WIKIPEDIA

Service locator pattern

The **service locator pattern** is a design pattern used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer. This pattern uses a central registry known as the "service locator", which on request returns the information necessary to perform a certain task.[1] Proponents of the pattern say the approach simplifies component-based applications where all dependencies are cleanly listed at the beginning of the whole application design, consequently making traditional dependency injection a more complex way of connecting objects. Critics of the pattern argue that it is an anti-pattern which obscures dependencies and makes software harder to test.[2]

Contents

Advantages

- The "service locator" can act as a simple run-time linker. This allows code to be added at run-time without re-compiling the application, and in some cases without having to even restart it.
- Applications can optimize themselves at run-time by selectively adding and removing items from the service locator. For example, an application can detect that it has a better library for reading JPG images available than the default one, and alter the registry accordingly.
- Large sections of a library or application can be completely separated. The only link between them becomes the registry.
- An application may use multiple structured service locators purposed for particular functionality/testing. Service locator does not mandate one single static class per process
- The solution may be simpler with service locator (vs. dependency injection) in applications with well-structured component/service design. In these cases the disadvantages may actually be considered as an advantage (e.g. no need to supply various dependencies to every class and maintain dependency configurations)

Disadvantages

- The registry hides the class' dependencies, causing run-time errors instead of compile-time errors when dependencies are missing (similar to using Dependency injection). But each library is compiled, just the discovery of the concrete Class might not be found and cause an error, it's more a deployment issue than a Service Locator issue.
- The registry makes code harder to test, since all tests need to interact with the same global service locator class to set the fake dependencies of a class under test. However, this is easily overcome by injecting application classes with a single service locator interface. Simulator can be implemented to simulate each interface provided by the service locator, so it's easy to swap the real implementation with a simulator.

See also

- Dependency injection
- Dependency inversion principle
- Java Naming and Directory Interface

References

1. http://martinfowler.com/articles/injection.html#UsingAServiceLocator
2. Seemann, Mark. "Service Locator is an Anti-Pattern" (http://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/). *blog.ploeh.dk*. Retrieved 2017-06-01.

External links

- Sample code (http://www.oracle.com/technetwork/java/servicelocator-137181.html)
- In Defense of Service Locator (http://bayou.io/draft/In_Defense_of_Service_Locator.html)
- Game Programming Patterns: Service Locator (http://gameprogrammingpatterns.com/service-locator.html)
- Dependencies In Disguise (https://thephp.cc/news/2015/09/dependencies-in-disguise)
- Software Engineering Myths and Truisms (https://github.com/azist/azos/blob/master/src/truisms.md)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Service_locator_pattern&oldid=971620696"