contact and arrange auditions for actors when gigs become available, so too do Dependency Injection frameworks use this principle to provide objects with components they depend on.

This phenomenon of providing an object with its dependencies is very similar to what we saw in construction by-hand (which is why it is sometimes referred to as manual Dependency Injection). However, with one key 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.

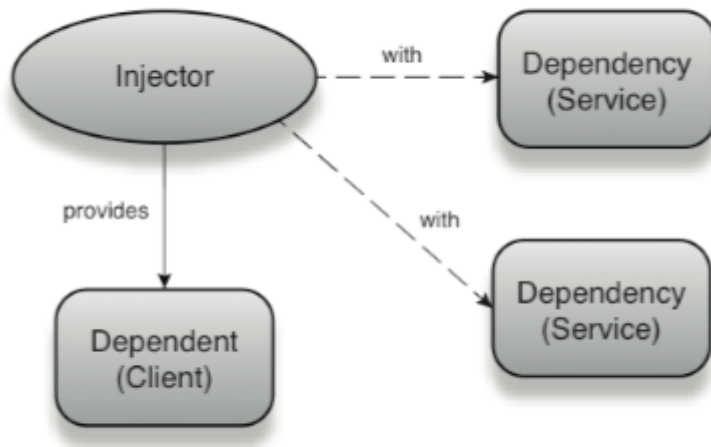


Figure 1.9 The Injector provides an object with its dependencies

Thus, control over the construction, wiring and assembly of services is given up to such a framework and no longer resides with the services themselves (or indeed, their clients). This reversal of responsibilities is often referred to as Inversion of Control and is roughly interchangeable with the idea behind the Hollywood Principle.

NOTE

Dependency Injection frameworks are often 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#). More on this dual terminology follows later in the chapter.

Listing 1.8 shows the Hollywood Principle in action.

Listing 1.8 A Heart and a user (client) of hearts: Man (in Java)

```

public class Heart {
    private int beats;
    public void beat() {
        beats++;
    }
}

public class Man {
    private Heart heart;
    public Man(Heart heart) {
        this.heart = heart;
    }
    public void live() {
        for (int i = 0; i < 1000; i++)
            heart.beat();
    }
}
(Annotation) <#1 Man is a client of his heart>
(Annotation) <#2 Beats a thousand times; a short-lived man!>
  
```

In this example, our dependent is **Man** ([#1]) and the service he uses ([#2]) is **Heart**. Notice that neither class is aware of how to construct or wire its graph; this is a significant change from the three other ways we've seen so far (construction by-hand, factories and service location) where clients were forced either to build or explicitly ask for a service. It should also highlight how your code can be made lean and concise--the primary virtue of the Hollywood Principle.

Note that in order to live, **Man** does not need to expose anything about his organs or how they work. Put another way, **Man** encapsulates **Heart** and the action of living is performed transparently; along with the action of **Heart** beating. Thus, any code that calls on **Man** to live need know nothing about hearts or their methods as corollary actions. That task is now performed by a dependency injector (see figure 1.10).

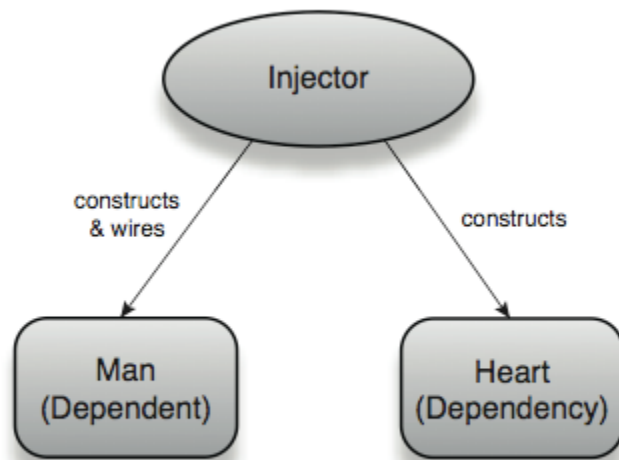


Figure 1.10 The Injector constructs and wires **Man** with his dependency (**Heart**)

Applying the same principle to our wonted email service example, it is easy to see how client code becomes simple and concise (listing 1.9 and figure 1.11).



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

Listing 1.9 Client code (in Java) that uses an email service provided by dependency injection

```

public class EmailSendingClient {
    private Emailer emailer;
    public EmailSendingClient(Emailer emailer) {
        this.emailer = emailer;
    }
}
#1
  
```

```

    public void sendEmail() {
        emailer.send("Josh", "Hi Josh!");
    }
}
(Annotation) <#1 Wired via the class's constructor>
(Annotation) <#2 Send Josh a message saying "Hi Josh!">

```

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

Elementary?

In the original four novels and fifty-six short stories by Arthur Conan Doyle, Sherlock Holmes never once says "Elementary, my dear Watson!"

Yes, I checked.

Though he does say "It is elementary" and "my dear Watson" separately quite often.

Of course, we have not yet shown how the wiring is actually done in a specific case; as this differs vastly from Injector to Injector, but what we've seen is still very instructive as it highlights the removal of infrastructure code (code meant for wiring and construction) from application code (the core purpose of a service). The next couple of sections explore this idea and its history in greater detail, 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 title suggests. You will come across the term *Inversion of Control* (or *IoC*) quite often, both in the context of Dependency Injection and outside it. In the previous section, we saw that IoC refers to the idea of a third-party responsibility over behavior; and that this is roughly the same idea that the Hollywood Principle purports. However, the phrase Inversion of Control is itself rather vague and doesn't quite evoke the lean and mean semantic of a dependent *provided* with its dependencies.

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

- a component inside a Java EE application server,
- a component 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 insist that all of these cases are consistent with its definition and that the Hollywood Principle (or more generally Dependency Injection itself) is simply one instance of IoC. While this may be true in a philosophical sense, I find that it adds little value to the discussion; and more importantly, it confuses the already rather murky dictionary of terms around Dependency Injection.

In colloquial use however, Dependency Injectors are frequently referred to as IoC Containers. Many Dependency Injection frameworks even style *themselves* as IoC containers adding mud to an already muddy

nomenclature. These two uses of "IoC" are somewhat paradoxical; one being rather vague and in the abstract meaning any kind of reversed responsibility, the other being a specific synonym to Dependency Injection.

For this reason I strongly discourage the use of the term "IoC" when referring to Dependency Injection or "IoC Container" when referring to a Dependency Injector. In fact I strongly discourage use of the term, period; as stated before, it provides scant additional value in any discussion.

Therefore, for the rest of this book we will abandon the term IoC and its evil cousin IoC Container in favor of the following, more precise terms:

- **Hollywood Principle** - the idea that responsibility for constructing and providing a dependent with its dependencies rests with a *third party*
- **Dependency Injector** - the aforementioned third party; a framework or library that embodies (or enacts) the Hollywood Principle
- **Dependency Injection** - the range of concerns behind designing applications built on the Hollywood Principle

Right, now that you are armed with the clarity of this knowledge, it is worth learning a bit of history. As near as I can determine, it wasn't until Martin Fowler wrote his seminal article, "Inversion of Control Containers and the Dependency Injection Pattern"⁴ that the term *Dependency Injection* came into popular use. Ostensibly, Martin coined the term.

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 Dependency Injection became more streamlined and tolerant to rapid structural change and refactoring.

Dependency Injection in Egypt

After Octavian Caesar defeats Marc Antony's armies in Egypt and lays siege to Alexandria, he sends an emissary (Proculeius) to Queen Cleopatra:

Proculeius:

Be of good cheer;
You're fall'n into a princely hand, fear nothing:
Make your full reference freely to my lord,
Who is so full of grace, that it flows over
On all that need: let me report to him
Your sweet **dependency**; and you shall find
A conqueror that will pray in aid for kindness,
Where he for grace is kneel'd to.
-- Act V, Scene II.

Whereupon she accepts his generous offer of patronage but famously takes her own life instead. This was the earliest written use of the word **dependency** that I could find. It is from Shakespeare's *Antony and Cleopatra*.

⁴ The article has been updated many times over the years. You can read the latest version at <http://martinfowler.com/articles/injection.html>

The modes of configuring an application and its Dependency Injector also evolved from verbose sets of contracts and xml files to more concise forms, using new language constructs such as *Annotations* (or *Attributes* in C#) and *Generics*. Dependency Injectors also took better advantage of class manipulation tools such as *reflection* and *instrumentation*. And began to exploit design patterns such as *Decorators*⁵, *Builders*⁶ 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.

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 functionality that interface with one another. Units are simple to test and can easily be swapped with mock or dummy counterparts as needed. DI allows objects to evolve independently, yet still be assembled into coherent whole applications with a minimum of effort. As easy as that sounds! 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 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. We'll 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 the problem of object graph construction, several possible solutions and hinted at a wonderful 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 next chapter, the nuts and bolts of Dependency Injection, come with me on 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 wave-of-the-hand and a passing 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 the following languages: Java, C#, C++, Python and Ruby. They will also be grouped accordingly. The majority of them are open-source and freely available with strong user communities and support groups. Not all of them are purely DI-focused, and almost none support the full gamut of Dependency Injection concepts and functionality described in this book. Nevertheless each is worth a look for different, unique takes on DI.

⁵ The Decorator Pattern. Design Patterns: Erich Gamma et al.

⁶ The Builder Pattern. Design Patterns: Erich Gamma et al.

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.

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. Its 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 the Avalon framework.

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 (author of *Expert One-on-one J2EE Design And Development*) and arose both from his book and from an enterprise project he was working on. It was established as an open-source project in 2003 and grew rapidly in scope and adoption from there. 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 wiring 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. Spring 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 in a very careful and philosophical manner and many of the discussions found on its website and documentation are of a theoretical nature. It was built as a lightweight "engine" for wiring components together. In this sense it is extremely well-suited to extensions and customization. This makes PicoContainer a powerful Dependency Injector to use under-the-covers. PicoContainer supports both setter and constructor wiring, but clearly demonstrates a bias toward the latter (as will I through this book). 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.

Many of the important theoretical concepts behind DI were evolved by PicoContainer and its authors.

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 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 (called *Vista*, funnily enough) 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 wiring, 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's plugin architecture. It supports both setter and constructor wiring 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 C# and .NET

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#.

StructureMap

StructureMap is one of the earliest Dependency Injection libraries for the .NET platform. It has been in use since 2004 (shortly after Spring's emergence) and is a fairly mature library. Newer versions of StructureMap offer flexible configuration and support stronger type and contract safety with the use of language features such as *Generics*. Like most modern DI libraries, StructureMap supports both setter and constructor wiring but also offers testing and diagnostic features such as logging, tracing and wiring of *mock objects*.

ObjectBuilder

ObjectBuilder is a Dependency Injection library whose stated purpose is more of a tool for building Dependency Injectors than being a Dependency Injector itself. ObjectBuilder was created by the *Microsoft patterns & practices* team to service a couple of very different DI use cases. While ObjectBuilder has been around for a while and is reasonably mature, it does not offer quite the range of features that StructureMap does.

Castle MicroKernel

The *Castle Project* is a large framework intended for building Web applications. It has several components to support and abstract common web development use cases, including a persistence module (that integrates

NHibernate), an MVC framework named *MonoRail* and a Dependency Injector named *MicroKernel*. *MicroKernel*, like *PicoContainer* is a lightweight DI library focused on extensibility and is intended for use under-the-covers. An abstraction for configuration and common-use cases is provided in the form of *Windsor*, which builds on *MicroKernel* (in the same way as *NanoContainer* builds on *PicoContainer*). Using *MicroKernel* or *Windsor* does not lock you into the other components of the Castle Project, though you may find them useful in conjunction.

PicoContainer.NET & Spring Framework.NET

The success of Spring and PicoContainer in Java have prompted .NET analogs. While in the case of Spring, this also includes a vast array of ancillary support and integration modules, PicoContainer.NET is a faithful companion to the original lightweight Java version. Due to both their origins in Java, they continue to evolve first there, with features and improvements flowing in later to their .NET counterparts.

1.4.3 DI in other Languages & Libraries

There are several other DI libraries that I have not mentioned above. One 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. Seam was created by Gavin King⁷ as part of the JBoss family of products. Seam's rise has inspired the creation of a new standard named *Web Beans* for the Java EE platform.

Similarly, there are many libraries that serve as an analog for a more successful counterpart in a different platform. We have already seen Spring's and PicoContainer's .NET analogs. *Ninject* is another such analog to Google Guice for the .NET platform. It offers many of the same kinds of features and simplicity of use, though it is newer and less mature than Guice.

Some platforms (or languages) make it harder to design Dependency Injectors because they lack advanced features such as *garbage collection* and *reflection*. C++ is a prime example of such a language. However, it is still possible to write powerful Dependency Injectors in these languages with other methods that substitute for such tools. Some C++ libraries use *precompilation* and automatic *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

⁷ Gavin King is the creator of Hibernate, the vastly popular Object-Relational Mapping and Persistence framework. Seam is Gavin's second major open-source project.

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, 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 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. However, underneath the locator, one *may* still have to code and register all of those factories. Finally, it requires that a client explicitly request each of its dependencies by key.

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!