Extracted from:

The Cucumber for Java Book

Behaviour-Driven Development
for Testers and Developers

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The Cucumber For Java Book

Behaviour-Driven Development for Testers and Developers

Seb Rose, Matt Wynne, and Aslak Hellesøy

edited by Jacquelyn Carter

The Cucumber for Java Book

Behaviour-Driven Development
for Testers and Developers

Seb Rose
Matt Wynne
Aslak Hellesøy

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

Pragmatic Bookshelf

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-941222-29-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2015

CHAPTER 11

Simplifying Design with Dependency Injection

In *Dependency Injection*, on page ?, we saw how Cucumber uses dependency injection (DI) to share an instance of KnowsTheDomain between our step definitions, but we really only scratched the surface. Now it's time to dig a little bit deeper.

In this chapter we'll discuss how DI can help improve the design of your test code and the various DI containers that are integrated with Cucumber. Then we'll dive in and refactor our ATM example to use DI more effectively, showing you how to do it with four of the popular DI containers.

DI and Cucumber

You don't *need* to use a DI container when you use Cucumber. When you use Cucumber without one of the DI integrations, it manages the creation of all your hooks and step definitions itself. Cucumber creates fresh instances of each step definition or hook class for each scenario. Of course that means that these classes need to have a default constructor; otherwise Cucumber won't know how to create them. This makes it hard to share state safely between several step definition classes.

DI can make some of your everyday work less tedious and error prone. As soon as you add one of the DI integrations, the DI container takes over the responsibility for creating the hooks and step definitions. All the DI containers can also *inject* objects into our hooks and step definitions when it creates them. Even better, the DI container will create instances of any object that needs to be created so you can easily build networks of dependent objects, leaving the hard work of wiring them all together to the DI container.

The two common ways that DI containers inject objects are *constructor injection* and *field injection*. In the following sections, we'll mainly be using constructor injection, but we'll also show you field injection in action.

Letting DI Manage Shared State

A DI container is just a tool that creates and manages instances of some classes for us. If you look back at the code we wrote to share a single instance of `KnowsTheDomain` among all our step definition classes, you'll see that we never create an instance of `KnowsTheDomain` using `new`. That's because our DI container, PicoContainer, has been doing it for us. What's more, PicoContainer created a new instance of `KnowsTheDomain` for each scenario and injected that instance into every step definition class that needed it. This made it easy for us to create a focused step definition class for each domain entity in our application, relying on PicoContainer to share state between them.

If we had done this without DI, it would have meant much more work for us. We could have shared state by creating a `static` instance of `KnowsTheDomain`, but that instance would then be shared by all our scenarios. Since we want each scenario to have its own fresh copy of `KnowsTheDomain`, we would have to add a `@Before` hook to reset the shared instance. But we don't need to do any of this, because a DI container will do it for us.

Cucumber's use of DI makes our lives much simpler by taking care of the creation of our hook and step definition classes, as well as all the shared state that they depend on. For each step definition that needs access to a scenario's shared state, we define a constructor that takes the shared class as a parameter. If a scenario needs access to instances of several different classes, we simply provide a constructor that has a parameter for each of them:

```
public SomeStepDefinitionOrHooks(Foo sharedFoo, Bar sharedBar) {
  // Store sharedFoo and sharedBar for later use
}
```

You'll want to use DI in most of your Cucumber projects, because it makes sharing state between step definition classes so much simpler. Cucumber has integrations with several DI containers, which we'll take a brief look at next.

DI Container Integrations

Cucumber ships with integrations to several of the more popular DI containers (as well as some unfamiliar to most people), shown in the following list. The code you write will look slightly different depending on which DI container you choose.

- cucumber-picocontainer—PicoContainer:[1] A lightweight DI container from Aslak Hellesøy, Paul Hammant, and Jon Tirsen
- cucumber-guice—Guice:[2] A lightweight DI container from Google
- cucumber-spring—Spring:[3] A popular framework that includes DI and much more
- cucumber-weld—CDI/Weld:[4] The reference implementation of the CDI (Context and Dependency Injection Framework for the Java EE platform)
- cucumber-openejb—OpenEJB:[5] A stand-alone EJB server from Apache, including a CDI implementation

You choose which framework to use by including the relevant Cucumber JAR in your classpath, but *only one* Cucumber DI JAR should ever be on the classpath. As soon as you put one of these JARs on your classpath, Cucumber will delegate the creation and management of your hooks and step definitions to the DI container you chose. The Cucumber JARs contain only the code to integrate the DI container—you'll also need to add a dependency on the DI container itself.

Which DI Container Should I Choose?

The various DI containers provide almost exactly the same functionality. Each needs slightly different configuration, but the choice mostly depends on what DI container you're already using in your application. If your application uses Spring, then choose cucumber-spring. If your application uses Guice, then choose cucumber-guice.

If your app isn't using DI at all, PicoContainer is a great choice because it's so simple to use.

Let's to go back to our ATM example and see how DI can improve the structure of our test code. You'll be surprised what a big difference it can make.

Improving Our Design Using DI

As we've seen, DI can make our lives simpler by doing some of our work for us. At the moment our application is using DI to share some state, but code in KnowsTheDomain is managing the creation of quite a few domain entities. As

1. http://picocontainer.codehaus.org
2. https://github.com/google/guice
3. http://projects.spring.io/spring-framework/
4. http://weld.cdi-spec.org
5. http://openejb.apache.org/index.html

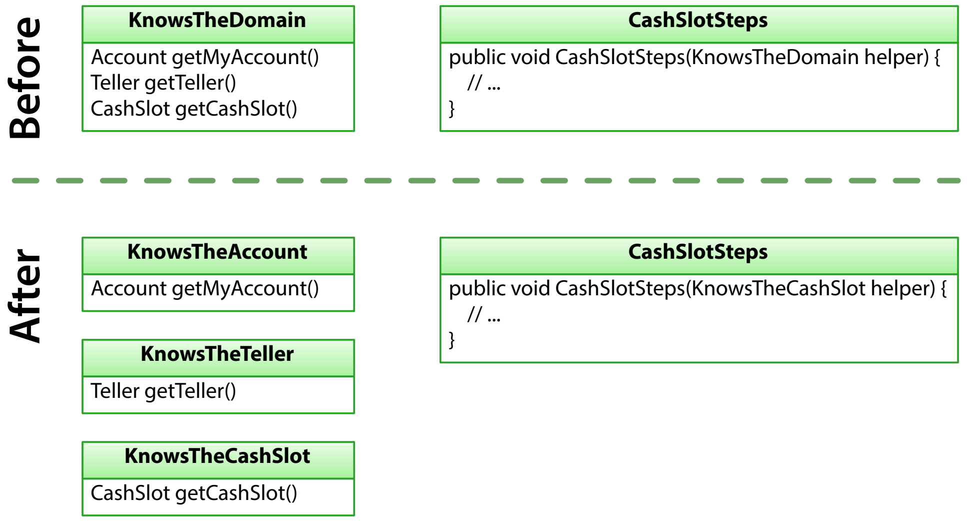
our application grows, we're likely to find more domain entities that need to be shared between our step definitions. The temptation would be to put all our shared domain entities into KnowsTheDomain, but this would soon grow huge, exhibiting the Monster Object antipattern.[6]

To keep our step definitions maintainable, it's a good idea to create a step definition class for each domain entity. It's clear that the Cash Slot step definition is going to need to interact with the Cash Slot domain entity, but will it ever need to know about a Customer entity? Probably not, so why would we pass it a helper that has access to the Customer?

Starting with the code from the end of Chapter 10, *Databases*, on page ?, we're going to refactor our ATM example to use DI more effectively. We'll take small steps,[7] looking at the result of each refactoring to see if there is another refactoring that could improve the design further. At the end you'll see a much cleaner application with fewer classes and a clearer architecture.

Decomposing KnowsTheDomain

We'll start by splitting KnowsTheDomain into several, smaller, cohesive helper classes—one for each domain entity—as as shown in the figure. Then we can make sure that a step definition only has access to entities that it *should* need to interact with by passing them in at construction time.



6. http://lostechies.com/chrismissal/2009/05/28/anti-patterns-and-worst-practices-monster-objects/
7. http://c2.com/cgi/wiki?RefactoringInVerySmallSteps

We'll move the functionality that's specific to each domain entity into a separate helper class. Take a look at KnowsTheCashSlot, for example:

dependency_injection/pico/02/src/test/java/support/KnowsTheCashSlot.java

```java
package support;
import nicebank.CashSlot;

public class KnowsTheCashSlot {
    private CashSlot cashSlot;

    public CashSlot getCashSlot() {
        if (cashSlot == null){
            cashSlot = new CashSlot();
        }
        return cashSlot;
    }
}
```

Similarly, we can create the KnowsTheTeller and KnowsTheAccount helper classes.

Next we have to change references to KnowsTheDomain, such as in CashSlotSteps:

dependency_injection/pico/02/src/test/java/nicebank/CashSlotSteps.java

```java
import support.KnowsTheCashSlot;

public class CashSlotSteps {
  KnowsTheCashSlot cashSlotHelper;

  public CashSlotSteps(KnowsTheCashSlot cashSlotHelper) {
      this.cashSlotHelper = cashSlotHelper;
  }
}
```

Not all changes are quite so simple. Our TellerSteps, for example, interacts with KnowsTheAccount as well as KnowsTheTeller, so we have to pass both in to the constructor. The DI framework can handle multiple parameters, and calls the constructor correctly without any extra work on our part:

dependency_injection/pico/02/src/test/java/nicebank/TellerSteps.java

```java
public TellerSteps(KnowsTheTeller tellerHelper, KnowsTheAccount accountHelper) {
    this.tellerHelper = tellerHelper;
    this.accountHelper = accountHelper;
}
```

Once we finish moving functionality into the new helper classes, we notice that there's still some code in KnowsTheDomain that's concerned with creating a shared EventFiringWebDriver. Since this is specific to the technology driving the user interface, it's not logically a concern of any of the KnowsTheXxx classes, so we need to decide where to put it, which we'll do next.

Extracting a Web Driver

Both AtmUserInterface and WebDriverHooks are dependent on a shared EventFiringWebDriver. This used to be managed by KnowsTheDomain but really has nothing to do with the domain. Instead, let's extract a new MyWebDriver class:

dependency_injection/pico/02/src/test/java/support/MyWebDriver.java
```java
package support;

import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.events.EventFiringWebDriver;

public class MyWebDriver extends EventFiringWebDriver{
    public MyWebDriver() {
        super(new FirefoxDriver());
    }
}
```

There's no code left in KnowsTheDomain, so we can delete it. However, we still have to inject the shared MyWebDriver instance into AtmUserInterface and WebDriverHooks by changing their constructors:

dependency_injection/pico/02/src/test/java/support/AtmUserInterface.java
```java
private final EventFiringWebDriver webDriver;

public AtmUserInterface(MyWebDriver webDriver) {
    this.webDriver = webDriver;
}
```

(The WebDriverHooks constructor looks the same.)

Now we run mvn clean test to make sure we haven't introduced any defects.

We've improved our design by decoupling the UI from the domain entities. Our DI container now has the responsibility of managing a shared instance of MyWebDriver and injecting it into any constructor that needs it. In the next step, we'll see that most of the helper classes are doing nothing (or very little) that our DI container can't do for us, so we'll refactor them away entirely.

Replacing the Helper Classes

At this point, we've created smaller, more cohesive helper classes, so now let's take a good look at each of them to see if we're happy with the structure of our test code. Now that we have smaller helper classes it's easier to see exactly what each one does and decide, armed with our deeper understanding of DI, if we could do it better. And if we think that we could improve things further, we still have the safety net of a passing scenario to ensure that we don't break anything while we continue the refactoring.

KnowsTheCashSlot

Taking a look at the class KnowsTheCashSlot, we can see that all it's doing is managing the creation of the domain entity CashSlot. The reason we're using DI in the first place is to manage the creation of shared objects, so it seems strange that we've ended up with our own class that does just that! What has happened is that by refactoring KnowsTheDomain, and with our new knowledge of how DI works, we can now see that we don't need KnowsTheCashSlot at all. So, let's simplify our codebase by deleting KnowsTheCashSlot and injecting the CashSlot directly into our step definition:

dependency_injection/pico/03/src/test/java/nicebank/CashSlotSteps.java
```java
CashSlot cashSlot;

public CashSlotSteps(CashSlot cashSlot) {
    this.cashSlot = cashSlot;
}
```

We also need to inject CashSlot into our ServerHooks:

dependency_injection/pico/03/src/test/java/hooks/ServerHooks.java
```java
private KnowsTheAccount accountHelper;
private CashSlot cashSlot;

public ServerHooks(KnowsTheAccount accountHelper, CashSlot cashSlot) {
    this.accountHelper = accountHelper;
    this.cashSlot = cashSlot;
}
```

Run mvn clean test to make sure we're still green!

KnowsTheTeller

KnowsTheTeller also only manages the creation of a Teller. If we make exactly the same changes to delete the class, we'll get a runtime error when trying to create TellerSteps. That's because the DI container doesn't know which implementation of the Teller interface to instantiate. We tell it which concrete class to instantiate by changing the signature of the TellerSteps constructor:

dependency_injection/pico/04/src/test/java/nicebank/TellerSteps.java
```java
KnowsTheAccount accountHelper;
Teller teller;

public TellerSteps(AtmUserInterface teller, KnowsTheAccount accountHelper) {
    this.teller = teller;
    this.accountHelper = accountHelper;
}
```

Run mvn clean test again.

KnowsTheAccount

Removing KnowsTheAccount is more complicated, because (as is now clear) it has several responsibilities. It does the following:

- Opens a database connection (if necessary)
- Deletes any existing accounts
- Creates an account with a specified account number

Let's start by moving the database connection and account deletion into ResetHooks. We need to ensure that this runs before any other hook that needs a database connection, so we specify a low order number:

dependency_injection/pico/05/src/test/java/hooks/ResetHooks.java

```
public class ResetHooks {
    @Before(order = 1)
    public void reset() {
        if (!Base.hasConnection()) {
            Base.open(
                    "com.mysql.jdbc.Driver",
                    "jdbc:mysql://localhost/bank",
                    "teller", "password");
        }
        Account.deleteAll();
        TransactionQueue.clear();
    }
}
```

Now KnowsTheAccount's only responsibility is to store a test account in the database, so the name no longer makes sense. Rename it TestAccount and have it extend Account:

dependency_injection/pico/05/src/test/java/support/TestAccount.java

```
package support;
import nicebank.Account;

public class TestAccount extends Account {
    public TestAccount() {
        super(1234);
        saveIt();
    }
}
```

Now when we inject a TestAccount directly into our steps, PicoContainer will ensure that they all get a reference to the same, newly created Account object. Try it—it still works fine.

We could have done everything in this section without using DI at all, but you can see how things are made so much simpler when the DI container takes care of managing the creation of our shared entities for us. It's a bit like the way garbage collection in the JVM frees us from having to worry too much about creating and deleting object instances.

So far we've been using PicoContainer as our DI container. Next, let's take a look at how we've integrated it with our application.

PicoContainer Is Almost Invisible

PicoContainer is probably the simplest DI container on the JVM, which is why we've been using it. It's certainly the simplest to integrate with Cucumber. Some of the other containers have more options, but PicoContainer is sufficient for most applications. The most likely reason to not use PicoContainer is that your application is already using another DI container.

Our use of PicoContainer is so unobtrusive that you may well have forgotten all about it by now. The only evidence that we're even using it at all are two dependencies in our pom.xml:

dependency_injection/pico/01/pom.xml
```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-picocontainer</artifactId>
    <version>${cucumber.version}</version>
    <scope>test</scope>
</dependency>
```

This dependency puts the cucumber-picocontainer JAR on the classpath, which tells Cucumber to let PicoContainer handle creation of the hooks and step definitions.

dependency_injection/pico/01/pom.xml
```
<dependency>
    <groupId>org.picocontainer</groupId>
    <artifactId>picocontainer</artifactId>
    <version>${picocontainer.version}</version>
    <scope>test</scope>
</dependency>
```

This second dependency puts the *PicoContainer implementation* JAR on the classpath. Without this Cucumber would be unable to find PicoContainer and wouldn't be able to delegate to it.

Apart from adding these dependencies, you can use PicoContainer without adding any code or annotations at all. In the following sections we'll port the example to other DI containers, and we'll see that PicoContainer is the only one that requires so little configuration by us.

Moving to Guice

Guice is a DI container from Google with many of the same features as PicoContainer. In our opinion it's not as easy to use as PicoContainer, but it does have some extra possibilities, as we'll see. Also, since it's part of the popular Google toolset, you may already be familiar with it.

We'll quickly modify our existing ATM solution to run with Guice, pointing out the differences as we go. Further details can be found on the Guice website.[8] Note that as of this writing Guice 4.0 is in beta, but this example uses Guice 3.0 and cucumber-guice-1.2.0.

Switching the DI Container

We'll start with the refactored code that we produced at the end of the previous section. The first thing to do is replace the dependencies on cucumber-picocontainer and PicoContainer in pom.xml:

dependency_injection/guice/01/pom.xml

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-guice</artifactId>
    <version>${cucumber.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.google.inject</groupId>
    <artifactId>guice</artifactId>
    <version>3.0</version>
</dependency>
```

If you run mvn clean test now you'll see a number of errors like this:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running RunCukesTest
Feature: Cash Withdrawal
Failure in before hook:ServerHooks.startServer()
Message: com.google.inject.ConfigurationException: Guice configuration errors:

1) Could not find a suitable constructor in hooks.ServerHooks.
    Classes must have either one (and only one) constructor annotated with
     @Inject or a zero-argument constructor that is not private.
  at hooks.ServerHooks.class(ServerHooks.java:19)
  while locating hooks.ServerHooks
```

8. https://github.com/google/guice

What this is telling us is that Guice is trying to instantiate an instance of one of our glue classes, but it doesn't know which constructor to use (even though they each have only one constructor).

@Inject Annotation

To tell Guice which constructor to use, we add the @Inject annotation to the constructor that we want to inject objects into:

dependency_injection/guice/02/src/test/java/nicebank/CashSlotSteps.java

```
CashSlot cashSlot;
➤ @Inject
public CashSlotSteps(CashSlot cashSlot) {
    this.cashSlot = cashSlot;
}
```

We also have to annotate the constructors in AtmUserInterface, AccountSteps, ServerHooks, TellerSteps, and WebDriverHooks. When we run mvn clean test both of our scenarios fail but with a different error:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running RunCukesTest
Feature: Cash Withdrawal
Listening on http://10.101.1.77:8887/
Failure in after hook:ServerHooks.stopServer()
Message: com.google.inject.ProvisionException: Guice provision errors:

1) Error injecting constructor, org.javalite.activejdbc.DBException:
   com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException:
   Duplicate entry '1234' for key 'number', Query:
   INSERT INTO accounts (number, balance) VALUES (?, ?), params: 1234,0.00
  at support.TestAccount.<init>(TestAccount.java:8)
  while locating support.TestAccount
    for parameter 0 at hooks.ServerHooks.<init>(ServerHooks.java:22)
  while locating hooks.ServerHooks
```

What's going on here? The error being reported is a MySQLIntegrityConstraintViolationException, which is being thrown because we're trying to create more than one account with the same account number. The unique constraint we put on the database won't allow this and that's the cause of the error.

The root cause of this problem is *scope*. We've hit a significant difference between PicoContainer and Guice, which we'll look at next.

@ScenarioScoped Annotation

The Cucumber-Guice documentation says:

> It is not recommended to leave your step definition classes with no scope as it means that Cucumber will instantiate a new instance of the class for each step within a scenario that uses that step definition.[9]

We have two potential scopes to choose from: @ScenarioScoped and @Singleton. In general we will use @ScenarioScoped, because this ensures that state is reset before each scenario starts to run.

Let's annotate our step definitions and hooks to indicate that we want to create them fresh for each scenario. The classes that need to be annotated are AccountSteps, CashSlotSteps, TellerSteps, BackgroundProcessHooks, ResetHooks, ServerHooks, and WebDriverHooks:

dependency_injection/guice/03/src/test/java/nicebank/AccountSteps.java

```
import cucumber.runtime.java.guice.ScenarioScoped;

@ScenarioScoped
public class AccountSteps {
}
```

When we run mvn clean test we get:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running RunCukesTest
Feature: Cash Withdrawal
Listening on http://10.101.1.77:8887/
Server shutdown

Scenario: Successful withdrawal from an account in credit
  Given my account has been credited with 1m<b>$100.00</b>
  com.google.inject.ProvisionException: Guice provision errors:

  1) Error injecting constructor, org.javalite.activejdbc.DBException:
     com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException:
     Duplicate entry '1234' for key 'number', Query:
     INSERT INTO accounts (number, balance) VALUES (?, ?), params: 1234,0.00
    at support.TestAccount.<init>(TestAccount.java:8)
    while locating support.TestAccount
      for parameter 0 at nicebank.AccountSteps.<init>(AccountSteps.java:24)
    at nicebank.AccountSteps.class(AccountSteps.java:24)
    while locating nicebank.AccountSteps
```

9. http://cukes.info/api/cucumber/jvm/javadoc/cucumber/api/guice/package-summary.html

We've fixed the failure in shutdownServer, but we're still trying to create a duplicate account. That's because, although we've annotated the glue code with @ScenarioScoped, Guice is still creating a new instance of every injected class each time it needs to be injected. Since we are using injection to *share* the same instance between several objects, we need to annotate these classes as well. The classes that need to be annotated are AtmUserInterface, CashSlot, MyWebDriver, and TestAccount.

The only class that presents a problem is CashSlot, because it is part of our production code, and we may not want to add an annotation just to keep our test framework happy. One alternative is to create a TestCashSlot that extends CashSlot, which keeps Guice annotations confined to our test code:

dependency_injection/guice/04/src/test/java/support/TestCashSlot.java

```
package support;

import cucumber.runtime.java.guice.ScenarioScoped;

import nicebank.CashSlot;

@ScenarioScoped
public class TestCashSlot extends CashSlot {
}
```

However, now we need to tell Guice to create an instance of TestCashSlot, not CashSlot. We can do this by changing the signatures of the @Inject-annotated constructors that currently take a CashSlot:

dependency_injection/guice/04/src/test/java/nicebank/CashSlotSteps.java

```
CashSlot cashSlot;

@Inject
public CashSlotSteps(TestCashSlot cashSlot) {
    this.cashSlot = cashSlot;
}
```

Now the test passes!

@Singleton Annotation

In the previous section we applied the @ScenarioScoped annotation indiscriminately to every class that is being managed by Guice. This means that a new instance will be created for every scenario run. That's not a problem for us at the moment because we have only one scenario, but this approach can be wasteful, especially if the objects are expensive to construct.

The other supported scope, @Singleton, tells Cucumber to construct only a single instance for all scenarios that will be run. This strategy should be used only when you're sure that the object stores no state that could allow one scenario to affect the outcome of another scenario. In our case we could apply this to our web driver, MyWebDriver:

dependency_injection/guice/05/src/test/java/support/MyWebDriver.java

```
import javax.inject.Singleton;

@Singleton
public class MyWebDriver extends EventFiringWebDriver{
    public MyWebDriver() {
        super(new FirefoxDriver());
    }
}
```

The test continues to pass, and now when we add more features they'll share the same browser instance.

There are lots more clever things that you can do with Guice, so take a look at the documentation online.

Spring in Your Steps

Spring is a popular and (very) large framework. Cucumber ships with an integration to Spring for handling step creation and DI, which doesn't require the whole Spring framework on the classpath. The current version of cucumber-spring is built against Spring 4 and is still under active development, so check the release notes online to see the changes in future releases of cucumber-spring.

Switching the DI Container

We'll start with the code as it was at the end of *PicoContainer Is Almost Invisible*, on page 13. As before, we'll change pom.xml to bring in the minimum dependency on Spring, our chosen DI framework:

dependency_injection/spring/01/pom.xml

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-spring</artifactId>
    <version>${cucumber.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${spring.version}</version>
    <scope>test</scope>
</dependency>
```

We'll also add a configuration file, cucumber.xml, as a test resource:

dependency_injection/spring/01/src/test/resources/cucumber.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <context:annotation-config/>
    <context:component-scan base-package="hooks, nicebank, support" />
</beans>
```

Here, we're telling Spring which of the base packages to scan for the classes that we're going to inject using the context:component-scan element. For this example we're interested in the hooks, nicebank, and support packages.

Some Spring Annotations

Spring uses the annotation @Autowired to identify what should be injected, rather than the @Inject annotation that other DI containers use. Unfortunately, the current cucumber-spring integration only supports no-argument constructors for step definition and hook classes. That means we'll have to use *field injection* instead of the constructor injection that we used in PicoContainer and Guice. Instead of having an annotated constructor that stores the injected objects that are passed to it, the DI container will inject the object directly into the annotated fields:

dependency_injection/spring/01/src/test/java/nicebank/CashSlotSteps.java

```
@Autowired
TestCashSlot cashSlot;
```

Now we just need to tell Spring which of our classes are candidates for injection by marking them with @Component and associating them with the correct scope. The cucumber-spring integration automatically associates all hooks and step definitions with a scope called cucumber-glue, but we'll need to associate any objects that we've created to share state with that scope too:

dependency_injection/spring/01/src/test/java/support/AtmUserInterface.java

```
@Component
@Scope("cucumber-glue")
public class AtmUserInterface implements Teller {
}
```

Now that we've done that, all our support classes are identified as *beans*, and Spring will happily instantiate and inject them on demand.

Run mvn clean test to check it's all working.

Some Spring Configuration Magic

The Spring configuration file is very powerful. In this section we'll see how we can use the configuration file to specify that a class is a bean without having to modify the Java code at all.

Beans with No-arg Constructors

Spring provides an alternative way to specify which classes are candidates for injection by configuring them as beans in the configuration file, in our case cucumber.xml. Since we're not saying anything about constructor arguments, these classes need to have a default (or no-arg) constructor:

dependency_injection/spring/02/src/test/resources/cucumber.xml
```
<bean class="support.AtmUserInterface" scope="cucumber-glue" />
<bean class="support.MyWebDriver" scope="cucumber-glue" />
<bean class="nicebank.CashSlot" scope="cucumber-glue" />
```

Now we can remove the annotations from AtmUserInterface and MyWebDriver, and delete TestCashSlot entirely (because we only created it to avoid having to edit the production code). We need to do a bit more to remove TestAccount, though, because we have to pass an account number to its constructor. One solution is to create AccountFactory and configure Spring to use that:

dependency_injection/spring/02/src/test/resources/cucumber.xml
```
<bean class="support.AccountFactory" factory-method="createTestAccount"
                                      lazy-init="true" scope="cucumber-glue" />
```

dependency_injection/spring/02/src/test/java/support/AccountFactory.java
```
package support;

import nicebank.Account;

public class AccountFactory {

    public static Account createTestAccount() {
        Account account = new Account(1234);
        account.saveIt();
        return account;
    }
}
```

Run mvn clean test and the scenario passes as before.

Bean Constructors with Arguments

In the previous section we saw how we could configure a class as a bean in the cucumber.xml file. That was fine for classes that had default, no-arg constructors. When we needed to call a constructor with an argument for Account we had to create an AccountFactory. Let's see if there's another way to do it. Take a look at MyWebDriver:

dependency_injection/spring/02/src/test/java/support/MyWebDriver.java

```java
package support;

import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.events.EventFiringWebDriver;

public class MyWebDriver extends EventFiringWebDriver{
    public MyWebDriver() {
        super(new FirefoxDriver());
    }
}
```

As you can see, its only difference from Selenium's EventFiringWebDriver is that the constructor is wired to use a Firefox browser. As a last example, let's use Spring to allow us to get rid of MyWebDriver entirely. We'll start by telling Spring how to create an EventFiringWebDriver for our scenarios to use:

dependency_injection/spring/03/src/test/resources/cucumber.xml

```xml
<bean class="org.openqa.selenium.support.events.EventFiringWebDriver"
                                    scope="cucumber-glue" destroy-method="close">
    <constructor-arg>
        <bean class="org.openqa.selenium.firefox.FirefoxDriver"
                                    scope="cucumber-glue"/>
    </constructor-arg>
</bean>
```

Now we need to replace all references to MyWebDriver in our test code with references to EventFiringWebDriver. Here's an example from WebDriverHooks:

dependency_injection/spring/03/src/test/java/hooks/WebDriverHooks.java

```java
@Autowired
private EventFiringWebDriver webDriver;
```

Another run of mvn clean test will verify that our scenario still passes.

We'll see more uses of Spring in Chapter 12, *Working with Web Applications*, on page ?.

CDI with Weld

The Java community have been working on a standardized approach to many of the challenges that software developers have to deal with. One of these is the Contexts and Dependency Injection for Java EE (CDI) standard, of which there are several implementations. Weld and OpenEJB are the two implementations that Cucumber is integrated with, but since they are so similar we'll only show an example using Weld.

Again, we'll start with the code as it was at the end of *PicoContainer Is Almost Invisible*, on page 13. As before we change pom.xml to bring in the minimum dependency on our chosen DI framework:

dependency_injection/weld/01/pom.xml

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-weld</artifactId>
    <version>${cucumber.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>javax.enterprise</groupId>
    <artifactId>cdi-api</artifactId>
    <version>${cdi-api.version}</version>
</dependency>
<dependency>
    <groupId>org.jboss.weld.se</groupId>
    <artifactId>weld-se</artifactId>
    <version>${weld.version}</version>
</dependency>
```

In the same way as we have seen with the preceding DI containers, we need to tell Weld/CDI how and where to inject our objects. We show where we want our objects injected using the @Inject annotation and we indicate that we want to share a single instance of each object using the @Singleton annotation. Examples of this can be seen in CashSlotSteps:

dependency_injection/weld/01/src/test/java/nicebank/CashSlotSteps.java

```
CashSlot cashSlot;
@Inject
public CashSlotSteps(TestCashSlot cashSlot) {
    this.cashSlot = cashSlot;
}
```

Both of these annotations come from the javax.inject package, so this code is identical to the Guice version. What is different, however, is that we now need tell the container about the classes that need injected. Guice used the classpath to find candidates for injection, but for Weld/CDI we need to create bean.xml:

dependency_injection/weld/01/src/test/resources/META-INF/beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

You'll notice that our bean.xml has almost nothing in it. We could use it to configure our beans in much the same way that we did in *Some Spring Configuration Magic,* on page 20. Since this book is about Cucumber, not Weld/CDI, we're leaving that as an exercise for the reader.

What We Just Learned

Congratulations, you're ready to inject dependencies!

In this chapter we've seen how Cucumber's use of dependency injection simplifies the management of the graph of objects needed to run our scenarios. We looked at most of the DI containers that Cucumber is integrated with and saw that PicoContainer is probably the simplest to use, if you aren't already using one of the others in your project. Most of the other DI containers require some form of annotation and configuration. Once we choose a DI container, no matter which one, it takes over responsibility for creating all our step definition and hook objects.

As we applied DI to our example we pushed more of the responsibilities for creating and initializing our objects onto the container. That left us with fewer, smaller, more cohesive objects. This style takes a bit of getting used to, but leads to a more *composable* architecture that can be easier to maintain and extend.

Try This

Using your DI container of choice, try the following:

- If we had more than one scenario, the browser would start up for each one, which takes time. How would you share a single browser session across all scenarios?
- Most of our examples used constructor injection. Change them to use field injection—you should then be able to delete the constructor entirely.
- Choose a DI container and browse its documentation. Experiment with different ways of creating the objects that need to be injected.