

Dependency injection

From Wikipedia, the free encyclopedia

Dependency injection is a software design pattern that allows the removal of hard-coded dependencies and makes it possible to change them, whether at run-time or compile-time.^[1]

This can be used, for example, as a simple way to load plugins dynamically or to choose stubs or mock objects in test environments vs. real objects in production environments. This software design pattern injects the depended-on element (object or value, etc.) to the destination automatically by knowing the requirement of the destination. Another pattern, called dependency lookup, is a regular process and reverse process to dependency injection.

One of its core principles is the separation of behavior from dependency resolution.

Contents

- 1 Definition
- 2 Motivation
- 3 Basics
- 4 Examples
 - 4.1 Highly coupled dependency
 - 4.2 Manually injected dependency
 - 4.3 Automatically injected dependency
 - 4.4 Unit testing using injected stub implementations
- 5 Benefits
- 6 Disadvantages
- 7 Types
- 8 See also
- 9 References
- 10 External links

Definition

Dependency injection involves at least three elements:

- a **dependent** consumer,
- a declaration of a component's **dependencies**, defined as interface contracts,
- an **injector** (sometimes referred to as a **provider** or **container**) that creates instances of classes that implement a given dependency interface on request.

The dependent object describes what software component it depends on to do its work. The injector decides what concrete classes satisfy the requirements of the dependent object, and provides them to the dependent.

In conventional software development, the dependent object decides for itself what concrete classes it will use. In the dependency injection pattern, this decision is delegated to the "injector" which can choose to substitute different concrete class implementations of a dependency contract interface at run-time rather than at compile time.

Being able to make this decision at run-time rather than compile time is the key advantage of dependency injection. Multiple, different implementations of a single software component can be created at run-time and passed (injected) into the same test code. The test code can then test each different software component without being aware that what has been injected is implemented differently.

Motivation

The primary purpose of the dependency injection pattern is to allow selection among multiple implementations of a given dependency interface at run time, or via configuration files, instead of at compile time. The pattern is particularly useful for providing stub test implementations of complex components when testing, but is often used for locating plugin components, or locating and initializing software services.

Unit testing of components in large software systems is difficult, because components under test often require the presence of a substantial amount of infrastructure and setup in order to operate at all. Dependency injection simplifies the process of bringing up a working instance of an isolated component for testing. Because components declare their dependencies, a test can automatically bring up only those dependencies required to perform testing.

More importantly, injectors can be configured to swap in simplified stub implementations of sub-components when testing, the idea being that the component under test can be tested in isolation as long as the substituted sub-components implement the contract of the dependent interface sufficiently to perform the unit test in question.

As an example, consider an automatic stock trading program that communicates with a live online trading service and stores historical analytic data in a distributed database. To test the component which recommends trades, one would ordinarily need to have a connection to the online service and an actual distributed database suitably populated with test data. Using dependency injection, the components that provide access to the online service and back-end databases could be replaced altogether with a test implementation of the dependency interface contracts that provide just enough behavior to perform tests on the component under tests.

Basics

Without dependency injection, a consumer component that needs a particular service in order to accomplish a task must create an instance of a class that concretely implements the dependency interface.

When using dependency injection, a consumer component specifies the service contract by interface, and the injector component selects an implementation on behalf of the dependent component.

In its simplest implementation, code that creates a dependent object supplies dependencies to that object via constructor arguments or by setting properties on the object.

More complicated implementations, such as Spring, Google Guice, and Microsoft Managed Extensibility Framework (MEF), automate this procedure. These frameworks identify constructor arguments or properties on the objects being created as requests for dependent objects, and automatically inject constructor arguments or set

properties with pre-constructed instances of dependencies as part of the process of creating the dependent object. The client makes a request to the dependency injection system for an implementation of a particular interface; the dependency injection system creates the object, automatically filling in dependencies as required.

Examples

Using the stock trading example mentioned above, the following Java examples show how coupled (manually injected) dependencies and framework-injected dependencies are typically staged.

The following interface contracts define the behavior of components in the sample system.

```
public interface IOnlineBrokerageService {
    String[] getStockSymbols();
    double getBidPrice(String stockSymbol);
    double getAskPrice(String stockSymbol);
    void putBuyOrder(String stockSymbol, int shares, double buyPrice);
    void putSellOrder(String stockSymbol, int shares, double sellPrice);
}

public interface IStockAnalysisService {
    double getEstimatedValue(String stockSymbol);
}

public interface IAutomatedStockTrader {
    void executeTrades();
}
```

Highly coupled dependency

The following example shows code with no dependency injection applied:

```
public class VerySimpleStockTraderImpl implements IAutomatedStockTrader {

    private IStockAnalysisService analysisService = new StockAnalysisServiceImpl();
    private IOnlineBrokerageService brokerageService = new NewYorkStockExchangeBrokerageServiceImpl();

    public void executeTrades() {
        ...
    }
}

public class MyApplication {
    public static void main(String[] args) {
        IAutomatedStockTrader stockTrader = new VerySimpleStockTraderImpl();
        stockTrader.executeTrades();
    }
}
```

The `VerySimpleStockTraderImpl` class creates instances of the `IStockAnalysisService`, and `IOnlineBrokerageService` by hard-coding constructor references to the concrete classes that implement those services.

Manually injected dependency

Refactoring the above example to use manual injection:

```
public class VerySimpleStockTraderImpl implements IAutomatedStockTrader {

    private IStockAnalysisService analysisService;
    private IOnlineBrokerageService brokerageService;

    public VerySimpleStockTraderImpl(
        IStockAnalysisService analysisService,
        IOnlineBrokerageService brokerageService) {
        this.analysisService = analysisService;
        this.brokerageService = brokerageService;
    }
    public void executeTrades() {
        ...
    }
}

public class MyApplication {
    public static void main(String[] args) {
        IStockAnalysisService analysisService = new StockAnalysisServiceImpl();
        IOnlineBrokerageService brokerageService = new NewYorkStockExchangeBrokerageServiceImpl();

        IAutomatedStockTrader stockTrader = new VerySimpleStockTraderImpl(
            analysisService,
            brokerageService);
        stockTrader.executeTrades();
    }
}
```

In this example, `MyApplication.main()` plays the role of dependency injector, selecting the concrete implementations of the dependencies required by `VerySimpleStockTraderImpl`, and supplying those dependencies via **constructor injection**.

Automatically injected dependency

There are several frameworks available that automate dependency management through delegation. Typically, this is done with a container using XML or metadata definitions. Refactoring the above example to use an external XML-definition framework:

```
<contract id="IAutomatedStockTrader">
    <implementation>VerySimpleStockTraderImpl</implementation>
</contract>
<contract id="IStockAnalysisService" singleton="true">
    <implementation>StockAnalysisServiceImpl</implementation>
</contract>
<contract id="IOnlineBrokerageService" singleton="true">
    <implementation>NewYorkStockExchangeBrokerageServiceImpl</implementation>
</contract>
```

```
public class VerySimpleStockTraderImpl implements IAutomatedStockTrader {
    private IStockAnalysisService analysisService;
```

```

private IOnlineBrokerageService brokerageService;

public VerySimpleStockTraderImpl(
    IStockAnalysisService analysisService,
    IOnlineBrokerageService brokerageService) {
    this.analysisService = analysisService;
    this.brokerageService = brokerageService;
}

public void executeTrades() {
    ...
}
}

public class MyApplication {
    public static void main(String[] args) {
        IAutomatedStockTrader stockTrader =
            (IAutomatedStockTrader) DependencyManager.create(IAutomatedStockTrader.class);
        stockTrader.executeTrades();
    }
}

```

In this case, a dependency injection service is used to retrieve an instance of a class that implements the `IAutomatedStockTrader` contract. From the configuration file the `DependencyManager` determines that it must create an instance of the `VerySimpleStockTraderImpl` class. By examining the constructor arguments via reflection, the `DependencyManager` further determines that the `VerySimpleStockTraderImpl` class has two dependencies; so it creates instances of the `IStockAnalysisService` and `IOnlineBrokerageService`, and supplies those dependencies as constructor arguments.

As there are many ways to implement dependency injection, only a small subset of examples are shown here. Dependencies can be registered, bound, located, externally injected, etc., by many different means. Hence, moving dependency management from one module to another can be accomplished in many ways.

Unit testing using injected stub implementations

Testing a stock trading application against a live brokerage service might have disastrous consequences. Dependency injection can be used to substitute test implementations in order to simplify unit testing. In the example given below, the unit test registers replacement implementations of the `IOnlineBrokerageService` and `IStockAnalysisService` in order to perform tests, and validate the behavior of `VerySimpleStockTraderImpl`.

```

public class VerySimpleStockBrokerTest {
    // Simplified stub implementation of IOnlineBrokerageService.
    public class StubBrokerageService implements IOnlineBrokerageService {
        public String[] getStockSymbols() {
            return new String[] { "ACME" };
        }
        public double getBidPrice(String stockSymbol) {
            return 100.0; // (just enough to complete the test)
        }
        public double getAskPrice(String stockSymbol) {
            return 100.25;
        }
        public void putBuyOrder(String stockSymbol, int shares, double buyPrice) {
            Assert.Fail("Should not buy ACME stock!");
        }
        public void putSellOrder(String stockSymbol, int shares, double sellPrice) {
            // not used in this test.
            throw new NotImplementedException();
        }
    }
}

```

```

    }
}

public class StubAnalysisService implements IStockAnalysisService {
    public double getEstimatedValue(String stockSymbol) {
        if (stockSymbol.equals("ACME"))
            return 1.0;
        return 100.0;
    }
}

public void TestVerySimpleStockTraderImpl() {
    // Direct the DependencyManager to use test implementations.
    DependencyManager.register(
        IOnlineBrokerageService.class,
        StubBrokerageService.class);
    DependencyManager.register(
        IStockAnalysisService.class,
        StubAnalysisService.class);

    IAutomatedStockTrader stockTrader =
        (IAutomatedStockTrader) DependencyManager.create(IAutomatedStockTrader.class);
    stockTrader.executeTrades();
}
}

```

Benefits

One benefit of using the dependency injection approach is the reduction of boilerplate code in the application objects since all work to initialize or set up dependencies is handled by a provider component.^[2]

Another benefit is that it offers configuration flexibility because alternative implementations of a given service can be used without recompiling code. This is useful in unit testing, as it is easy to inject a fake implementation of a service into the object being tested by changing the configuration file, or overriding component registrations at run-time.

Furthermore, dependency injection facilitates the writing of testable code.

Disadvantages

- Dependencies registered in the container are effectively black boxes with regards to the rest of the system. This makes it harder to detect and recover from their errors, and may make the system as a whole less reliable.
- Dependency injection hides the class' dependencies, causing run-time errors instead of compile-time errors when dependencies are missing or incompletely implemented.
- The dependency injection container makes the code more difficult to maintain, because it becomes unclear when you would be introducing a breaking change.
- If the dependency is injected at runtime, the performance of the application is reduced.
- The benefits may not be accurately measured against the costs. i.e. an application with a highly constrained context will not realize any benefit if that is the single context, yet supporting additional unrealized contexts will have come at a cost

Types

Martin Fowler identifies three ways in which an object can get a reference to an external module, according to the pattern used to provide the dependency:^[3]

- *Type 1* or *interface injection*, in which the exported module provides an interface that its users must implement in order to get the dependencies at runtime.
- *Type 2* or *setter injection*, in which the dependent module exposes a setter method that the framework uses to inject the dependency.
- *Type 3* or *constructor injection*, in which the dependencies are provided through the class constructor.

It is possible for other frameworks to have other types of injection beyond those presented above.^[4]

See also

- Architecture description language
- Factory pattern
- Inversion of control
- Plug-in (computing)
- Strategy pattern

References

1. ^ Niko Schwarz, Mircea Lungu, Oscar Nierstrasz, "Seuss: Decoupling responsibilities from static methods for fine-grained configurability", Journal of Object Technology, Volume 11, no. 1 (April 2012), pp. 3:1-23, doi:10.5381/jot.2012.11.1.a3 (<http://dx.doi.org/10.5381/jot.2012.11.1.a3>).
2. ^ "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 330" (<http://jcp.org/en/jsr/detail?id=330>). Jcp.org. Retrieved 2013-12-11.
3. ^ Martin Fowler (2004-01-23). "Inversion of Control Containers and the Dependency Injection pattern" (<http://www.martinfowler.com/articles/injection.html#FormsOfDependencyInjection>). Martinfowler.com. Retrieved 2013-12-11.
4. ^ "Yan - Dependency Injection Types" (<http://yan.codehaus.org/Dependency+Injection+Types>). Yan.codehaus.org. Retrieved 2013-12-11.

External links

- A beginners guide to Dependency Injection (<http://www.theserverside.com/tt/articles/article.tss?I=IOCBeginners>)
- Dependency Injection & Testable Objects: Designing loosely coupled and testable objects (<http://www.ddj.com/185300375>) - Jeremy Weiskotten; Dr. Dobb's Journal, May 2006.
- Design Patterns: Dependency Injection -- MSDN Magazine, September 2005 (<http://msdn.microsoft.com/en-us/magazine/cc163739.aspx>)
- Martin Fowler's original article that introduced the term Dependency Injection (<http://martinfowler.com/articles/injection.html>)
- P of EAA: Plugin (<http://martinfowler.com/eaCatalog/plugin.html>)
- The Rich Engineering Heritage Behind Dependency Injection (<http://www.javalobby.org/articles/di-heritage/>) - Andrew McVeigh - A detailed history of dependency injection.
- What is Dependency Injection? (<http://tutorials.jenkov.com/dependency-injection/index.html>) - An

alternative explanation - Jakob Jenkov

- Writing More Testable Code with Dependency Injection -- Developer.com, October 2006 (<http://www.developer.com/net/net/article.php/3636501>)
- Managed Extensibility Framework Overview -- MSDN (<http://msdn.microsoft.com/en-us/library/dd460648.aspx>)
- Old fashioned description of the Dependency Mechanism by Hunt 1998 (<http://www.midmarsh.co.uk/planetjava/tutorials/language/WatchingtheObservables.PDF>)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Dependency_injection&oldid=585560467"

Categories: Software design patterns | Component-based software engineering | Software architecture

- This page was last modified on 11 December 2013 at 07:26.
 - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.
- Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.