Design Patterns Explained – Service Locator Pattern with Code Examples

THORBEN JANSSEN | JUNE 11, 2018 |

The service locator pattern is a relatively old pattern that was very popular with Java EE. Martin Fowler described it in 2004 . The goal of this pattern is to improve the modularity of your application by removing the dependency between the client and the implementation of an interface.

Interfaces are one of the most flexible and powerful tools to decouple software components and to improve the maintainability of your code. I wrote a lot about them in my series about the SOLID design principles:

- Following the , you use one or more interfaces to ensure that your component is open for extension but closed for modification.
- The requires you to implement your interfaces in a way that you can replace its implementations without changing the code that uses the interface.
- The ensures that you design your interfaces so that clients don't depend on parts of the interface.
- And to follow the , you need to introduce an interface as an abstraction between a higher and a lower level component to split the dependency between both components.

All of these principles enable you to implement robust and maintainable applications. But they all share the same problem: At some point, you need to provide an implementation of the interface. If that's done by the same class that uses the interface, you still got a dependency between the client and the implementation of the interface.

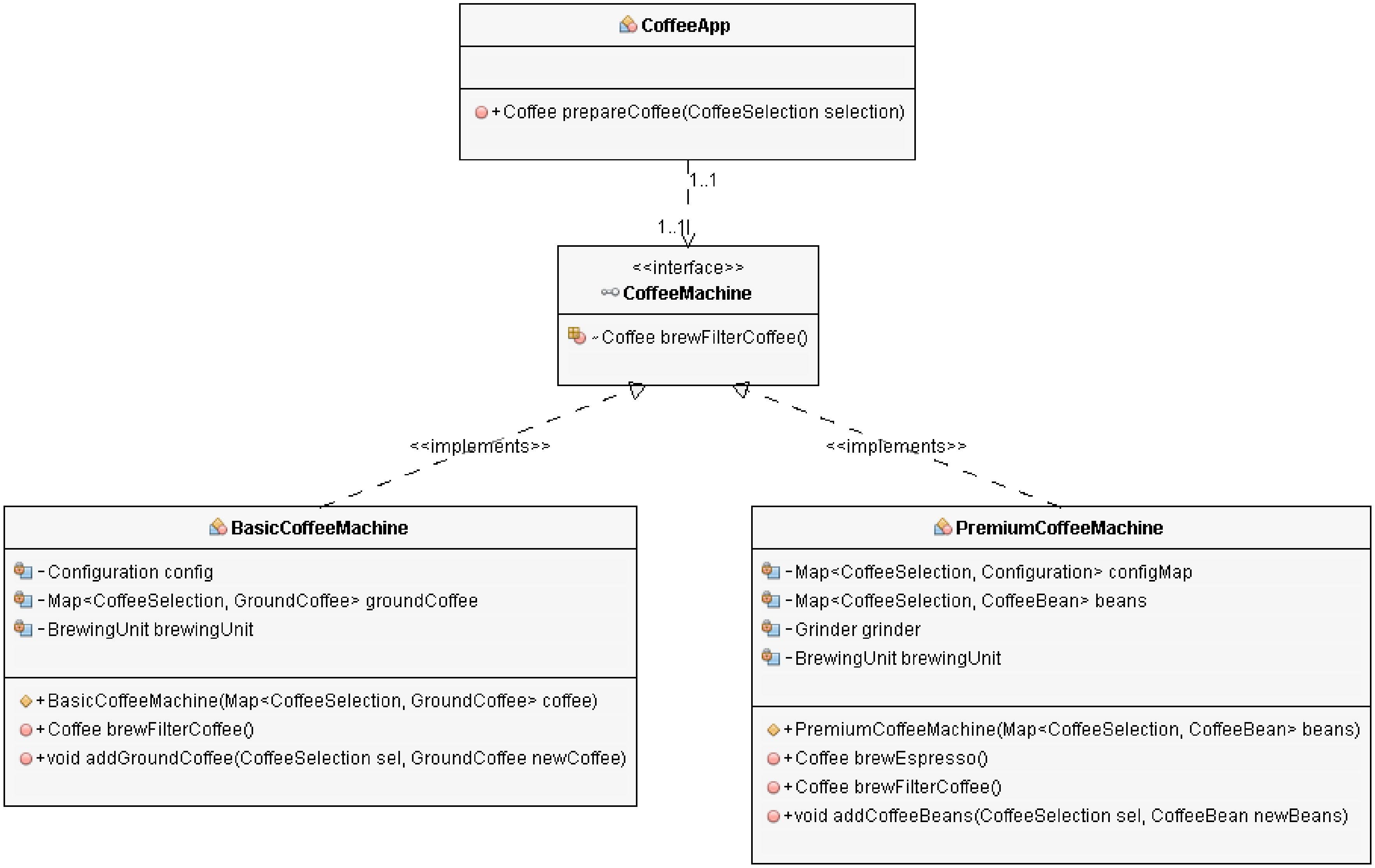
The service locator pattern is one option to avoid this dependency. It acts as a central registry that provides implementations of different interfaces. By doing that, your component that uses an interface no longer needs to know the class that implements the interface. Instead of instantiating that class itself, it gets an implementation from the service locator.

That might seem like a great approach, and it was very popular with Java EE. But over the years, developers started to question this pattern. You don't get the decoupling of the client and the implementation of the interface for free, and there are other options to achieve the same goal, e.g., the Dependency Injection pattern. But that doesn't mean that this pattern is no longer valid. Let's first take a closer look at the service locator pattern before we dive into the details of that discussion.

The service locator pattern

In this article, I use the same example as I used in my article about the . It consists of a *CoffeeApp* class that uses the *CoffeeMachine* interface to brew a cup of coffee with different coffee machines. There are two machines available, the *BasicCoffeeMachine* and the

PremiumCoffeeMachine class. Both of them implement the *CoffeeMachine* interface.



CoffeeApp
+Coffee prepareCoffee(CoffeeSelection selection)
1..1
1..1
<<interface>>
CoffeeMachine
~Coffee brewFilterCoffee()
<<implements>>
<<implements>>
BasicCoffeeMachine
-Configuration config
-Map<CoffeeSelection, GroundCoffee> groundCoffee
-BrewingUnit brewingUnit
+BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee)
+Coffee brewFilterCoffee()
+void addGroundCoffee(CoffeeSelection sel, GroundCoffee newCoffee)
PremiumCoffeeMachine
-Map<CoffeeSelection, Configuration> configMap
-Map<CoffeeSelection, CoffeeBean> beans
-Grinder grinder
-BrewingUnit brewingUnit
+PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans)
+Coffee brewEspresso()
+Coffee brewFilterCoffee()
+void addCoffeeBeans(CoffeeSelection sel, CoffeeBean newBeans)

As you can see in the diagram, the *CoffeeMachine* interface ensures that there are no dependencies between the *CoffeeApp*, *BasicCoffeeMachine*, and *PremiumCoffeeMachine*. All three classes only depend on the interface. That improves the maintainability of all classes, and enables you to introduce new coffee machines without changing the existing code.

But it also introduces a new problem: How does the *CoffeeApp* get an implementation of the *CoffeeMachine* interface without creating a dependency to that specific class? In my article about the Dependency Inversion Principle, I provided a *CoffeeMachine* object as a constructor parameter to the *CoffeeApp*.

```
public class CoffeeApp {
    private CoffeeMachine coffeeMachine;

    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;
    }
}
```

That moved the task of the object instantiation and the dependency from the *CoffeeApp* to the *CoffeeAppStarter* class.

```java
public class CoffeeAppStarter {

    public static void main(String[] args) {
        // create a Map of available coffee beans
        Map<CoffeeSelection, CoffeeBean> beans = new HashMap<CoffeeSelection, CoffeeBean>();
        beans.put(CoffeeSelection.ESPRESSO, new CoffeeBean(
            "My favorite espresso bean", 1000));
        beans.put(CoffeeSelection.FILTER_COFFEE, new CoffeeBean(
            "My favorite filter coffee bean", 1000));

        // get a new CoffeeMachine object
        PremiumCoffeeMachine machine = new PremiumCoffeeMachine(beans);

        // Instantiate CoffeeApp
        CoffeeApp app = new CoffeeApp(machine);

        // brew a fresh coffee
        try {
            app.prepareCoffee(CoffeeSelection.ESPRESSO);
        } catch (CoffeeException e) {
            e.printStackTrace();
        }
    }
}
```

Introducing the service locator

The service locator pattern provides a different approach. It acts as a singleton registry for all services that are used by your application, and enables the *CoffeeApp* to request an implementation of the *CoffeeMachine* interface.

There are different options to implement the service locator. You can use a static service locator that uses a field for each service to store an object reference. Or you can create a dynamic one that keeps a *java.util.Map* with all service references. This one can be dynamically extended to support new services.

Both implementations follow the same approach, but the static service locator is a little bit easier to understand. So, I will use the static one in my coffee machine example.

Adding a static service locator

Before you implement your service locator, you need to decide which interface implementation it shall return, or if you want to use an external configuration parameter that specifies the name of the class that implements the interface. The latter approach is more flexible, but also more complex. To keep the example easy to understand, I will instantiate a *PremiumCoffeeMachine* object without using any external configuration parameters. If you decide to use the service locator pattern in your application, I recommend to make it as configurable as possible and to provide the name of the class as a configuration parameter.

As I explained earlier, the service locator is a singleton. The *CoffeeServiceLocator* class, therefore, only has a private constructor and keeps a reference to itself. You can get a *CoffeeServiceLocator* instance by calling the static *getInstance* method on the *CoffeeServiceLocator* class.

```
public class CoffeeServiceLocator {

    private static CoffeeServiceLocator locator;

    private CoffeeMachine coffeeMachine;

    private CoffeeServiceLocator() {
        // configure and instantiate a CoffeeMachine
        Map<CoffeeSelection, CoffeeBean> beans = new HashMap<CoffeeSelection, CoffeeBean>();
        beans.put(CoffeeSelection.ESPRESSO, new CoffeeBean(
            "My favorite espresso bean", 1000));
        beans.put(CoffeeSelection.FILTER_COFFEE, new CoffeeBean(
            "My favorite filter coffee bean", 1000));
        coffeeMachine = new PremiumCoffeeMachine(beans);
    }

    public static CoffeeServiceLocator getInstance() {
        if (locator == null) {
            locator = new CoffeeServiceLocator();
        }
        return locator;
    }

    public CoffeeMachine coffeeMachine() {
        return coffeeMachine;
    }
}
```

In the next step, you can refactor the *CoffeeApp.* It can now get the *CoffeeMachine* object from the *CoffeeServiceLocator*, and not as a constructor parameter.

```
public class CoffeeApp {


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

That's all you need to do to introduce the service locator pattern into the coffee machine example. As you have seen, the implementation of a simple service locator class isn't complicated. You just need a singleton that returns instances of the different service interfaces used in your application.

Arguments against the service locator pattern

After we discussed the implementation details of the service locator pattern, it's time to take a closer look at the discussions about the pattern and its alternatives.

As you will see in the following paragraphs, there are several valid concerns about this pattern. Some of them can be avoided by using the Dependency Injection pattern. If you're building your application using Jakarta EE, previously called Java EE, or Spring, you already have a very powerful Dependency Injection implementation. In these situations, it's better to use the Dependency Injection pattern instead of the service locator pattern. If that's not the case, the service locator pattern is still a good option to remove the dependency between the client and the implementation of an interface.

The three most common arguments against the service locator pattern are:

- All components need to have a reference to the service locator, which is a singleton.
- The service locator makes the application hard to test.
- A service locator makes it easier to introduce breaking changes in interface implementations.

All components need to reference the service locator

This is a valid concern. If you use your components in different applications and environments, introducing a dependency to your service locator class might be problematic because the class might not exist in all environments. You can try to avoid that by adding one or more interfaces that abstract the service locator and enable you to provide an .

Implementing the service locator as a singleton can also create scalability problems in highly concurrent environments.

You can avoid both problems by using the Dependency Injection pattern instead of the service locator pattern. Both patterns have the same goal, but use very different approaches to achieve them. I will explain the Dependency Injection pattern in more details in my next article.

It makes the application hard to test

The validity of this argument against the service locator pattern depends on the quality of your code. As long as you implement your service locator carefully, you can replace it during your tests with an implementation that provides test stubs for different services. That might not be as easy as it could be if you had used the Dependency Injection pattern, but it's still possible.

Higher risk to introduce breaking changes

That is a general issue that is caused by the interface abstraction of your service and not by the service locator pattern. As soon as you implement a reusable component and use an interface as an abstraction to make the implementation replaceable, you are taking the risk that the next change on your interface implementation will break some external component. That is the price you have to pay if you want to create reusable and replaceable code.

The best way to handle this risk is to create a well-defined contract for your interface. You then need to document this contract and implement a test suite that validates it. This test suite belongs to the interface and should be used to verify all implementations of it. That enables you to find breaking changes before they cause runtime errors in production.

Summary

You can choose between different patterns that enable you to decouple a client from the implementation of an interface. The service locator pattern is one of them.

This pattern introduces a singleton registry that provides an instance of a service interface. That moves the dependency to the interface implementations from the client of the interface to the service locator class.

The service locator pattern is relatively old and still valid. But and provide powerful implementations of the Dependency Injection pattern. This pattern has the same goal as the service locator pattern, and I will explain it in more details in my next article. If you are building your application with Jakarta EE or Spring, you should prefer the Dependency Injection pattern.

With APM, server health metrics, and error log integration, improve your application performance with Stackify Retrace.

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 .

Search Stackify

Topics/Keywords

Popular Posts

Recent Posts

Get In Touch

Products

Solutions

Resources

Company

PO Box 2159
Mission, KS 66201