Dependency injection

From Wikipedia, the free encyclopedia

In software engineering, **dependency injection** is a technique whereby one object supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state.[1] Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

This fundamental requirement means that using values (services) produced within the class from new or static methods is prohibited. The class should accept values passed in from outside.

The intent behind dependency injection is to decouple objects to the extent that no client code has to be changed simply because an object it depends on needs to be changed to a different one.

Dependency injection is one form of the broader technique of inversion of control. Rather than low level code calling up to high level code, high level code can receive lower level code that it can call down to. This inverts the typical control pattern seen in procedural programming.

As with other forms of inversion of control, dependency injection supports the dependency inversion principle. The client delegates the responsibility of providing its dependencies to external code (the injector). The client is not allowed to call the injector code.[2] It is the injecting code that constructs the services and calls the client to inject them. This means the client code does not need to know about the injecting code. The client does not need to know how to construct the services. The client does not need to know which actual services it is using. The client only needs to know about the intrinsic interfaces of the services because these define how the client may use the services. This separates the responsibilities of use and construction.

There are three common means for a client to accept a dependency injection: setter-, interface- and constructor-based injection. Setter and constructor injection differ mainly by when they can be used. Interface injection differs in that the dependency is given a chance to control its own injection. All require that separate construction code (the injector) take responsibility for introducing a client and its dependencies to each other.[3]

Contents

Overview

Dependency injection separates the creation of a client's dependencies from the client's behavior, which allows program designs to be loosely coupled[7] and to follow the dependency inversion and single responsibility principles.[4][8] It directly contrasts with the service locator pattern, which allows clients to know about the system they use to find dependencies.

An injection, the basic unit of dependency injection, is not a new or a custom mechanism. It works in the same way that "parameter passing" works.[9] Referring to "parameter passing" as an injection carries the added implication that it's being done to isolate the client from details.

An injection is also about what is in control of the passing (never the client) and is independent of how the passing is accomplished, whether by passing a reference or a value.

Dependency injection for five-year-olds

When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.

What you should be doing is stating a need, "I need something to drink with lunch," and then we will make sure you have something when you sit down to eat.

John Munsch, 28 October 2009.[4][5][6]

Dependency injection involves four roles:

- the **service** object(s) to be used
- the **client** object that is depending on the services it uses
- the **interfaces** that define how the client may use the services
- the **injector**, which is responsible for constructing the services and injecting them into the client

Any object that may be used can be considered a **service**. Any object that uses other objects can be considered a **client**. The names have nothing to do with what the objects are for and everything to do with the role the objects play in any one injection.

The **interfaces** are the types the client expects its dependencies to be. At issue is what they make accessible. They may truly be interface types implemented by the services but also may be abstract classes or even the concrete services themselves, though this last would violate DIP[10] and sacrifice the dynamic decoupling that enables testing. It's only required that the client does not know which they are and therefore never treats them as concrete, say by constructing or extending them.

The client should have no concrete knowledge of the specific implementation of its dependencies. It should only know the interface's name and API. As a result, the client won't need to change even if what is behind the interface changes. However, if the interface is refactored from being a class to an interface type (or vice versa) the client will need to be recompiled.[11] This is significant if the client and services are published separately. This unfortunate coupling is one that dependency injection cannot resolve.

The **injector** introduces the services into the client. Often, it also constructs the client. An injector may connect together a very complex object graph by treating an object like a client and later as a service for another client. The injector may actually be many objects working together but may not be the client. The injector may be referred to by other names such as: assembler, provider, container, factory, builder, spring, construction code, or main.

Dependency injection can be applied as a discipline, one that asks that all objects separate construction and behavior. Relying on a DI framework to perform construction can lead to forbidding the use of the **new** keyword, or, less strictly, only allowing direct construction of value objects.[12][13][14][15]

Taxonomy

Inversion of control (IoC) is more general than DI. Put simply, IoC means letting other code call you rather than insisting on doing the calling. An example of IoC without DI is the template method pattern. Here polymorphism is achieved through subclassing, that is, inheritance.[16]

Dependency injection implements IoC through composition so is often identical to that of the strategy pattern, but while the strategy pattern is intended for dependencies to be interchangeable throughout an object's lifetime, in dependency injection it may be that only a single instance of a dependency is used.[17] This still achieves polymorphism, but through delegation and composition.

Dependency injection frameworks

Application frameworks such as CDI (http://cdi-spec.org/) and its implementation Weld (http://weld.cdi-spec.org), Spring, Guice, Play framework, Salta (http://ruediste.github.io/salta/), Glassfish HK2, Dagger (https://google.github.io/dagger/), and Managed Extensibility Framework (MEF) support dependency injection but are not required to do dependency injection.[18][19]

Advantages

- Dependency injection allows a client the flexibility of being configurable. Only the client's behavior is fixed. The client may act on anything that supports the intrinsic interface the client expects.
- Dependency injection can be used to externalize a system's configuration details into configuration files, allowing the system to be reconfigured without recompilation. Separate configurations can be written for different situations that require different implementations of components. This includes, but is not limited to, testing.
- Because dependency injection doesn't require any change in code behavior it can be applied to legacy code as a refactoring. The result is clients that are more independent and that are easier to unit test in isolation using stubs or mock objects that simulate other objects not under test. This ease of testing is often the first benefit noticed when using dependency injection.
- Dependency injection allows a client to remove all knowledge of a concrete implementation that it needs to use. This helps isolate the client from the impact of design changes and defects. It promotes reusability, testability and maintainability.[20]
- Reduction of boilerplate code in the application objects, since all work to initialize or set up dependencies is handled by a provider component.[20]
- Dependency injection allows concurrent or independent development. Two developers can independently develop classes that use each other, while only needing to know the interface the classes will communicate through. Plugins are often developed by third party shops that never even talk to the developers who created the product that uses the plugins.
- Dependency Injection decreases coupling between a class and its dependency.[21][22]

Disadvantages

- Dependency injection creates clients that demand configuration details be supplied by construction code. This can be onerous when obvious defaults are available.
- Dependency injection can make code difficult to trace (read) because it separates behavior from construction. This means developers must refer to more files to follow how a system performs.
- Dependency injection typically requires more upfront development effort since one can not summon into being something right when and where it is needed but must ask that it be injected and then ensure that it has been injected.

- Dependency injection forces complexity to move out of classes and into the linkages between classes which might not always be desirable or easily managed.[23]
- Ironically, dependency injection can encourage dependence on a dependency injection framework.[23][24][25]

Examples

Without dependency injection

In the following Java example, the Client class contains a Service member variable that is initialized by the Client constructor. The client controls which implementation of service is used and controls its construction. In this situation, the client is said to have a hard-coded dependency on ServiceExample.

```
// An example without dependency injection
public class Client {
    // Internal reference to the service used by this client
    private ServiceExample service;

    // Constructor
    Client() {
        // Specify a specific implementation in the constructor instead of using dependency injection
        service = new ServiceExample();
    }

    // Method within this client that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

Dependency injection is an alternative technique to initialize the member variable rather than explicitly creating a service object as shown above.

Three types of dependency injection

There are at least three ways an object can receive a reference to an external module:[26]

- *constructor injection*: the dependencies are provided through a class constructor.
- *setter injection*: the client exposes a setter method that the injector uses to inject the dependency.
- *interface injection*: the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency.

Other types

It is possible for DI frameworks to have other types of *injection* beyond those presented above.[27]

Testing frameworks may also use other types. Some modern testing frameworks do not even require that clients actively accept dependency injection thus making legacy code testable. In particular, in the Java language it is possible to use reflection to make private attributes public when testing and thus accept injections by assignment.[28]

Some attempts at Inversion of Control do not provide full removal of dependency but instead simply substitute one form of dependency for another. As a rule of thumb, if a programmer can look at nothing but the client code and tell what framework is being used, then the client has a hard-coded dependency on the framework.

Constructor injection

This method requires the client to provide a parameter in a constructor for the dependency.

```
// Constructor
Client(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

Setter injection

This method requires the client to provide a setter method for the dependency.

```
// Setter method
public void setService(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

Interface injection

This is simply the client publishing a role interface to the setter methods of the client's dependencies. It can be used to establish how the injector should talk to the client when injecting dependencies.

```
// Service setter interface.
public interface ServiceSetter {
    public void setService(Service service);
}

// Client class
public class Client implements ServiceSetter {
    // Internal reference to the service used by this client.
    private Service service;

    // Set the service that this client is to use.
    @Override
    public void setService(Service service) {
        this.service = service;
    }
}
```

Constructor injection comparison

Preferred when all dependencies can be constructed first because it can be used to ensure the client object is always in a valid state, as opposed to having some of its dependency references be null (not be set). However, on its own, it lacks the flexibility to have its dependencies changed later. This can be a first step towards making the client immutable and therefore thread safe.

```
// Constructor
Client(Service service, Service otherService) {
    if (service == null) {
        throw new InvalidParameterException("service must not be null");
    }
    if (otherService == null) {
        throw new InvalidParameterException("otherService must not be null");
    }

    // Save the service references inside this client
    this.service = service;
    this.otherService = otherService;
}
```

Setter injection comparison

Requires the client to provide a setter method for each dependency. This gives the freedom to manipulate the state of the dependency references at any time. This offers flexibility, but if there is more than one dependency to be injected, it is difficult for the client to ensure that all dependencies are injected before the client could be provided for use.

```
// Set the service to be used by this client
public void setService(Service service) {
    if (service == null) {
        throw new InvalidParameterException("service must not be null");
    }
    this.service = service;
}

// Set the other service to be used by this client
public void setOtherService(Service otherService) {
    if (otherService == null) {
        throw new InvalidParameterException("otherService must not be null");
    }
    this.otherService = otherService;
}
```

Because these injections happen independently there is no way to tell when the injector is finished wiring the client. A dependency can be left null simply by the injector failing to call its setter. This forces the check that injection was completed from when the client is assembled to whenever it is used.

```
// Set the service to be used by this client
public void setService(Service service) {
    this.service = service;
}

// Set the other service to be used by this client
public void setOtherService(Service otherService) {
    this.otherService = otherService;
}

// Check the service references of this client
private void validateState() {
    if (service == null) {
        throw new IllegalStateException("service must not be null");
    }
    if (otherService == null) {
        throw new IllegalStateException("otherService must not be null");
    }
}

// Method that uses the service references
public void doSomething() {
    validateState();
    service.doYourThing();
    otherService.doYourThing();
}
```

Interface injection comparison

The advantage of interface injection is that dependencies can be completely ignorant of their clients yet can still receive a reference to a new client and, using it, send a reference-to-self back to the client. In this way, the dependencies become injectors. The key is that the injecting method (which could just be a classic setter method) is provided through an interface.

An assembler is still needed to introduce the client and its dependencies. The assembler would take a reference to the client, cast it to the setter interface that sets that dependency, and pass it to that dependency object which would turn around and pass a reference-to-self back to the client.

For interface injection to have value, the dependency must do something in addition to simply passing back a reference to itself. This could be acting as a factory or sub-assembler to resolve other dependencies, thus abstracting some details from the main assembler. It could be reference-counting so that the dependency knows how many clients are using it. If the dependency maintains a collection of clients, it could later inject them all with a different instance of itself.

Assembling examples

Manually assembling in main by hand is one way of implementing dependency injection.

```java
public class Injector {
    public static void main(String[] args) {
        // Build the dependencies first
        Service service = new ServiceExample();

        // Inject the service, constructor style
        Client client = new Client(service);

        // Use the objects
        System.out.println(client.greet());
    }
}
```

The example above constructs the object graph manually and then invokes it at one point to start it working. Important to note is that this injector is not pure. It uses one of the objects it constructs. It has a purely construction-only relationship with ServiceExample but mixes construction and using of Client. This should not be common. It is, however, unavoidable. Just like object oriented software needs a non-object oriented static method like main() to get started, a dependency injected object graph needs at least one (preferably only one) entry point to get the using started.

Manual construction in the main method may not be this straight forward and may involve calling builders, factories, or other construction patterns as well. This can be fairly advanced and abstract. The line is crossed from manual dependency injection to framework dependency injection once the constructing code is no longer custom to the application and is instead universal.[29]

Frameworks like Spring can construct these same objects and wire them together before returning a reference to client. Notice that all mention of ServiceExample has been removed from the code.

```java
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Injector {
    public static void main(String[] args) {
        // -- Assembling objects -- //
        BeanFactory beanfactory = new ClassPathXmlApplicationContext("Beans.xml");
        Client client = (Client) beanfactory.getBean("client");

        // -- Using objects -- //
        System.out.println(client.greet());
    }
}
```

Frameworks like Spring allow assembly details to be externalized in configuration files. This code (above) constructs objects and wires them together according to Beans.xml (below). ServiceExample is still constructed even though it's only mentioned below. A long and complex object graph can be defined this way and the only class mentioned in code would be the one with the entry point method, which in this case is greet().

```xml
 <?xml version="1.0" encoding="UTF-8"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="service" class="ServiceExample">
    </bean>
    <bean id="client" class="Client">
        <constructor-arg value="service" />
    </bean>
</beans>
```

In the example above Client and Service have not had to undergo any changes to be provided by spring. They are allowed to remain simple POJOs.[30][31][32] This shows how spring can connect services and clients that are completely ignorant of its existence. This could not be said if spring annotations are added to the classes. By keeping spring specific annotations and calls from spreading out among many classes, the system stays only loosely dependent on spring.[24] This can be important if the system intends to outlive spring.

The choice to keep POJOs pure doesn't come without cost. Rather than spending the effort to develop and maintain complex configuration files it is possible to simply use annotations to mark classes and let spring do the rest of the work. Resolving dependencies can be simple if they follow a convention such as matching by type or by name. This is choosing convention over configuration.[33] It is also arguable that, when refactoring to another framework, removing framework specific annotations would be a trivial part of the task[34] and many injection annotations are now standardized.[35][36]

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Injector {
    public static void main(String[] args) {
        // Assemble the objects
        BeanFactory beanfactory = new AnnotationConfigApplicationContext(MyConfiguration.class);
        Client client = beanfactory.getBean(Client.class);

        // Use the objects
        System.out.println(client.greet());
    }
}
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@ComponentScan
static class MyConfiguration {
    @Bean
    public Client client(ServiceExample service) {
        return new Client(service);
    }
}
```

```
@Component
public class ServiceExample {
    public String getName() {
        return "World!";
    }
}
```

Assembly comparison

The different injector implementations (factories, service locators, and dependency injection containers) are not that different as far as dependency injection is concerned. What makes all the difference is where they are allowed to be used. Move calls to a factory or a service locator out of the client and into main and suddenly main makes a fairly good dependency injection container.

By moving all knowledge of the injector out, a clean client, free of knowledge of the outside world, is left behind. However, any object that uses other objects can be considered a client. The object that contains main is no exception. This main object is not using dependency injection. It's actually using the service locator pattern. This can't be avoided because the choice of service implementations must be made somewhere.

Externalizing the dependencies into configuration files doesn't change this fact. What makes this reality part of a good design is that the service locator is not spread throughout the code base. It's confined to one place per application. This leaves the rest of the code base free to use dependency injection to make clean clients.

AngularJS example

In the AngularJS framework, there are only three ways a component (object or function) can directly access its dependencies:

1. The component can create the dependency, typically using the `new` operator.
2. The component can look up the dependency, by referring to a global variable.
3. The component can have the dependency passed to it where it is needed.

The first two options of creating or looking up dependencies are not optimal because they hard code the dependency to the component. This makes it difficult, if not impossible, to modify the dependencies. This is especially problematic in tests, where it is often desirable to provide mock dependencies for test isolation.

The third option is the most viable, since it removes the responsibility of locating the dependency from the component. The dependency is simply handed to the component.

```
function SomeClass(greeter) {
  this.greeter = greeter;
}

SomeClass.prototype.doSomething = function(name) {
  this.greeter.greet(name);
}
```

In the above example `SomeClass` is not concerned with creating or locating the greeter dependency, it is simply handed the greeter when it is instantiated.

This is desirable, but it puts the responsibility of getting hold of the dependency on the code that constructs `SomeClass`.

To manage the responsibility of dependency creation, each AngularJS application has an injector. The injector is a service locator that is responsible for construction and look-up of dependencies.

Here is an example of using the injector service:

```
// Provide the wiring information in a module
var myModule = angular.module('myModule', []);

// Teach the injector how to build a greeter service.
// Notice that greeter is dependent on the $window service.
// The greeter service is an object that
// contains a greet method.
myModule.factory('greeter', function($window) {
  return {
    greet: function(text) {
      $window.alert(text);
    }
  };
});
```

Create a new injector that can provide components defined in the `myModule` module and request our greeter service from the injector. (This is usually done automatically by the AngularJS bootstrap).

```
var injector = angular.injector(['myModule', 'ng']);
var greeter = injector.get('greeter');
```

Asking for dependencies solves the issue of hard coding, but it also means that the injector needs to be passed throughout the application. Passing the injector breaks the Law of Demeter. To remedy this, we use a declarative notation in our HTML templates, to hand the responsibility of creating components over to the injector, as in this example:

```
<div ng-controller="MyController">
  <button ng-click="sayHello()">Hello</button>
</div>
```

```
function MyController($scope, greeter) {
  $scope.sayHello = function() {
    greeter.greet('Hello World');
  };
}
```

When AngularJS compiles the HTML, it processes the `ng-controller` directive, which in turn asks the injector to create an instance of the controller and its dependencies.

```
injector.instantiate(MyController);
```

This is all done behind the scenes. Notice that by having the `ng-controller` ask the injector to instantiate the class, it can satisfy all of the dependencies of `MyController` without the controller ever knowing about the injector. This is the best outcome. The application code simply declares the dependencies it needs, without having to deal with the injector. This setup does not break the Law of Demeter.

See also

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

References

1. I.T., Titanium. "James Shore: Dependency Injection Demystified" (http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html). *www.jamesshore.com*. Retrieved 2015-07-18.
2. "HollywoodPrinciple" (http://c2.com/cgi/wiki?HollywoodPrinciple). *http://c2.com*. Retrieved 2015-07-19. External link in `|website=` (help)
3. "Inversion of Control Containers and the Dependency Injection pattern" (http://martinfowler.com/articles/injection.html). Retrieved 2015-07-18.
4. Seeman, Mark (October 2011). *Dependency Injection in .NET*. Manning Publications. p. 4. ISBN 9781935182504.
5. "Dependency Injection in NET" (http://philkildea.co.uk/james/books/Dependency.Injection.in.NET.pdf) (PDF). *http://philkildea.co.uk*. p. 4. Retrieved 2015-07-18. External link in `|website=` (help)
6. "How to explain dependency injection to a 5-year-old?" (https://stackoverflow.com/questions/1638919/how-to-explain-dependency-injection-to-a-5-year-old). *stackoverflow.com*. Retrieved 2015-07-18.

7. Seemann, Mark. "Dependency Injection is Loose Coupling" (http://blog.ploeh.dk/2010/04/07/DependencyInjectionisLooseCoupling/). *blog.ploeh.dk*. Retrieved 2015-07-28.
8. 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
9. "Passing Information to a Method or a Constructor (The Java™ Tutorials > Learning the Java Language > Classes and Objects)" (https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html). *docs.oracle.com*. Retrieved 2015-07-18.
10. "A curry of Dependency Inversion Principle (DIP), Inversion of Control (IoC), Dependency Injection (DI) and IoC Container - CodeProject" (http://www.codeproject.com/Articles/538536/A-curry-of-Dependency-Inversion-Principle-DIP-Inve). *www.codeproject.com*. Retrieved 2015-08-08.
11. "How to force "program to an interface" without using a java Interface in java 1.6" (http://programmers.stackexchange.com/questions/257976/how-to-force-program-to-an-interface-without-using-a-java-interface-in-java-1). *programmers.stackexchange.com*. Retrieved 2015-07-19.
12. "To "new" or not to "new"…" (http://misko.hevery.com/2008/09/30/to-new-or-not-to-new/). Retrieved 2015-07-18.
13. "How to write testable code" (http://www.loosecouplings.com/2011/01/how-to-write-testable-code-overview.html). *www.loosecouplings.com*. Retrieved 2015-07-18.
14. "Writing Clean, Testable Code" (http://www.ethanresnick.com/blog/testableCode.html). *www.ethanresnick.com*. Retrieved 2015-07-18.
15. Sironi, Giorgio. "When to inject: the distinction between newables and injectables - Invisible to the eye" (http://www.giorgiosironi.com/2009/07/when-to-inject-distinction-between.html). *www.giorgiosironi.com*. Retrieved 2015-07-18.
16. "Inversion of Control vs Dependency Injection" (https://stackoverflow.com/questions/6550700/inversion-of-control-vs-dependency-injection). *stackoverflow.com*. Retrieved 2015-08-05.
17. "What is the difference between Strategy pattern and Dependency Injection?" (https://stackoverflow.com/questions/4176520/what-is-the-difference-between-strategy-pattern-and-dependency-injection). *stackoverflow.com*. Retrieved 2015-07-18.
18. "Dependency Injection != using a DI container" (http://www.loosecouplings.com/2011/01/dependency-injection-using-di-container.html). *www.loosecouplings.com*. Retrieved 2015-07-18.
19. "Black Sheep » DIY-DI » Print" (http://blacksheep.parry.org/archives/diy-di/print/). *blacksheep.parry.org*. Retrieved 2015-07-18.
20. "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 330" (https://jcp.org/en/jsr/detail?id=330). *jcp.org*. Retrieved 2015-07-18.
21. "the urban canuk, eh: On Dependency Injection and Violating Encapsulation Concerns" (http://www.bryancook.net/2011/08/on-dependency-injection-and-violating.html). *www.bryancook.net*. Retrieved 2015-07-18.
22. "The Dependency Injection Design Pattern" (https://msdn.microsoft.com/en-us/library/vstudio/hh323705(v=vs.100).aspx). *msdn.microsoft.com*. Retrieved 2015-07-18.
23. "What are the downsides to using Dependency Injection?" (https://stackoverflow.com/questions/2407540/what-are-the-downsides-to-using-dependency-injection). *stackoverflow.com*. Retrieved 2015-07-18.
24. "Dependency Injection Inversion - Clean Coder" (https://sites.google.com/site/unclebobconsultingllc/blogs-by-robert-martin/dependency-injection-inversion). *sites.google.com*. Retrieved 2015-07-18.
25. "Decoupling Your Application From Your Dependency Injection Framework" (http://www.infoq.com/news/2010/01/dependency-injection-inversion). *InfoQ*. Retrieved 2015-07-18.
26. Martin Fowler (2004-01-23). "Inversion of Control Containers and the Dependency Injection pattern - Forms of Dependency Injection" (http://www.martinfowler.com/articles/injection.html#FormsOfDependencyInjection). Martinfowler.com. Retrieved 2014-03-22.
27. "Yan - Dependency Injection Types" (https://web.archive.org/web/20130818125650/http://yan.codehaus.org:80/Dependency%20Injection%20Types). Yan.codehaus.org. Archived from the original (http://yan.codehaus.org/Dependency+Injection+Types) on 2013-08-18. Retrieved 2013-12-11.
28. "AccessibleObject (Java Platform SE 7)" (http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/AccessibleObject.html). *docs.oracle.com*. Retrieved 2015-07-18.
29. Riehle, Dirk (2000), *Framework Design: A Role Modeling Approach* (http://www.riehle.org/computer-science/research/dissertation/diss-a4.pdf) (PDF), Swiss Federal Institute of Technology
30. "Spring Tips: A POJO with annotations is not Plain" (http://springtips.blogspot.com/2007/07/pojo-with-annotations-is-not-plain.html). Retrieved 2015-07-18.
31. "Annotations in POJO – a boon or a curse? | Techtracer" (http://techtracer.com/2007/04/07/annotations-in-pojo-a-boon-or-a-curse/). Retrieved 2015-07-18.
32. "Pro Spring Dynamic Modules for OSGi Service Platforms" (https://books.google.com/books?id=FCVnsq1ZUI0C&pg=PA64&lpg=PA64&dq=spring+pojo+annotation+free&source=bl&ots=YzORtPWR8g&sig=6evcrQuZzscyLIT6uggpyCKn5Sc&hl=en&sa=X&ei=4GCaVcb7EoWMNpC0mdgM&ved=0CGIQ6AEwCQ#v=onepage&q=spring%20pojo%20annotation%20free&f=false). APress. Retrieved 2015-07-06.

33. "Captain Debug's Blog: Is ‘Convention Over Configuration’ Going Too Far?" (http://www.captaindebug.com/2011/08/is-convention-over-configuration-going.html#.VapTMvlVhHx). *www.captaindebug.com*. Retrieved 2015-07-18.
34. Decker, Colin. "What's the issue with @Inject? | Colin's Devlog" (http://blog.cgdecker.com/2010/01/whats-issue-with-inject.html). *blog.cgdecker.com*. Retrieved 2015-07-18.
35. Morling, Gunnar (2012-11-18). "Dagger - A new Java dependency injection framework" (http://musingsofaprogrammingaddict.blogspot.com/2012/11/dagger-new-java-dependency-injection.html). *Dagger - A new Java dependency injection framework - Musings of a Programming Addict*. Retrieved 2015-07-18.
36. "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 330" (https://www.jcp.org/en/jsr/detail?id=330). *www.jcp.org*. Retrieved 2015-07-18.

External links

Wikimedia Commons has media related to ***Dependency injection***.

- A beginners guide to Dependency Injection (http://www.theserverside.com/tt/articles/article.tss?l=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/eaaCatalog/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 (https://web.archive.org/web/20120425150101/http://www.midmarsh.co.uk/planetjava/tutorials/language/WatchingtheObservables.PDF)
- Refactor Your Way to a Dependency Injection Container (http://blog.thecodewhisperer.com/2011/12/07/refactor-your-way-to-a-dependency-injection-container/)
- Understanding DI in PHP (http://php-di.org/doc/understanding-di.html)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Dependency_injection&oldid=783274223"

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

Inversion of control

From Wikipedia, the free encyclopedia

In software engineering, **inversion of control** (**IoC**) is a design principle in which custom-written portions of a computer program receive the flow of control from a generic framework. A software architecture with this design inverts control as compared to traditional procedural programming: in traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks, but with inversion of control, it is the framework that calls into the custom, or task-specific, code.

Inversion of control is used to increase modularity of the program and make it extensible,[1] and has applications in object-oriented programming and other programming paradigms. The term was popularized by Robert C. Martin and Martin Fowler.

The term is related to, but different from, the dependency inversion principle, which concerns itself with decoupling dependencies between high-level and low-level layers through shared abstractions. The general concept is also related to event-driven programming in that it is often implemented using IoC, so that the custom code is commonly only concerned with handling of events, whereas the event loop and dispatch of events/messages is handled by the framework or the runtime environment.

Contents

Overview

As an example, with traditional programming, the main function of an application might make function calls into a menu library to display a list of available commands and query the user to select one.[2] The library thus would return the chosen option as the value of the function call, and the main function uses this value to execute the associated command. This style was common in text based interfaces. For example, an email client may show a screen with commands to load new mails, answer the current mail, start a new mail, etc., and the program execution would block until the user presses a key to select a command.

With inversion of control, on the other hand, the program would be written using a software framework that knows common behavioral and graphical elements, such as windowing systems, menus, controlling the mouse, and so on. The custom code "fills in the blanks" for the framework, such as supplying a table of menu items and registering a code subroutine for each item, but it is the framework that monitors the user's actions and invokes the subroutine when a menu item is selected. In the mail client example, the framework could follow both the keyboard and mouse inputs and call the command invoked by the user by either means, and at the same time monitor the network interface to find out if new messages arrive and refresh the screen when some network activity is detected. The same framework could be used as the skeleton for a spreadsheet program or a text editor. Conversely, the framework knows nothing about Web browsers, spreadsheets or text editors; implementing their functionality takes custom code.

Inversion of control carries the strong connotation that the reusable code and the problem-specific code are developed independently even though they operate together in an application. Software frameworks, callbacks, schedulers, event loops and dependency injection are examples of design patterns that follow the inversion of control principle, although the term is most commonly used in the context of object-oriented programming.

Inversion of control serves the following design purposes:

- To decouple the execution of a task from implementation.
- To focus a module on the task it is designed for.
- To free modules from assumptions about how other systems do what they do and instead rely on contracts.
- To prevent side effects when replacing a module.

Inversion of control is sometimes facetiously referred to as the "Hollywood Principle: Don't call us, we'll call you".

Background

Inversion of control is not a new term in computer science. Martin Fowler traces the etymology of the phrase back to 1988.[3] Dependency injection is a specific type of IoC using contextualized lookup.[2] A service locator such as the Java Naming and Directory Interface (JNDI) is similar. In an article by Loek Bergman,[4] it is presented as an architectural principle.

In an article by Robert C. Martin,[5] the dependency inversion principle and abstraction by layering come together. His reason to use the term "inversion" is in comparison with traditional software development methods. He describes the uncoupling of services by the abstraction of layers when he is talking about dependency inversion. The principle is used to find out where system borders are in the design of the abstraction layers.

Description

In traditional programming, the flow of the business logic is determined by objects that are statically bound to one another. With inversion of control, the flow depends on the object graph that is built up during program execution. Such a dynamic flow is made possible by object interactions that are defined through abstractions. This run-time binding is achieved by mechanisms such as dependency injection or a service locator. In IoC, the code could also be linked statically during compilation, but finding the code to execute by reading its description from external configuration instead of with a direct reference in the code itself.

In dependency injection, a dependent object or module is coupled to the object it needs at run time. Which particular object will satisfy the dependency during program execution typically cannot be known at compile time using static analysis. While described in terms of object interaction here, the principle can apply to other programming methodologies besides object-oriented programming.

In order for the running program to bind objects to one another, the objects must possess compatible interfaces. For example, class `A` may delegate behavior to interface `I` which is implemented by class `B`; the program instantiates `A` and `B`, and then injects `B` into `A`.

Implementation techniques

In object-oriented programming, there are several basic techniques to implement inversion of control. These are:

- Using a factory pattern
- Using a service locator pattern

- Using dependency injection, for example
  - Constructor injection
  - Parameter injection
  - Setter injection
  - Interface injection
- Using a contextualized lookup
- Using template method design pattern
- Using strategy design pattern

In an original article by Martin Fowler,[6] the first three different techniques are discussed. In a description about inversion of control types,[7] the last one is mentioned. Often the contextualized lookup will be accomplished using a service locator.

More important than the applied technique, however, is the optimization of the purposes.

Examples

Most frameworks such as .NET or Enterprise JavaBeans display this pattern:

```
public class ServerFacade {
   public <K, V> V respondToRequest(K request) {
      if (businessLayer.validateRequest(request)) {
         DAO.getData(request);
         return Aspect.convertData(request);
      }
      return null;
   }
}
```

This basic outline in Java gives an example of code following the IoC methodology. It is important, however, that in the `ServerFacade` a lot of assumptions are made about the data returned by the data access object (DAO).

Although all these assumptions might be valid at some time, they couple the implementation of the `ServerFacade` to the DAO implementation. Designing the application in the manner of inversion of control would hand over the control completely to the DAO object. The code would then become

```
public class ServerFacade {
   public <K, V> V respondToRequest(K request, DAO dao) {
      return dao.getData(request);
   }
}
```

The example shows that the way the method `respondToRequest` is constructed determines if IoC is used. It is the way that parameters are used that define IoC. This resembles the message-passing style that some object-oriented programming languages use.

See also

- Abstraction layer
- Asynchronous I/O
- Callback (computer science)
- Closure (computer science)
- Continuation
- Delegate (CLI)
- Dependency inversion principle
- Flow-based programming
- Implicit invocation

- Interrupt handler
- Message Passing
- Monad (functional programming)
- Observer pattern
- Publish/subscribe
- Service locator pattern
- Signal (computing)
- Software framework
- Strategy pattern
- User exit
- Visitor pattern
- XSLT

References

1. Ralph E. Johnson & Brian Foote (June–July 1988). "Designing Reusable Classes" (http://www.laputan.org/drc/drc.html). *Journal of Object-Oriented Programming, Volume 1, Number 2*. Department of Computer Science University of Illinois at Urbana-Champaign. pp. 22–35. Retrieved 29 April 2014.
2. Dependency Injection (http://martinfowler.com/articles/injection.html).
3. Inversion of Control (http://martinfowler.com/bliki/InversionOfControl.html) on Martin Fowler's Bliki
4. Archived (https://web.archive.org/web/*/http://loekbergman.nl/InsideArchitecture/TheVision/InversionOfControl) index at the Wayback Machine. Inside Architecture: write once, run anywhere by Loek Bergman
5. The Dependency Inversion principle (https://web.archive.org/web/20041221102842/http://www.objectmentor.com/resources/articles/dip.pdf) by Robert C. Martin
6. Inversion of Control Containers and the Dependency Injection Pattern (http://www.martinfowler.com/articles/injection.html) by Martin Fowler
7. IoC Types (http://docs.codehaus.org/display/PICO/IoC+Types)

External links

- Inversion of Control explanation and implementation example (http://javaprogrammernotes.blogspot.com/2015/03/inversion-of-control-dependency.html)

Wikimedia Commons has media related to ***Inversion of control***.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Inversion_of_control&oldid=779472960"

Categories: Software design patterns | Software architecture | Architectural pattern (computer science) | Java (programming language) | Programming principles | Component-based software engineering