Why field injection is evil

November 22nd, 2013

I'm quite frequently getting pulled into discussions on Twitter about the different flavors of Dependency Injection. Also, I've repeatedly expressed my distaste for field injection but as Twitter is not the right communication channel to give an in-depth rational about my opinion. So here we go.

Let's discuss this stuff with a bit of (quite generic) context: we want to code a component that has a collaborator. As we know, Dependency Injection is the means to connect the two the apparently easiest way to achieve this is the following:

```
class MyComponent {

  @Inject MyCollaborator collaborator;

  public void myBusinessMethod() {
    collaborator.doSomething();
  }
}
```

So what's wrong with this code?

It's NullpointerExceptions begging to happen

Well, first of all it's broken by default. What's the API you get to create instances of this class which you will need in your unit test?

```
MyComponent component = new MyComponent();
component.myBusinessMethod(); // -> NullPointerException
```

The core of the problem here is that code you've written allows clients to create instances of the class in an invalid state. The whole purpose of a type is clients being able to rely on the invariants it enforces. It's one of the reasons you use an `EmailAddress` type over a plain `String` to represent email adresses in your code: clients can be sure the instance they get is a valid email address as the value object enforces this constraint during construction. A `String` can potentially be anything, validated or not - how do you know?

So you can probably guess what this is heading to: constructor injection. Let's rewrite the code shown before in a way it actually enforces the traits I just outlined:

```
class MyComponent {

  private final MyCollaborator collaborator;

  @Inject
  public MyComponent(MyCollaborator collaborator) {
    Assert.notNull(collaborator, "MyCollaborator must not be null!");
    this.collaborator = collaborator;
  }

  public void myBusinessMethod() {
    collaborator.doSomething();
  }
}
```

"Ohh boy!", I can hear you say, "So much boilerplate code!". Let me get back to this argument in a bit and just recap, what we've achieved:

1. *You can only create instances of `MyComponent` by providing a `MyCollaborator`.* You *force* clients to provide mandatory dependencies, making sure every object created is in a valid state after construction.
2. *You communicate mandatory dependencies publicly.* Remember when we bashed service locators for hiding dependencies in the implementations? Field injections is just lipstick on the pig in that regard. You still don't know about the dependencies just looking at the public interface of the type (e.g. while skimming the JavaDoc).

Especially if you share code amongst projects field injection based types become a “run-and-wait-for-the-NPE-to-happen,-declare-missing-beans-and-repeat” kind of approach.
3. *Final fields also add to the immutable nature application components get.* You can clearly distinguish between mandatory dependencies (`final`) and optional ones (non-`final`) usually injected through setter injection.

An often faced argument I get is: “Constructors just get too verbose if I have 6 or 7 dependencies. With fields only, this is fine”. Awesome, you’ve effectively worked around a clear indicator that the code you write is doing way too much. An increase in the number of dependencies a type has *should* hurt, as it makes you think about whether you should split up the component into multiple ones. You want to really cure the pain, not blindly apply pain killers to it, don’t you?

Testability

Coming back to the amount of code to be written for the constructor injection based variant. Assuming we sticked to the field injection variant, we would have much less code to write, right? Well, I guess you’re writing tests for your code, right? So how do you actually inject a dependency into your component while testing?

```
MyCollaborator collaborator = … // mock dependency
MyComponent component = new MyComponent();
// Inject dependency by some reflection magic
component.myBusinessMethod();
```

Reflection is the answer here, fine. No matter how comfortable you make this by using a helper method or the like, it’s still a messy workaround isn’t it? Especially if the alternative to that is a simple:

```
MyCollaborator collaborator = … // mock dependency
MyComponent component = new MyComponent(collaborator);
component.myBusinessMethod();
```

You get code completion on the constructor call and when you add a dependency to the type under test, refactoring applies, no unset dependencies etc.

Boilerplate buster Lombok

Admittedly I’ve been turned off by the amount of code to be written for constructor injection in the first place as well. This is clearly a shortcoming of Java as a languages. Unfortunately a lot of good OO practices like value objects, favoring delegation over inheritance and constructor DI are significantly easier to implement in languages like Scala.

However, Project Lombok is a really awesome helper to reduce the amount of boilerplate you have to write to do “the right things” (™). There’s a ton of helpful features in Lombok but I want to concentrate on the one related to the discussion here. With Lombok the constructor DI based variant of my component up there would look something like this:

```
@RequiredArgsConstructor(onConstructor = @__(@Inject))
class MyComponent {

  private final @NonNull MyCollaborator collaborator;

  public void myBusinessMethod() {
    collaborator.doSomething();
  }
}
```

The `@RequiredArgsAnnotation` will cause a constructor being added during the compilation process taking all final fields as parameters. The additional `@NonNull` annotation will cause the parameter be checked for `null` as well. The weird looking `onConstructor` is Lombok’s way of letting you add annotations to the constructor generated. So with an additional annotation you effectively get the API we’re looking for.

To summarize, here are the results of the comparison that I get to:

Field injection:

```
++ less code to write
-- unsafe code
```

```
 - more complicated to test
```

Constructor injection:

```
++ safe code
 - more code to write (see the hint to Lombok)
 + easy to test
```

Related Posts

- 01 Sep 2023 » Conference Autumn
- 25 Jul 2023 » Sliced Onion Architecture
- 02 Jul 2021 » Moduliths 1.1 released

Oliver Drotbohm
Soul Power!

github.com/odrotbohm
twitter.com/odrotbohm

info@odrotbohm.de