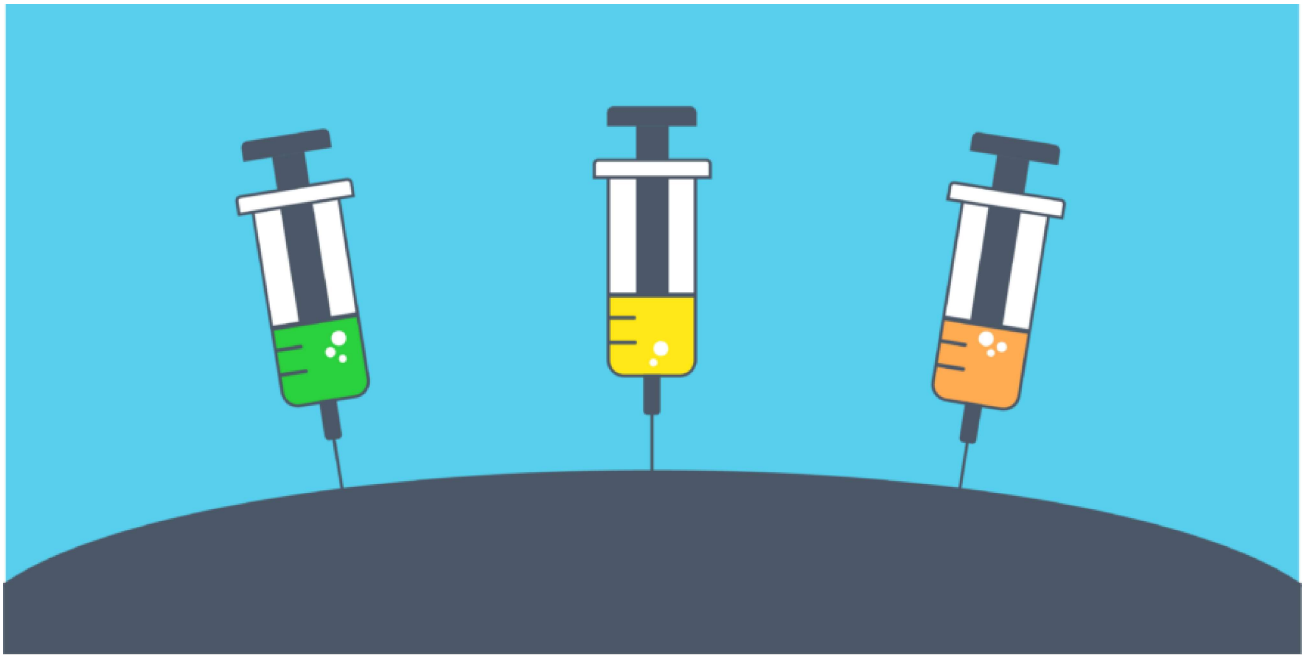


Design Patterns Explained – Dependency Injection with Code Examples

THORBEN JANSSEN | JUNE 19, 2018 |
DEVELOPER TIPS, TRICKS & RESOURCES (HTTPS://STACKIFY.COM/DEVELOPERS/)

Dependency injection is a programming technique that makes a class independent of its dependencies. It achieves that by decoupling the usage of an object from its creation. This helps you to follow SOLID's dependency inversion (https://stackify.com/dependency-inversion-principle/) and single responsibility principles (https://stackify.com/solid-design-principles/).

As I explained in my previous articles about the SOLID design principles, their goal is to improve the reusability of your code. They also aim to reduce the frequency with which you need to change a class. Dependency injection supports these goals by decoupling the creation of the usage of an object. That enables you to replace dependencies without changing the class that uses them. It also reduces the risk that you have to change a class just because one of its dependencies changed.

The dependency injection technique is a popular alternative to the service locator pattern (https://stackify.com/service-locator-pattern/). A lot of modern application frameworks implement it. These frameworks provide the technical

parts of the technique so that you can focus on the implementation of your business logic. Popular examples are:

- Weld (http://weld.cdi-spec.org/), the reference implementation of Jakarta EE's Context and Dependency Injection for Java (CDI) (http://www.cdi-spec.org/) specification
- Spring (https://spring.io/)
- Guice (https://github.com/google/guice)
- Play framework (https://www.playframework.com/)
- Dagger (https://square.github.io/dagger/)

Tip: Find application errors and performance problems instantly with Stackify Retrace

Troubleshooting and optimizing your code is easy with integrated errors, logs and code level performance insights.

Try today for free (https://cta-redirect.hubspot.com/cta/redirect/207384/0198bd97-1455-4b24-b560-8944c63d3f3f)

The dependency injection technique

You can introduce interfaces to break the dependencies between higher and lower level classes. If you do that, both classes depend on the interface and no longer on each other. I explained this approach in great details in my article about the dependency inversion principle (https://stackify.com/dependency-inversion-principle/).

That principle improves the reusability of your code and limits the ripple effect if you need to change lower level classes. But even if you implement it perfectly, you still keep a dependency on the lower level class. The interface only decouples the usage of the lower level class but not its instantiation. At some place in your code, you need to instantiate the implementation of the interface. That prevents you from replacing the implementation of the interface with a different one.

Product Pricing Solutions
Learn Login
(https://stackify.com/)

The goal of the dependency injection technique is to remove this dependency by separating the usage from the creation of the object. This reduces the amount of required boilerplate code and improves flexibility.

But before we take a look at an example, I want to tell you more about the dependency injection technique.

The 4 roles in dependency injection

If you want to use this technique, you need classes that fulfill four basic roles. These are:

1. The **service** you want to use.
2. The **client** that uses the service.
3. An **interface** that's used by the client and implemented by the service.
4. The **injector** which creates a service instance and injects it into the client.

You already implement three of these four roles by following the dependency inversion principle. The service and the client are the two classes between which the dependency inversion principle intends to remove the dependency by introducing an interface.

You can skip the interface role and inject the service object directly into the client. But by doing that, you break with the dependency inversion principle and your client has an explicit dependency on the service class. In some situations, this might be ok. But most often, it's better to introduce an interface to remove the dependency between the client and the service implementation.

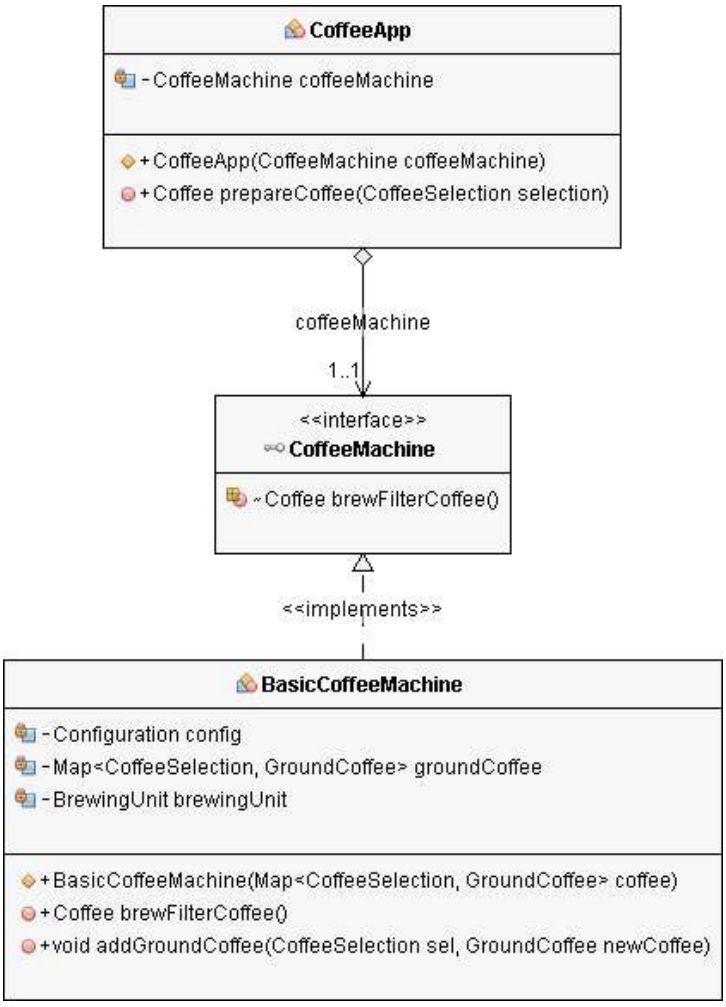
The injector is the only role that isn't required by the dependency inversion principle. But that's not an issue because you don't need to implement it. All frameworks that I listed at the beginning of this article provide ready-to-use implementations of it.

As you can see, dependency injection is a great fit for applications that follow the dependency inversion principle. You already implement most of the required roles, and the dependency injection technique enables you to remove the last dependency to the service implementation.

Using Dependency Injection to make the CoffeeApp more flexible

I used the CoffeeApp example in my article about the dependency inversion principle (https://stackify.com/dependency-inversion-principle/). Let's change it so that it uses the dependency injection technique.

This small application enables you to control different coffee machines, and you can use it to brew a fresh cup of filter coffee. It consists of a *CoffeeApp* class that calls the *brewFilterCoffee* method on the *CoffeeMachine* interface to brew a fresh cup of coffee. The class *BasicCoffeeMachine* implements the *CoffeeMachine* interface.



As you can see in the diagram, this application already follows the dependency inversion principle. It also provides three of the four roles required by the dependency inversion technique:

- The *CoffeeApp* implements the client role.
- The *BasicCoffeeMachine* class acts as the services.
- The *CoffeeMachine* interface fulfills the interface role.

(https://stackify.com/)
Product Pricing Solutions Learn Login

The only thing that's missing is the injector. I will introduce an implementation of this role by using the Weld framework. It's the reference implementation of Jakarta EE's CDI specification. Since version 2.0, you can bootstrap it in a Java SE environment without adding a huge framework stack.

CDI 2.0 is part of all Jakarta EE 8 application servers. If you deploy your application into such a server, you don't need to bootstrap the CDI container.

And if you're using a different application framework, e.g., Spring, you should use the dependency injection implementation provided by that framework. It enables you to use the same concepts that I show you in the following example.

Try Stackify's free code profiler, Prefix (https://stackify.com/prefix), to write better code on your workstation. Prefix works with .NET, Java, PHP, Node.js, Ruby, and Python.

Bootstrapping CDI

Before you can use CDI's dependency injection feature, you need to bootstrap a CDI container. But don't worry, Weld makes that very simple. It provides a build-in main method (http://docs.jboss.org/weld/reference/latest-master/en-US/html_single/#_bootstrapping_cdi_se) which bootstraps a container for you. You can run it by executing the following command.

```
java org.jboss.weld.environment.se.StartMain
```

But bootstrapping the CDI container without an application doesn't make much sense. You can add your application in two steps which require almost no code.

Weld-SE dependency

You need to add a dependency to weld-se to your application.

```
<dependency>
        <groupId>org.jboss.weld.se</groupId>
        <artifactId>weld-se-core</artifactId>
        <version>3.0.4.Final</version>
</dependency>
```

Writing and running the CoffeeAppStart class

Product Pricing Solutions
Learn Login
(https://stackify.com/)

The next step is the only implementation task required to run your application in the CDI container. You need to write a method that observes the *ContainerInitialized* event. I did that in the *CoffeeAppStarter* class. Similar to the examples in my articles about the dependency inversion principle and the service locator pattern (https://stackify.com/service-locator-pattern/), this class starts the *CoffeeApp*.

```
@Singleton
public class CoffeeAppStarter {

	private CoffeeApp app;

	@Inject
	public CoffeeAppStarter (CoffeeApp app) {
		this.app = app;
	}

	public void startCoffeeMachine(@Observes ContainerInitialized event) {
		try {
			app.prepareCoffee(CoffeeSelection.FILTER_COFFEE);
		} catch (CoffeeException e) {
			e.printStackTrace();
		}
	}
}
```

Observing events in CDI is simple and powerful. You just need to annotate a method parameter with *@Observes*. The container will call this method as soon as someone fires an event for the type of the annotated parameter. Weld fires the *ContainerInitialized* event after it started the CDI container. So, this method will be called during application startup.

Using dependency injection with CDI

You probably already recognized the *@Inject* annotation in the previous code snippet. It tells the CDI container to inject a *CoffeeApp* object when the constructor of the *CoffeeAppStarter* class gets called. So, you can use the *CoffeeApp app* attribute in the *startCoffeeMachine* method to brew a cup of filter coffee.

This approach obviously doesn't follow the dependency inversion principle because the interface is missing. But I think this is one of the rare occasions in which it is acceptable to inject the service implementation directly. The only task

Product Pricing Solutions
Learn Login
(https://stackify.com/)

of the *CoffeeAppStarter* class is to start the coffee machine by calling the *prepareCoffee* method on the injected *CoffeeApp* object. I don't see any need to add another abstraction to make the *CoffeeApp* replaceable.

But that's not the case for the *coffeeMachine* attribute of the *CoffeeApp* class. In the future, this application will need to control different kinds of coffee machines. I want to make it as easy as possible to replace them. That's why I introduced the *CoffeeMachine* interface in the previous articles. As you can see in the following code snippet, the *CoffeeApp* class only depends on the *CoffeeMachine* interface. It has no dependency on any interface implementation.

```
public class CoffeeApp {

	private CoffeeMachine coffeeMachine;

	@Inject
	public CoffeeApp(CoffeeMachine coffeeMachine) {
		this.coffeeMachine = coffeeMachine;
	}

	public Coffee prepareCoffee(CoffeeSelection selection)
			throws CoffeeException {
		Coffee coffee = this.coffeeMachine.brewFilterCoffee();
		System.out.println("Coffee is ready!");
		return coffee;
	}
}
```

In the previous articles, the *CoffeeAppStarter* class had to instantiate a specific implementation of the *CoffeeMachine* interface. It provided that object as a constructor parameter while instantiating a *CoffeeApp* object.

Constructor injection now enables you to replace the compile time dependency to a specific implementation class with a runtime dependency to any implementation class. That makes it very easy to replace the *CoffeeMachine* implementation. You only need to add a different implementation of the *CoffeeMachine* interface to your classpath when you start the application.

Summary

Stackify's dependency inversion principle (https://stackify.com/dependency-inversion-principle/) introduces interfaces between a higher-level class and its dependencies. That decouples the higher-level class from its dependencies so that you can change the code of a lower-level class without changing the code that uses it. The only code that uses a dependency directly is the one that instantiates an object of a specific class that implements the interface.

The dependency injection technique enables you to improve this even further. It provides a way to separate the creation of an object from its usage. By doing that, you can replace a dependency without changing any code and it also reduces the boilerplate code in your business logic.

With APM, server health metrics, and error log integration, improve your application performance with Stackify Retrace. Try your free two week trial today (https://s1.stackify.com/account/createclient?_ga=2.57834090.1973545731.1588002198-1971815645.1570122931&_gac=1.238281396.1584390051.EAIaIQobChMIvenD6Oif6AI

Improve Your Code with Retrace APM

Stackify's APM tools are used by thousands of .NET, Java, PHP, Node.js, Python, & Ruby developers all over the world.
Explore Retrace's product features to learn more.

(/retrace-application-performance-management/)

App Performance Management (https://stackify.com/retrace-application-performance-management/)

(/retrace-code-profiling/)

Code Profiling (https://stackify.com/retrace-code-profiling/)

(https://stackify.com/)

Product Pricing Solutions

Learn Login

(/retrace-error-monitoring/)

Error Tracking (https://stackify.com/retrace-error-monitoring/)

(/retrace-log-management/)

Centralized Logging (https://stackify.com/retrace-log-management/)

(/retrace-app-metrics/)

App & Server Metrics (https://stackify.com/retrace-app-metrics/)

Learn More
(/retrace/)

About the Author | Latest Posts

About Thorben Janssen

Thorben is an independent trainer and author of the Amazon bestselling book *Hibernate Tips - More than 70 solutions to common Hibernate problems*. He writes about Java EE related topics on his blog Thoughts on Java (https://www.thoughts-on-java.org).

(https://twitter.com/thjanssen123)

(https://www.linkedin.com/in/thorbenjanssen/)

Email

Sign Up Today

Topics/Keywords

ASP.NET (https://stackify.com/?tag=asp.net,net-core)

.NET Core (https://stackify.com/content/net-core/)

Java (https://stackify.com/content/java/)

Azure (https://stackify.com/content/azure/)

AWS (https://stackify.com/content/AWS/)

Cloud (https://stackify.com/?tag=cloud,azure,aws)

Product Updates (https://stackify.com/stackify/)

App Monitoring (https://stackify.com/?tag=monitoring,apm)

App Performance Tips (https://stackify.com/?tag=performance,profiler,apm)

Error Handling (https://stackify.com/?tag=exception,exceptions,error,errors)

Logging Tips (https://stackify.com/?tag=logs,logging)

DevOps (https://stackify.com/content/DevOps/)

Popular Posts

ASP.NET Performance: 9 Types of Tools You Need to Know!

How to Troubleshoot IIS Worker Process (w3wp) High CPU Usage

How to Monitor IIS Performance: From the Basics to Advanced IIS Performance Monitoring

SQL Performance Tuning: 7 Practical Tips for Developers

Looking for New Relic Alternatives & Competitors? Learn Why Developers Pick Retrace

Recent Posts

Picking The Right Programming Language for Your Applicatio... (https://stackify.com/picking-the-right-programming-language-for-your-application/)

4 API Security Best Practices To Safeguard Sensitive Data... (https://stackify.com/4-api-security-best-practices-to-safeguard-sensitive-data/)

10 Myths About Custom Website Development (https://stackify.com/10-myths-about-custom-website-development/)

(https://stackify.com/10-myths-about-custom-website-development/)

(https://stackify.com/)

Mistakes to Avoid in Software Development Projects...

(https://stackify.com/mistakes-to-avoid-in-software-development-projects/)

Mobile Cloud Computing: Overview, Challenges and Scope...

(https://stackify.com/mobile-cloud-computing-overview-challenges-and-scope/)

(/stackify-developer-ebooks/)

(https://stackify.com/guest-blogging-guidelines/)

Get In Touch

Contact Us

Request a Demo

Start Free Trial

Products

.NET Monitoring

Java Monitoring

PHP Monitoring

Node.js Monitoring

Ruby Monitoring

Python Monitoring

Retrace vs New Relic

Retrace vs Application Insights

Solutions

Application Performance Management

Centralized Logging

Code Profiling

Full Transaction Tracing

Application & Server Monitoring

Real User Monitoring

Retrace Deployment Tracking

For Developers

For DevOps

Resources

What is APM?

Pricing

Case Studies

Blog

Documentation

Free eBooks

Company

About Us

News

Careers

GDPR

Security Information

Terms & Conditions

Privacy Policy

PO Box 2159
Mission, KS 66201
816-888-5055 (tel:18168885055)

© 2022 Netreo