What is dependency injection?

Asked 16 years, 3 months ago Modified 1 month ago

Viewed 1.3m times

3669

There have been several questions already posted with specific questions about dependency injection, such as when to use it and what frameworks are there for it. However,

What is dependency injection and when/why should or shouldn't it be used?

design-patterns language-agnostic dependency-injection terminology

Share

Improve this question

Follow

edited Mar 19, 2014 at 20:42

Peter Mortensen

31.6k ● 22 ● 109 ● 133

asked Sep 25, 2008 at 0:28

AR.

40.5k ● 9 ● 46 ● 53

See my discussion on Dependency Injection Here. – Kevin S. Sep 25, 2008 at 8:27

46 I agree with the comments regarding links. I can understand you may want to reference someone else. But

at least add why you are linking them and what makes this link better than the other links I could get by using google – Christian Payne Jun 1, 2009 at 0:27

@AR: Technically, Dependency Injection is *not* a special form of IoC. Rather, IoC is one technique that is used to provide Dependency Injection. Other techniques could be used to provide Dependency Injection (although IoC is the only one in common use), and IoC is used for many other problems as well. – Sean Reilly Aug 28, 2009 at 12:50

158 Regarding links, remember that they often disappear one way or another. There is a growing number of dead links in SO answers. So, no matter how good the linked article is, it's no good at all if you can't find it. – DOK Oct 28, 2010 at 16:26

An overview of Dependency Injection and its relationship to other OOP principles: deviq.com/dependency-injection – ssmith Nov 17, 2015 at 14:14

40 Answers

Sorted by: Highest score (default)

1 2 Next

2850

The best definition I've found so far is one by James Shore:

> "Dependency Injection" is a 25-dollar term for a 5-cent concept. [...] Dependency injection means giving an object its instance variables. [...].

There is an article by Martin Fowler that may prove useful, too.

Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself. It's a very useful technique for testing, since it allows dependencies to be mocked or stubbed out.

Dependencies can be injected into objects by many means (such as constructor injection or setter injection). One can even use specialized dependency injection frameworks (e.g. Spring) to do that, but they certainly aren't required. You don't need those frameworks to have dependency injection. Instantiating and passing objects (dependencies) explicitly is just as good an injection as injection by framework.

Share Improve this answer

Follow

edited Jan 29, 2017 at 21:43
ardila
1,285 ● 1 ● 13 ● 24

answered Sep 26, 2008 at 16:50
Thiago Arrais
34.1k ● 7 ● 33 ● 35

---

66 I like the explanation of James' article, especially the end: " Still, you have to marvel at any approach that takes three concepts ('TripPlanner,' 'CabAgency,' and 'AirlineAgency'), turns them into nine-plus classes, and then adds dozens of lines of glue code and configuration XML before a single line of application logic is written." This is what I have seen very often (sadly) - that dependency injection (which is good per se as explained by him) is misused to overcomplicate things that could have been done easier - ending up writing "supporting" code ... – Matt Nov 26, 2015 at 9:25

7 Re: "Instantiating and passing objects (dependencies) explicitly is just as good an injection as injection by framework.". So why people made frameworks doing that? – dzieciou Dec 5, 2015 at 11:52

30 For the same reason that every framework gets (or at least should get) written: because there is a lot of repeated/boilerplate code that needs to be written once you reach a certain complexity. The problem is many times folks will reach for a framework even when it isn't strictly needed. – Thiago Arrais Jan 11, 2016 at 17:54

31 $25 term for a 5-cent concept is dead on. Here's a good article that's helped me: codeproject.com/Articles/615139/… – Christine Sep 30, 2016 at 22:25

2 @dzieciou also nice to have the object graph built out for you when you use a DI container and also nice to be able to swap out one implementation for another one in just one place. Generally for silly, simple stuff I might pass in the dependency but it is so easy to use the DI container in most frameworks often just as easy to do that too. – M H Oct 21, 2020 at 9:58

2176

Dependency Injection is passing dependency to other **objects** or **framework**(dependency injector).

Dependency injection makes testing easier. The injection can be done through **constructor**.

`SomeClass()` has its constructor as following:

```
public SomeClass() {
    myObject = Factory.getObject();
}
```

Problem: If `myObject` involves complex tasks such as disk access or network access, it is **hard** to do unit test on `SomeClass()`. Programmers have to mock `myObject` and might **intercept** the factory call.

Alternative solution:

- Passing `myObject` in as an argument to the constructor

```
public SomeClass (MyClass myObject) {
    this.myObject = myObject;
}
```

`myObject` can be passed directly which makes testing easier.

- One common alternative is defining a **do-nothing constructor**. Dependency injection can be done through setters. (h/t @MikeVella).
- Martin Fowler documents a third alternative (h/t @MarcDix), where **classes explicitly implement an interface** for the dependencies programmers wish injected.

It is harder to isolate components in unit testing without dependency injection.

In 2013, when I wrote this answer, this was a major theme on the Google Testing Blog. It remains the biggest advantage to me, as programmers not always need the extra flexibility in their run-time design (for instance, for service locator or similar patterns). Programmers often need to isolate the classes during testing.

Share Improve this answer Follow

edited Mar 22, 2019 at 6:34

Trieu Toan
9 ● 2

answered Sep 25, 2008 at 0:49

wds
32.2k ● 11 ● 62 ● 84

30 Acknowledging that Ben Hoffstein's referenceto Martin Fowler's article is necessary as pointing a 'must-read' on the subject, I'm accepting wds' answer because it actually answers the question here on SO. – AR. Sep 26, 2008 at 16:55

138 +1 for explanation and motivation: *making the creation of objects on which a class depends someone else's problem*. Another way to say it is that DI makes classes more cohesive (they have fewer responsibilities). – Fuhrmanator Nov 29, 2012 at 18:26

15 You say the dependency is passed "in to the constructor" but as I understand it this isn't strictly true. It's still dependency injection if the dependency is set as a property after the object has been instantiated, correct? – Mike Vella Aug 7, 2013 at 19:52

2 @MikeVella Yes, that is correct. It makes no real difference in most cases, though properties are generally a bit more flexible. I will edit the text slightly to point that out. – wds Aug 8, 2013 at 15:14

3 One of the best answers I've found so far, thus I'm really interested in improving it. It's missing a description of the

third form of dependency injection: Interface injection.
– Marc Dix May 27, 2016 at 21:27

841

I found this funny example in terms of loose coupling:

Source: *Understanding dependency injection*

Any application is composed of many objects that collaborate with each other to perform some useful stuff. Traditionally each object is responsible for obtaining its own references to the dependent objects (dependencies) it collaborate with. This leads to highly coupled classes and hard-to-test code.

For example, consider a `Car` object.

A `Car` depends on wheels, engine, fuel, battery, etc. to run. Traditionally we define the brand of such dependent objects along with the definition of the `Car` object.

Without Dependency Injection (DI):

```
class Car{
  private Wheel wh = new NepaliRubberWheel();
  private Battery bt = new ExcideBattery();

  //The rest
}
```

Here, the `Car` object *is responsible for creating the dependent objects.*

What if we want to change the type of its dependent object - say `Wheel` - after the initial `NepaliRubberWheel()` punctures? We need to recreate the Car object with its new dependency say `ChineseRubberWheel()`, but only the `Car` manufacturer can do that.

Then what does the `Dependency Injection` do for us...?

When using dependency injection, objects are given their dependencies *at run time rather than compile time (car manufacturing time)*. So that we can now change the `Wheel` whenever we want. Here, the `dependency` (`wheel`) can be injected into `Car` at run time.

After using dependency injection:

Here, we are **injecting** the **dependencies** (Wheel and Battery) at runtime. Hence the term : *Dependency Injection.* We normally rely on DI frameworks such as Spring, Guice, Weld to create the dependencies and inject where needed.

```
class Car{
  private Wheel wh; // Inject an Instance of Wheel (de
runtime
  private Battery bt; // Inject an Instance of Battery
runtime
  Car(Wheel wh,Battery bt) {
      this.wh = wh;
      this.bt = bt;
  }
  //Or we can have setters
  void setWheel(Wheel wh) {
      this.wh = wh;
```

```
    }
}
```

The advantages are:

- decoupling the creation of object (in other word, separate usage from the creation of object)
- ability to replace dependencies (eg: Wheel, Battery) without changing the class that uses it(Car)
- promotes "Code to interface not to implementation" principle
- ability to create and use mock dependency during test (if we want to use a Mock of Wheel during test instead of a real instance.. we can create Mock Wheel object and let DI framework inject to Car)

Share Improve this answer Follow

edited Jul 29, 2020 at 2:09

answered May 22, 2011 at 4:01

gtiwari333
25.1k ● 15 ● 76 ● 105

29 The way I understand this is, instead of instantiating a new object as part of another object, we can inject said object when and if it is needed thus removing the first object's dependency on it. Is that right? – JeliBeanMachine May 28, 2014 at 20:08

I have described this with a coffee shop example here:digigene.com/design-patterns/dependency-injection-coffeeshop – Ali Nem Nov 12, 2016 at 10:41

16 Really like this analogy because it's plain English using a simple analogy. Say I'm Toyota, already spent too much financially and man power on making a car from design to rolling off the assemble line, if there are existing reputable tire producers, why should I start from scratch to make a tire manufacture division i.e. to `new` a tire? I don't. All I have to do is to buy (inject via param) from them, install and wah-lah! So, coming back to programming, say a C# project needs to use an existing library/class, there are two ways to run/debug, 1-add reference to the entire project of this – Jeb50 Aug 9, 2017 at 23:52

(con't),.. external library/class, or 2-add it from the DLL. Unless we have to see what's inside of this external class, adding it as DLL is a easier way. So option 1 is to `new` it, option 2 is pass it in as param. May not be accurate, but simple stupid easy to understand. – Jeb50 Aug 9, 2017 at 23:56

3 @JeliBeanMachine (sorry for extremely late reply to a comment..) it's not that we remove the first object's dependency on the wheel object or battery object, it's that we pass it the dependency, so that we can change the instance or implementation of the dependency. Before: Car has a hardcoded dependency on NepaliRubberWheel. After: Car has an injected dependency on instance of Wheel. – Mikael Ohlson Jan 3, 2018 at 15:43

298

Dependency Injection is a practice where objects are designed in a manner where they receive instances of the objects from other pieces of code, instead of constructing them internally. This means that any object implementing the interface which is required by the object can be substituted in without changing the code, which simplifies testing, and improves decoupling.

For example, consider these clases:

```
public class PersonService {
  public void addManager( Person employee, Person newM
  public void removeManager( Person employee, Person o
  public Group getGroupByManager( Person manager ) { .
}

public class GroupMembershipService() {
  public void addPersonToGroup( Person person, Group g
  public void removePersonFromGroup( Person person, Gr
}
```

In this example, the implementation of `PersonService::addManager` and `PersonService::removeManager` would need an instance of the `GroupMembershipService` in order to do its work. Without Dependency Injection, the traditional way of doing this would be to instantiate a new `GroupMembershipService` in the constructor of `PersonService` and use that instance attribute in both functions. However, if the constructor of `GroupMembershipService` has multiple things it requires, or worse yet, there are some initialization "setters" that need to be called on the `GroupMembershipService`, the code grows rather quickly, and the `PersonService` now depends not only on the `GroupMembershipService` but also everything else that `GroupMembershipService` depends on. Furthermore, the linkage to `GroupMembershipService` is hardcoded into the `PersonService` which means that you can't "dummy up" a `GroupMembershipService` for testing purposes, or to use a strategy pattern in different parts of your application.

With Dependency Injection, instead of instantiating the `GroupMembershipService` within your `PersonService`, you'd either pass it in to the `PersonService` constructor, or else add a Property (getter and setter) to set a local instance of it. This means that your `PersonService` no longer has to worry about how to create a `GroupMembershipService`, it just accepts the ones it's given, and works with them. This also means that anything which is a subclass of `GroupMembershipService`, or implements the `GroupMembershipService` interface can be "injected" into the `PersonService`, and the `PersonService` doesn't need to know about the change.

Share Improve this answer Follow

edited Feb 15, 2019 at 12:40
Mike
14.5k ● 31 ● 118 ● 175

answered Sep 25, 2008 at 6:49
Adam Ness
6,273 ● 4 ● 29 ● 39

---

37 Would have been great if you could give the same code example AFTER using DI – CodyBugstein May 12, 2014 at 9:23

2 "This also means that anything which is a subclass of GroupMembershipService, or implements the GroupMembershipService interface can be "injected" into the PersonService, and the PersonService doesn't need to know about the change." ... This was a very useful takeway for me - thanks! – DefinedRisk Apr 9, 2021 at 8:06

1 "In this example, the implementation of PersonService::addManager and

PersonService::removeManager would need an instance of the GroupMembershipService in order to do its work." My apologies, but I do not see anything in the code sample that shows this. – EvilSnack Dec 26, 2023 at 19:09

204

The accepted answer is a good one - but I would like to add to this that DI is very much like the classic avoiding of hardcoded constants in the code.

When you use some constant like a database name you'd quickly move it from the inside of the code to some config file and pass a variable containing that value to the place where it is needed. The reason to do that is that these constants usually change more frequently than the rest of the code. For example if you'd like to test the code in a test database.

DI is analogous to this in the world of Object Oriented programming. The values there instead of constant literals are whole objects - but the reason to move the code creating them out from the class code is similar - the objects change more frequently then the code that uses them. One important case where such a change is needed is tests.

Share Improve this answer Follow

answered Jan 6, 2011 at 18:33

zby
2,455 ● 1 ● 16 ● 12

22 +1 "the objects change more frequently then the code that uses them". To generalize, add an indirection at points of

flux. Depending on the point of flux, the indirections are called by different names!! – Chethan Mar 30, 2014 at 11:26

204

Let's try simple example with **Car** and **Engine** classes, any car need an engine to go anywhere, at least for now. So below how code will look without dependency injection.

```
public class Car
{
    public Car()
    {
        GasEngine engine = new GasEngine();
        engine.Start();
    }
}

public class GasEngine
{
    public void Start()
    {
        Console.WriteLine("I use gas as my fuel!");
    }
}
```

And to instantiate the Car class we will use next code:

```
Car car = new Car();
```

The issue with this code that we tightly coupled to GasEngine and if we decide to change it to ElectricityEngine then we will need to rewrite Car class. And the bigger the application the more issues and headache we will have to add and use new type of engine.

In other words with this approach is that our high level Car class is dependent on the lower level GasEngine class which violate Dependency Inversion Principle(DIP) from SOLID. DIP suggests that we should depend on abstractions, not concrete classes. So to satisfy this we introduce IEngine interface and rewrite code like below:

```
    public interface IEngine
    {
        void Start();
    }

    public class GasEngine : IEngine
    {
        public void Start()
        {
            Console.WriteLine("I use gas as my fuel!")
        }
    }

    public class ElectricityEngine : IEngine
    {
        public void Start()
        {
            Console.WriteLine("I am electrocar");
        }
    }

    public class Car
    {
        private readonly IEngine _engine;
        public Car(IEngine engine)
        {
            _engine = engine;
        }

        public void Run()
```

```
            {
                _engine.Start();
            }
        }
```

Now our Car class is dependent on only the IEngine interface, not a specific implementation of engine. Now, the only trick is how do we create an instance of the Car and give it an actual concrete Engine class like GasEngine or ElectricityEngine. That's where **Dependency Injection** comes in.

```
    Car gasCar = new Car(new GasEngine());
    gasCar.Run();
    Car electroCar = new Car(new ElectricityEngine());
    electroCar.Run();
```

Here we basically inject(pass) our dependency(Engine instance) to Car constructor. So now our classes have loose coupling between objects and their dependencies, and we can easily add new types of engines without changing the Car class.

The main benefit of the **Dependency Injection** that classes are more loosely coupled, because they do not have hard-coded dependencies. This follows the Dependency Inversion Principle, which was mentioned above. Instead of referencing specific implementations, classes request abstractions (usually **interfaces**) which are provided to them when the class is constructed.

> So in the end **Dependency injection** is just a technique for achieving loose coupling between objects and their dependencies. Rather than directly instantiating dependencies that class needs in order to perform its actions, dependencies are provided to the class (most often) via constructor injection.

Also when we have many dependencies it is very good practice to use Inversion of Control(IoC) containers which we can tell which interfaces should be mapped to which concrete implementations for all our dependencies and we can have it resolve those dependencies for us when it constructs our object. For example, we could specify in the mapping for the IoC container that the **IEngine** dependency should be mapped to the **GasEngine** class and when we ask the IoC container for an instance of our **Car** class, it will automatically construct our **Car** class with a **GasEngine** dependency passed in.

UPDATE: Watched course about EF Core from Julie Lerman recently and also liked her short definition about DI.

> Dependency injection is a pattern to allow your application to inject objects on the fly to classes that need them, without forcing those classes to be responsible for those objects. It allows your code to be more loosely coupled, and Entity

> Framework Core plugs in to this same system of services.

Share Improve this answer Follow

edited Nov 19, 2017 at 19:44
jcjr
1,503 ● 24 ● 40

answered Jul 6, 2017 at 9:42
user2771704
6,194 ● 6 ● 40 ● 42

4 just out of curiosity, how is this different from strategy pattern? This pattern is encapsulating the algorithms and making them interchangeable. It feels like dependency injection and strategy patterns are very similar. – elixir Jan 24, 2018 at 19:15

140

Let's imagine that you want to go fishing:

- Without dependency injection, you need to take care of everything yourself. You need to find a boat, to buy a fishing rod, to look for bait, etc. It's possible, of course, but it puts a lot of responsibility on you. In software terms, it means that you have to perform a lookup for all these things.
- With dependency injection, someone else takes care of all the preparation and makes the required equipment available to you. You will receive ("be injected") the boat, the fishing rod and the bait - all ready to use.

Share Improve this answer Follow

edited Jan 27, 2014 at 23:01
rgettman
178k ● 30 ● 279 ● 360

answered Oct 22, 2012 at 4:47
Olivier Liechti
3,188 ● 2 ● 21 ● 27

---

65 The flipside is, imagine you hire a plumber to redo your bathroom, who then says, "Great, here's a list of the tools and material I need you to get for me". Shouldn't that be the plumber's job? – jscs Jun 12, 2013 at 19:52

So that someone needs to take care of some person it has no business knowing of.. but still decides to gather the list of boat, stick and bait - albeit ready to use. – user2537701 Oct 25, 2013 at 4:00

---

29 @JoshCaswell No, that would the the plumber's emplyer's job. As a client you need plumbing done. For that you need a plumber. The plumber needs it's tools. To get those, it gets equipped by the plumbing company. As a client you don't want to know exactly what the plumber does or need. As a plumber you know what you need, but you just want to do your job, not get everything. As the plumbers employer you are responsible for outfitting your plumbers with what they need before sending them to people's houses. – sara Feb 12, 2016 at 11:00

---

1 @KingOfAllTrades: Of course at some point you have to have someone employing and outfitting plumbers, or you have no plumbers. But you don't have the customer doing it. The customer just asks for a plumber, and gets one already outfitted with what he needs to do his job. With DI, you still eventually have some code to fulfill the dependencies. But you're separating it from the code that does real work. If you take it to its fullest extent, your objects just make their

dependencies known, and the object-graph-building happens outside, often in the init code. – cHao Apr 6, 2017 at 16:33

1 @ThomasRones: pretty much. There are actually different ways to pass the dependencies to the object. One way is to pass them as arguments in the constructor. Another way is to use setter methods (or some sort of annotation). Some people strongly recommend constructor-based DI (see odrotbohm.de/2013/11/why-field-injection-is-evil).
– Olivier Liechti Jul 19, 2022 at 7:28

125

This is the most simple explanation about **Dependency Injection** and **Dependency Injection Container** I have ever seen:

Without Dependency Injection

- Application needs Foo (e.g. a controller), so:
- Application creates Foo
- Application calls Foo
    - Foo needs Bar (e.g. a service), so:
    - Foo creates Bar
    - Foo calls Bar
        - Bar needs Bim (a service, a repository, …), so:
        - Bar creates Bim

        - Bar does something

With Dependency Injection

- Application needs Foo, which needs Bar, which needs Bim, so:
- Application creates Bim
- Application creates Bar and gives it Bim
- Application creates Foo and gives it Bar
- Application calls Foo
    - Foo calls Bar
        - Bar does something

Using a Dependency Injection Container

- Application needs Foo so:
- Application gets Foo from the Container, so:
    - Container creates Bim
    - Container creates Bar and gives it Bim
    - Container creates Foo and gives it Bar
- Application calls Foo
    - Foo calls Bar

        - Bar does something

Dependency Injection and **dependency Injection Containers** are different things:

- Dependency Injection is a method for writing better code
- a DI Container is a tool to help injecting dependencies

You don't need a container to do dependency injection. However a container can help you.

Share Improve this answer Follow

edited Sep 21, 2016 at 9:37

answered May 5, 2016 at 11:53

Peyman Mohamadpour
17.9k ● 24 ● 92 ● 101

Imho, best answer here, because it is not mixing ioc or constructor injection with di. – David Jul 5, 2022 at 8:24

109

Before going to the technical description first visualize it with a real-life example because you will find a lot of technical stuff to learn dependency injection but the majority of the people can't get the core concept of it.

In the first picture, assume that you have a **car factory** with a lot of units. A car is actually built in the **assembly**

unit but it needs **engine**, **seats** as well as **wheels**. So an **assembly unit** is dependent on these all units and they are the **dependencies** of the factory.

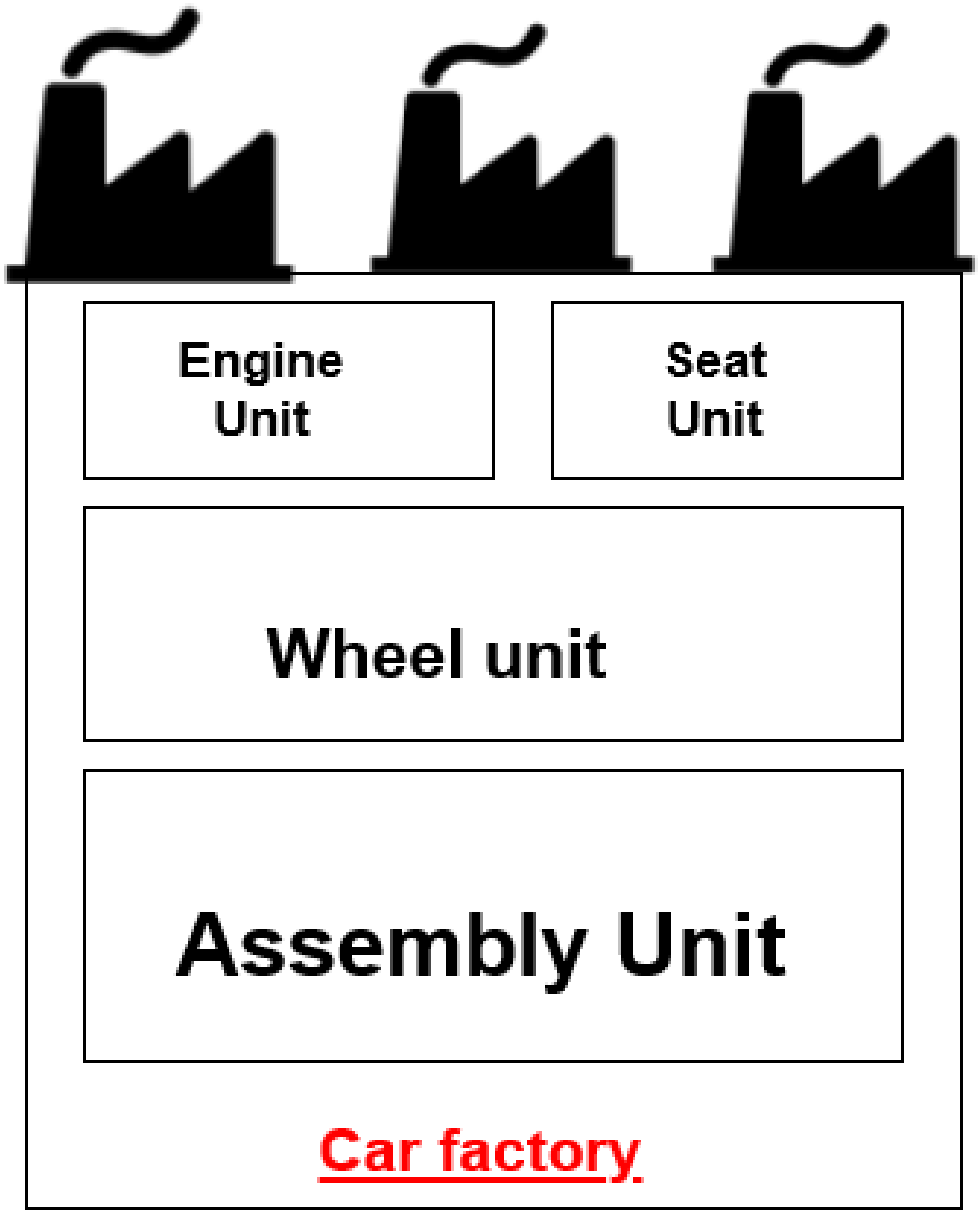
You can feel that now it is too complicated to maintain all of the tasks in this factory because along with the main task (assembling a car in the Assembly unit) you have to also focus on **other units**. It is now very costly to maintain and the factory building is huge so it takes your extra bucks for rent.

Now, look at the second picture. If you find some provider companies that will provide you with the **wheel**, **seat**, and **engine** for cheaper than your self-production cost then now you don't need to make them in your factory. You can rent a smaller building now just for your **assembly unit** which will lessen your maintenance task and reduce your extra rental cost. Now you can also focus only on your main task (Car assembly).

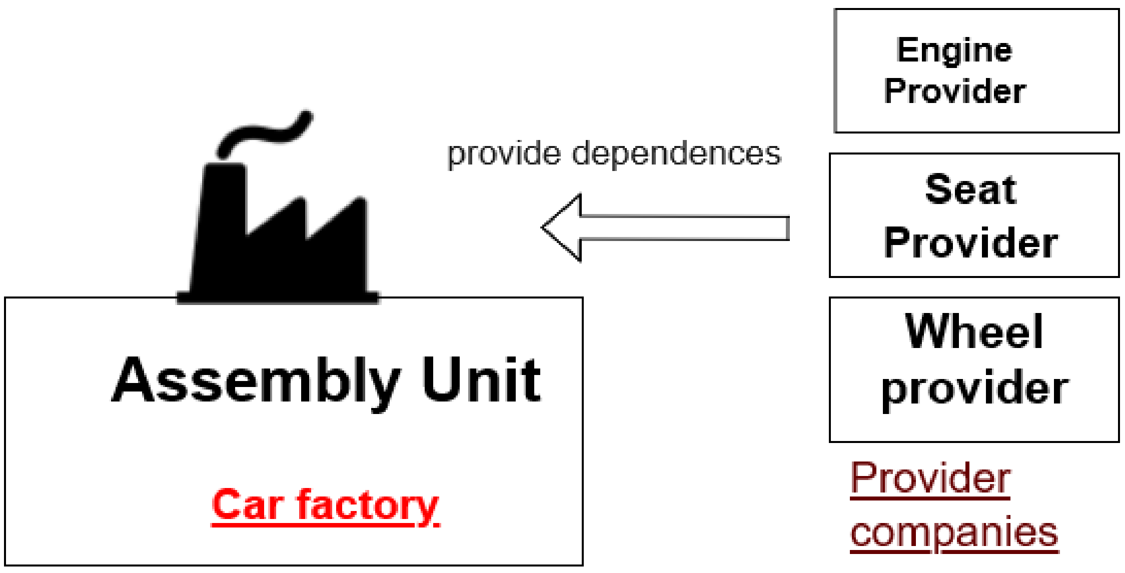
Now we can say that all the **dependencies** for assembling a car are **injected** on the factory from the **providers**. It is an example of a real-life ***Dependency Injection (DI)***.

Now in the technical word, dependency injection is a technique whereby one object (or static method) supplies the dependencies of another object. So, transferring the task of creating the object to someone else and directly using the dependency is called dependency injection.

This will help you now to learn DI with a technical explanation. This will show when to use DI and when you should not.



.



Share Improve this answer

Follow

edited Oct 4, 2020 at 21:18

Lauren Yim
14k ● 2 ● 37 ● 65

answered Jan 27, 2020 at 9:42

Gk Mohammad Emon
6,878 ● 3 ● 47 ● 45

7 Kudos for a visual representation of the concept!
– Bryan Green Mar 10, 2022 at 21:46

70

Doesn't "dependency injection" just mean using parameterized constructors and public setters?

James Shore's article shows the following examples for comparison.

> Constructor without dependency injection:
>
> ```
> public class Example {
>   private DatabaseThingie myDatabase;
> ```

```
  public Example() {
    myDatabase = new DatabaseThingie();
  }

  public void doStuff() {
    ...
    myDatabase.getData();
    ...
  }
}
```

Constructor with dependency injection:

```
public class Example {
  private DatabaseThingie myDatabase;

  public Example(DatabaseThingie useThisDatabaseIn
    myDatabase = useThisDatabaseInstead;
  }

  public void doStuff() {
    ...
    myDatabase.getData();
    ...
  }
}
```

Share Improve this answer Follow

edited Jan 30, 2020 at 15:57

TylerH
21.2k ● 76 ● 79 ● 110

answered May 2, 2013 at 0:40

JaneGoodall
1,506 ● 3 ● 15 ● 23

Surely in the DI version you wouldn't want to initialise the myDatabase object in the no argument constructor? There seems no point and would serve to throw an exception if you tried to call DoStuff without calling the overloaded constructor? – Matt Wilko Nov 21, 2013 at 13:40

Only if `new DatabaseThingie()` doesn't generate a valid myDatabase instance. – JaneGoodall Nov 22, 2013 at 22:49

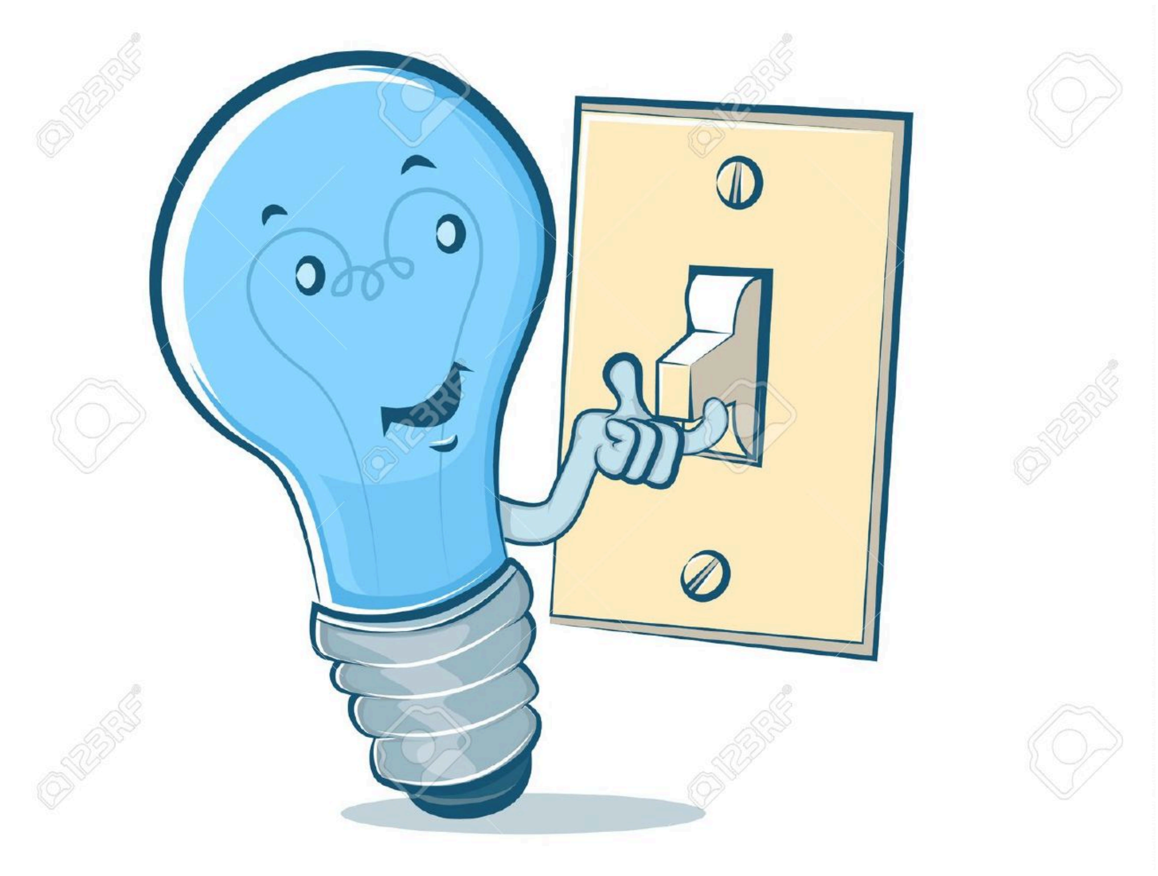
48

To make Dependency Injection concept simple to understand. Let's take an example of switch button to toggle(on/off) a bulb.

Without Dependency Injection

Switch needs to know beforehand which bulb I am connected to (hard-coded dependency). So,

Switch -> PermanentBulb *//switch is directly connected to permanent bulb, testing not possible easily*



```
Switch(){
PermanentBulb = new Bulb();
PermanentBulb.Toggle();
}
```

With Dependency Injection

Switch only knows I need to turn on/off whichever Bulb is passed to me. So,

Switch -> Bulb1 OR Bulb2 OR NightBulb (injected dependency)

```
Switch(AnyBulb){ //pass it whichever bulb you like
AnyBulb.Toggle();
}
```

Modifying James Example for Switch and Bulb:

```
public class SwitchTest {
  TestToggleBulb() {
    MockBulb mockbulb = new MockBulb();

    // MockBulb is a subclass of Bulb, so we can
    // "inject" it here:
    Switch switch = new Switch(mockBulb);

    switch.ToggleBulb();
    mockBulb.AssertToggleWasCalled();
  }
}

public class Switch {
  private Bulb myBulb;
```

```
   public Switch() {
     myBulb = new Bulb();
   }

   public Switch(Bulb useThisBulbInstead) {
     myBulb = useThisBulbInstead;
   }

   public void ToggleBulb() {
     ...
     myBulb.Toggle();
     ...
   }
 }`
```

Share Improve this answer Follow

edited Oct 29, 2016 at 8:23

answered Oct 28, 2016 at 20:30

wakqasahmed
2,089 ● 1 ● 18 ● 23

38

What is Dependency Injection (DI)?

As others have said, *Dependency Injection(DI)* removes the responsibility of direct creation, and management of the lifespan, of other object instances upon which our class of interest (consumer class) is dependent (in the UML sense). These instances are instead passed to our consumer class, typically as constructor parameters or via property setters (the management of the dependency object instancing and passing to the consumer class is usually performed by an *Inversion of Control (IoC)* container, but that's another topic).

DI, DIP and SOLID

Specifically, in the paradigm of Robert C Martin's SOLID principles of Object Oriented Design, `DI` is one of the possible implementations of the Dependency Inversion Principle (DIP). The DIP is the `D` of the `SOLID` mantra - other DIP implementations include the Service Locator, and Plugin patterns.

The objective of the DIP is to decouple tight, concrete dependencies between classes, and instead, to loosen the coupling by means of an abstraction, which can be achieved via an `interface`, `abstract class` or `pure virtual class`, depending on the language and approach used.

Without the DIP, our code (I've called this 'consuming class') is directly coupled to a concrete dependency and is also often burdened with the responsibility of knowing how to obtain, and manage, an instance of this dependency, i.e. conceptually:

```
"I need to create/use a Foo and invoke method
`GetBar()`"
```

Whereas after application of the DIP, the requirement is loosened, and the concern of obtaining and managing the lifespan of the `Foo` dependency has been removed:

```
"I need to invoke something which offers
`GetBar()`"
```

Why use DIP (and DI)?

Decoupling dependencies between classes in this way allows for *easy substitution* of these dependency classes with other implementations which also fulfil the prerequisites of the abstraction (e.g. the dependency can be switched with another implementation of the same interface). Moreover, as others have mentioned, possibly *the* most common reason to decouple classes via the DIP is to allow a consuming class to be tested in isolation, as these same dependencies can now be stubbed and/or mocked.

One consequence of DI is that the lifespan management of dependency object instances is no longer controlled by a consuming class, as the dependency object is now passed into the consuming class (via constructor or setter injection).

This can be viewed in different ways:

- If lifespan control of dependencies by the consuming class needs to be retained, control can be re-established by injecting an (abstract) factory for creating the dependency class instances, into the consumer class. The consumer will be able to obtain instances via a `Create` on the factory as needed, and dispose of these instances once complete.
- Or, lifespan control of dependency instances can be relinquished to an IoC container (more about this below).

When to use DI?

- Where there likely will be a need to substitute a dependency for an equivalent implementation,
- Any time where you will need to unit test the methods of a class in isolation of its dependencies,
- Where uncertainty of the lifespan of a dependency may warrant experimentation (e.g. Hey, `MyDepClass` is thread safe - what if we make it a singleton and inject the same instance into all consumers?)

Example

Here's a simple C# implementation. Given the below Consuming class:

```
public class MyLogger
{
   public void LogRecord(string somethingToLog)
   {
      Console.WriteLine("{0:HH:mm:ss} - {1}",
DateTime.Now, somethingToLog);
   }
}
```

Although seemingly innocuous, it has two `static` dependencies on two other classes, `System.DateTime` and `System.Console`, which not only limit the logging output options (logging to console will be worthless if no one is watching), but worse, it is difficult to automatically test given the dependency on a non-deterministic system clock.

We can however apply `DIP` to this class, by abstracting out the the concern of timestamping as a dependency, and coupling `MyLogger` only to a simple interface:

```
public interface IClock
{
    DateTime Now { get; }
}
```

We can also loosen the dependency on `Console` to an abstraction, such as a `TextWriter`. Dependency Injection is typically implemented as either `constructor` injection (passing an abstraction to a dependency as a parameter to the constructor of a consuming class) or `Setter Injection` (passing the dependency via a `setXyz()` setter or a .Net Property with `{set;}` defined). Constructor Injection is preferred, as this guarantees the class will be in a correct state after construction, and allows the internal dependency fields to be marked as `readonly` (C#) or `final` (Java). So using constructor injection on the above example, this leaves us with:

```
public class MyLogger : ILogger // Others will
depend on our logger.
{
    private readonly TextWriter _output;
    private readonly IClock _clock;

    // Dependencies are injected through the
constructor
    public MyLogger(TextWriter stream, IClock
clock)
    {
        _output = stream;
        _clock = clock;
    }

    public void LogRecord(string somethingToLog)
    {
        // We can now use our dependencies through
the abstraction
        // and without knowledge of the lifespans
of the dependencies
        _output.Write("{0:yyyy-MM-dd HH:mm:ss} -
{1}", _clock.Now, somethingToLog);
    }
}
```

(A concrete `Clock` needs to be provided, which of course could revert to `DateTime.Now`, and the two dependencies need to be provided by an IoC container via constructor injection)

An automated Unit Test can be built, which definitively proves that our logger is working correctly, as we now have control over the dependencies - the time, and we can spy on the written output:

```
[Test]
public void
LoggingMustRecordAllInformationAndStampTheTime()
{
    // Arrange
    var mockClock = new Mock<IClock>();
    mockClock.Setup(c => c.Now).Returns(new
DateTime(2015, 4, 11, 12, 31, 45));
    var fakeConsole = new StringWriter();

    // Act
    new MyLogger(fakeConsole, mockClock.Object)
        .LogRecord("Foo");

    // Assert
    Assert.AreEqual("2015-04-11 12:31:45 - Foo",
fakeConsole.ToString());
}
```

Next Steps

Dependency injection is invariably associated with an Inversion of Control container(IoC), to inject (provide) the concrete dependency instances, and to manage lifespan instances. During the configuration / bootstrapping process, `IoC` containers allow the following to be defined:

- mapping between each abstraction and the configured concrete implementation (e.g. *"any time a consumer requests an `IBar`, return a `ConcreteBar` instance"*)
- policies can be set up for the lifespan management of each dependency, e.g. to create a new object for each consumer instance, to share a singleton dependency instance across all consumers, to share the same dependency instance only across the same thread, etc.
- In .Net, IoC containers are aware of protocols such as `IDisposable` and will take on the responsibility of `Disposing` dependencies in line with the configured lifespan management.

Typically, once IoC containers have been configured / bootstrapped, they operate seamlessly in the background allowing the coder to focus on the code at hand rather than worrying about dependencies.

> The key to DI-friendly code is to avoid static coupling of classes, and not to use new() for the creation of Dependencies

As per above example, decoupling of dependencies does require some design effort, and for the developer, there is a paradigm shift needed to break the habit of `new`ing dependencies directly, and instead trusting the container to manage dependencies.

But the benefits are many, especially in the ability to thoroughly test your class of interest.

Note : The creation / mapping / projection (via `new ..()`) of POCO / POJO / Serialization DTOs / Entity Graphs / Anonymous JSON projections et al - i.e. "Data only" classes or records - used or returned from methods are *not* regarded as Dependencies (in the UML sense) and not subject to DI. Using `new` to project these is just fine.

Share Improve this answer Follow

edited May 23, 2017 at 11:33

Community Bot
1 ● 1

answered Apr 28, 2015 at 20:17

StuartLC
107k ● 18 ● 220 ● 292

1 The problem is DIP != DI. DIP is about decoupling abstraction from implementation: A. High-level modules should not depend on low-level modules. Both should depend on abstractions. B. Abstractions should not depend on details.

Details should depend on abstractions. DI is a way to decouple object creation from object use. – Ricardo Rivaldo Oct 1, 2015 at 0:43

Yes, the distinction is clearly stated in my paragraph 2, *"DI one of the possible implementations of DIP"*, in Uncle Bob's SOLID paradigm. I've also made this clear in an earlier post. – StuartLC Mar 16, 2017 at 6:13

28

The whole point of Dependency Injection (DI) is to keep application source code **clean** and **stable**:

- **clean** of dependency initialization code
- **stable** regardless of dependency used

Practically, every design pattern separates concerns to make future changes affect minimum files.

The specific domain of DI is delegation of dependency configuration and initialization.

Example: DI with shell script

If you occasionally work outside of Java, recall how `source` is often used in many scripting languages (Shell, Tcl, etc., or even `import` in Python misused for this purpose).

Consider simple `dependent.sh` script:

```
#!/bin/sh
# Dependent
```

```
touch         "one.txt" "two.txt"
archive_files "one.txt" "two.txt"
```

The script is dependent: it won't execute successfully on its own (`archive_files` is not defined).

You define `archive_files` in `archive_files_zip.sh` implementation script (using `zip` in this case):

```
#!/bin/sh
# Dependency
function archive_files {
    zip files.zip "$@"
}
```

Instead of `source`-ing implementation script directly in the dependent one, you use an `injector.sh` "container" which wraps both "components":

```
#!/bin/sh
# Injector
source ./archive_files_zip.sh
source ./dependent.sh
```

The `archive_files` *dependency* has just been *injected* into *dependent* script.

You could have injected dependency which implements `archive_files` using `tar` or `xz`.

Example: removing DI

If `dependent.sh` script used dependencies directly, the approach would be called *dependency lookup* (which is opposite to *dependency injection*):

```
#!/bin/sh
# Dependent

# dependency look-up
source ./archive_files_zip.sh

touch         "one.txt" "two.txt"
archive_files "one.txt" "two.txt"
```

Now the problem is that dependent "component" has to perform initialization itself.

The "component"'s source code is neither **clean** nor **stable** because every changes in initialization of dependencies requires new release for "components"'s source code file as well.

Last words

DI is not as largely emphasized and popularized as in Java frameworks.

But it's a generic approach to split concerns of:

- application **development** (**single** source code release lifecycle)
- application **deployment** (**multiple** target environments with independent lifecycles)

Using configuration only with *dependency lookup* does not help as number of configuration parameters may change per dependency (e.g. new authentication type) as well as number of supported types of dependencies (e.g. new database type).

Share Improve this answer Follow

edited Sep 25, 2015 at 17:56

community wiki
8 revs, 3 users 97%
uvsmtid

I would add the ability to complete a particular class (testing) without having to complete its dependencies, as a purpose for DI. – David Sep 7, 2016 at 19:39

28

All the above answers are good, my aim is to explain the concept in a simple way so that anyone without a programming knowledge can also understand concept

Dependency injection is one of the design pattern that help us to create complex systems in a simpler manner.

We can see a wide variety of application of this pattern in our day to day life. Some of the examples are Tape recorder, VCD, CD Drive etc.

The above image is an image of Reel-to-reel portable tape recorder, mid-20th century. Source.

The primary intention of a tape recorder machine is to record or playback sound.

While designing a system it require a reel to record or playback sound or music. There are two possibilities for designing this system

1. we can place the reel inside the machine
2. we can provide a hook for the reel where it can be placed.

If we use the first one we need to open the machine to change the reel. if we opt for the second one, that is placing a hook for reel, we are getting an added benefit of playing any music by changing the reel. and also reducing the function only to playing whatever in the reel.

Like wise dependency injection is the process of externalizing the dependencies to focus only on the specific functionality of the component so that independent components can be coupled together to form a complex system.

The main benefits we achieved by using dependency injection.

- High cohesion and loose coupling.
- Externalizing dependency and looking only on responsibility.
- Making things as components and to combine to form a large systems with high capabilities.
- It helps to develop high quality components since they are independently developed they are properly tested.
- It helps to replace the component with another if one fails.

Now a days these concept forms the basis of well known frameworks in programming world. The Spring Angular etc are the well-known software frameworks built on the top of this concept

Dependency injection is a pattern used to create instances of objects that other objects rely upon without knowing at compile time which class will be used to provide that functionality or simply the way of injecting properties to an object is called dependency injection.

Example for Dependency injection

Previously we are writing code like this

```
Public MyClass{
 DependentClass dependentObject
 /*
  At somewhere in our code we need to instantiate
  the object with new operator  inorder to use it
or perform some method.
  */
  dependentObject= new DependentClass();
  dependentObject.someMethod();
}
```

With Dependency injection, the dependency injector will take off the instantiation for us

```
Public MyClass{
 /* Dependency injector will instantiate object*/
 DependentClass dependentObject

 /*
  At somewhere in our code we perform some method.
  The process of  instantiation will be handled by
the dependency injector
 */

  dependentObject.someMethod();
}
```

You can also read

[Difference between Inversion of Control & Dependency Injection]

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12

Community Bot
1 ● 1

answered Nov 3, 2017 at 7:36

SAMUEL
8,542 ● 3 ● 45 ● 43

20

Example, we have 2 class `Client` and `Service`. `Client` will use `Service`

```
public class Service {
    public void doSomeThingInService() {
        // ...
    }
}
```

Without Dependency Injection

Way 1)

```
public class Client {
    public void doSomeThingInClient() {
        Service service = new Service();
        service.doSomeThingInService();
    }
}
```

Way 2)

```
public class Client {
    Service service = new Service();
    public void doSomeThingInClient() {
        service.doSomeThingInService();
    }
}
```

Way 3)

```
public class Client {
    Service service;
    public Client() {
        service = new Service();
    }
    public void doSomeThingInClient() {
        service.doSomeThingInService();
    }
}
```

1) 2) 3) Using

```
Client client = new Client();
client.doSomeThingInService();
```

Advantages

- Simple

Disadvantages

- Hard for test `Client` class
- When we change `Service` constructor, we need to change code in all place create `Service` object

Use Dependency Injection

Way 1) Constructor injection

```
public class Client {
    Service service;

    Client(Service service) {
        this.service = service;
    }

    // Example Client has 2 dependency
    // Client(Service service, IDatabas database)
{
    //    this.service = service;
    //    this.database = database;
    // }

    public void doSomeThingInClient() {
        service.doSomeThingInService();
    }
}
```

Using

```
Client client = new Client(new Service());
// Client client = new Client(new Service(), new
SqliteDatabase());
client.doSomeThingInClient();
```

Way 2) Setter injection

```
public class Client {
    Service service;

    public void setService(Service service) {
        this.service = service;
    }
    public void doSomeThingInClient() {
        service.doSomeThingInService();
    }
}
```

Using

```
Client client = new Client();
client.setService(new Service());
client.doSomeThingInClient();
```

Way 3) Interface injection

Check https://en.wikipedia.org/wiki/Dependency_injection

===

Now, this code is already follow `Dependency Injection` and it is easier for test `Client` class.
However, we still use `new Service()` many time and it is not good when change `Service` constructor. To prevent it, we can use DI injector like
1) Simple manual `Injector`

```
public class Injector {
    public static Service provideService(){
        return new Service();
    }

    public static IDatabase provideDatatBase(){
        return new SqliteDatabase();
    }
    public static ObjectA provideObjectA(){
        return new ObjectA(provideService(...));
    }
}
```

Using

```
Service service = Injector.provideService();
```

2) Use library: For Android dagger2

Advantages

- Make test easier
- When you change the `Service`, you only need to change it in Injector class
- If you use use `Constructor Injection`, when you look at constructor of `Client`, you will see how many dependency of `Client` class

Disadvantages

- If you use use `Constructor Injection`, the `Service` object is created when `Client` created, sometime we use function in `Client` class without use `Service` so created `Service` is wasted

Dependency Injection definition

https://en.wikipedia.org/wiki/Dependency_injection

> A dependency is an object that can be used (`Service`)

> An injection is the passing of a dependency (`Service`) to a dependent object (`Client`) that would use it

Share Improve this answer

Follow

edited Aug 14, 2018 at 8:32

Imam Bux

1,136 ● 12 ● 29

answered Mar 14, 2018 at 4:41

Linh

60.8k ● 24 ● 273 ● 295

17

What is dependency Injection?

Dependency Injection(DI) means to decouple the objects which are dependent on each other. Say object A is dependent on Object B so the idea is to decouple these object from each other. We don't need to hard code the object using new keyword rather sharing dependencies to objects at runtime in spite of compile time. If we talk about

How Dependency Injection works in Spring:

We don't need to hard code the object using new keyword rather define the bean dependency in the configuration file. The spring container will be responsible for hooking up all.

Inversion of Control (IOC)

IOC is a general concept and it can be expressed in many different ways and Dependency Injection is one concrete example of IOC.

Two types of Dependency Injection:

1. Constructor Injection
2. Setter Injection

1. Constructor-based dependency injection:

Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.

```
public class Triangle {

private String type;

public String getType(){
    return type;
 }

public Triangle(String type){   //constructor
injection
    this.type=type;
 }
}
<bean id=triangle" class
="com.test.dependencyInjection.Triangle">
```

```
        <constructor-arg value="20"/>
  </bean>
```

2. Setter-based dependency injection:

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

```
public class Triangle{

 private String type;

 public String getType(){
    return type;
  }
 public void setType(String type){
//setter injection
    this.type = type;
  }
 }

<!-- setter injection -->
 <bean id="triangle"
class="com.test.dependencyInjection.Triangle">
        <property name="type"
value="equivialteral"/>
```

NOTE: It is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies. Note that the if we use annotation based than @Required annotation on a setter can be used to make setters as a required dependencies.

Share Improve this answer Follow

edited Mar 5, 2016 at 9:46

answered Dec 4, 2015 at 5:50

Harleen
803 ● 4 ● 16 ● 25

15

The best analogy I can think of is the surgeon and his assistant(s) in an operation theater, where the surgeon is the main person and his assistant who provides the various surgical components when he needs it so that the surgeon can concentrate on the one thing he does best (surgery). Without the assistant the surgeon has to get the components himself every time he needs one.

DI for short, is a technique to remove a common additional responsibility (burden) on components to fetch the dependent components, by providing them to it.

DI brings you closer to the Single Responsibility (SR) principle, like the `surgeon who can concentrate on surgery`.

When to use DI : I would recommend using DI in almost all production projects (small/big), particularly in ever changing business environments :)

Why : Because you want your code to be easily testable, mockable etc so that you can quickly test your changes and push it to the market. Besides why would you not when you there are lots of awesome free tools/frameworks to support you in your journey to a codebase where you have more control.

Share Improve this answer Follow

answered Apr 5, 2016 at 16:15

Anwar Husain
1,814 ● 18 ● 20

@WindRider Thanks. I can't agree more. The human life and human body are magnificent examples of design excellence..the spine is an excellent example of an ESB :)... – Anwar Husain May 18, 2016 at 9:12

14

It means that objects should only have as many dependencies as is needed to do their job and the dependencies should be few. Furthermore, an object’s dependencies should be on interfaces and not on “concrete” objects, when possible. (A concrete object is any object created with the keyword new.) Loose coupling promotes greater reusability, easier maintainability, and allows you to easily provide “mock” objects in place of expensive services.

The “Dependency Injection” (DI) is also known as “Inversion of Control” (IoC), can be used as a technique for encouraging this loose coupling.

There are two primary approaches to implementing DI:

1. Constructor injection
2. Setter injection

Constructor injection

It’s the technique of passing objects dependencies to its constructor.

Note that the constructor accepts an interface and not concrete object. Also, note that an exception is thrown if the orderDao parameter is null. This emphasizes the importance of receiving a valid dependency. Constructor Injection is, in my opinion, the preferred mechanism for giving an object its dependencies. It is clear to the developer while invoking the object which dependencies need to be given to the “Person” object for proper execution.

Setter Injection

But consider the following example… Suppose you have a class with ten methods that have no dependencies, but you’re adding a new method that does have a dependency on IDAO. You could change the constructor to use Constructor Injection, but this may force you to changes to all constructor calls all over the place. Alternatively, you could just add a new constructor that takes the dependency, but then how does a developer easily know when to use one constructor over the other. Finally, if the dependency is very expensive to create, why should it be created and passed to the constructor when it may only be used rarely? “Setter Injection” is another DI technique that can be used in situations such as this.

Setter Injection does not force dependencies to be passed to the constructor. Instead, the dependencies are set onto public properties exposed by the object in need. As implied previously, the primary motivators for doing this include:

1. Supporting dependency injection without having to modify the constructor of a legacy class.
2. Allowing expensive resources or services to be created as late as possible and only when needed.

Here is the example of how the above code would look like:

```
public class Person {
    public Person() {}

    public IDAO Address {
        set { addressdao = value; }
        get {
            if (addressdao == null)
              throw new
MemberAccessException("addressdao" +
                             " has not been
initialized");
            return addressdao;
        }
    }

    public Address GetAddress() {
       // ... code that uses the addressdao object
       // to fetch address details from the
datasource ...
    }

    // Should not be called directly;
    // use the public property instead
    private IDAO addressdao;
```

Share Improve this answer

Follow

edited Dec 9, 2015 at 19:54

CubeJockey
2,219 ● 8 ● 25 ● 31

answered Jan 7, 2014 at 11:45

Piyush Deshpande
195 ● 1 ● 5

3 I think your first paragraph strays away from the question, and isn't at all the definition of DI (i.e., you are trying to define SOLID, not DI). Technically, even if you have 100 dependencies, you could still use dependency injection. Similarly, it is possible to inject concrete dependencies--it is still dependency injection. – Jay Sullivan Jan 20, 2014 at 5:36

11

I know there are already many answers, but I found this very helpful: http://tutorials.jenkov.com/dependency-injection/index.html

No Dependency:

```
public class MyDao {

  protected DataSource dataSource = new DataSourceImpl
    "driver", "url", "user", "password");

  //data access methods...
  public Person readPerson(int primaryKey) {...}
}
```

Dependency:

```
public class MyDao {

  protected DataSource dataSource = null;

  public MyDao(String driver, String url, String user,
    this.dataSource = new DataSourceImpl(driver, url,
  }

  //data access methods...
  public Person readPerson(int primaryKey) {...}
}
```

Notice how the `DataSourceImpl` instantiation is moved into a constructor. The constructor takes four parameters which are the four values needed by the `DataSourceImpl`. Though the `MyDao` class still depends on these four values, it no longer satisfies these dependencies itself. They are provided by whatever class creating a `MyDao` instance.

Share Improve this answer

Follow

edited Mar 29, 2020 at 11:28

Kevin Languasco

2,426 ● 1 ● 16 ● 25

answered Nov 12, 2014 at 14:37

Alex

6,149 ● 13 ● 45 ● 81

2 Wouldn't DI pass you by interface your DataSourceImp already constructed? – PmanAce May 1, 2016 at 2:04

10

Dependency Injection means a way (actually **any-way**) for one part of code (e.g a class) to have access to dependencies (other parts of code, e.g other classes, it depends upon) in a modular way without them being hardcoded (so they can change or be overriden freely, or even be loaded at another time, as needed)

(and ps , yes it has become an overly-hyped 25$ name for a rather simple, concept), my `.25` cents

Share Improve this answer Follow

answered Dec 10, 2015 at 15:50

Nikos M.
8,325 ● 4 ● 40 ● 45

10

I think since everyone has written for DI, let me ask a few questions..

1. When you have a configuration of DI where all the actual implementations(not interfaces) that are going to be injected into a class (for e.g services to a controller) why is that not some sort of hard-coding?
2. What if I want to change the object at runtime? For example, my config already says when I instantiate MyController, inject for FileLogger as ILogger. But I might want to inject DatabaseLogger.
3. Every time I want to change what objects my AClass needs, I need to now look into two places - The class itself and the configuration file. How does that make life easier?

4. If Aproperty of AClass is not injected, is it harder to mock it out?
5. Going back to the first question. If using new object() is bad, how come we inject the implementation and not the interface? I think a lot of you are saying we're in fact injecting the interface but the configuration makes you specify the implementation of that interface ..not at runtime .. it is hardcoded during compile time.

This is based on the answer @Adam N posted.

Why does PersonService no longer have to worry about GroupMembershipService? You just mentioned GroupMembership has multiple things(objects/properties) it depends on. If GMService was required in PService, you'd have it as a property. You can mock that out regardless of whether you injected it or not. The only time I'd like it to be injected is if GMService had more specific child classes, which you wouldn't know until runtime. Then you'd want to inject the subclass. Or if you wanted to use that as either singleton or prototype. To be honest, the configuration file has everything hardcoded as far as what subclass for a type (interface) it is going to inject during compile time.

EDIT

A nice comment by Jose Maria Arranz on DI

DI increases cohesion by removing any need to determine the direction of dependency and write any glue *code.*

False. The direction of dependencies is in XML form or as annotations, your dependencies are written as XML code and annotations. XML and annotations ARE source code.

DI reduces coupling by making all of your components modular (i.e. replaceable) and have well-defined interfaces to each other.

False. You do not need a DI framework to build a modular code based on interfaces.

About replaceable: with a very simple .properties archive and Class.forName you can define which classes can change. If ANY class of your code can be changed, Java is not for you, use an scripting language. By the way: annotations cannot be changed without recompiling.

In my opinion there is one only reason for DI frameworks: boiler plate reduction. With a well done factory system you can do the same, more controlled and more predictable as your preferred DI framework, DI frameworks promise code reduction (XML and annotations are source code too). The problem is this boiler plate reduction is just real in very very simple cases (one instance-per class and similar), sometimes in the real world picking the appropriated service object is not as easy as mapping a class to a singleton object.

Share Improve this answer Follow

edited Jun 5, 2020 at 7:18

drac_o
437 ● 5 ● 12

answered Oct 25, 2013 at 3:56

user2537701

8

The popular answers are unhelpful, because they define dependency injection in a way that isn't useful. Let's agree that by "dependency" we mean some pre-existing other object that our object X needs. But we don't say we're doing "dependency injection" when we say

```
$foo = Foo->new($bar);
```

We just call that passing parameters into the constructor. We've been doing that regularly ever since constructors were invented.

"Dependency injection" is considered a type of "inversion of control", which means that some logic is taken out of the caller. That isn't the case when the caller passes in parameters, so if that were DI, DI would not imply inversion of control.

DI means there is an intermediate level between the caller and the constructor which manages dependencies. A Makefile is a simple example of dependency injection. The "caller" is the person typing "make bar" on the command line, and the "constructor" is the compiler. The Makefile specifies that bar depends on foo, and it does a

```
gcc -c foo.cpp; gcc -c bar.cpp
```

before doing a

```
gcc foo.o bar.o -o bar
```

The person typing "make bar" doesn't need to know that bar depends on foo. The dependency was injected between "make bar" and gcc.

The main purpose of the intermediate level is not just to pass in the dependencies to the constructor, but to list all the dependencies in *just one place*, and to hide them from the coder (not to make the coder provide them).

Usually the intermediate level provides factories for the constructed objects, which must provide a role that each requested object type must satisfy. That's because by having an intermediate level that hides the details of construction, you've already incurred the abstraction penalty imposed by factories, so you might as well use factories.

Share Improve this answer Follow

edited Jun 2, 2015 at 18:32

answered Jun 2, 2015 at 18:09

Phil Goetz

605 ● 5 ● 14

7

From the Book, 'Well-Grounded Java Developer: Vital techniques of Java 7 and polyglot programming

> DI is a particular form of IoC, whereby the process of finding your dependencies is outside the direct control of your currently executing code.

Share Improve this answer Follow

answered May 18, 2013 at 19:27

TastyCode
5,799 ● 4 ● 40 ● 43

7

Dependency injection is one possible solution to what could generally be termed the "Dependency Obfuscation" requirement. Dependency Obfuscation is a method of taking the 'obvious' nature out of the process of providing a dependency to a class that requires it and therefore obfuscating, in some way, the provision of said dependency to said class. This is not necessarily a bad thing. In fact, by obfuscating the manner by which a dependency is provided to a class then something outside the class is responsible for creating the dependency which means, in various scenarios, a different implementation of the dependency can be supplied to the class without making any changes to the class. This is great for switching between production and testing modes (eg., using a 'mock' service dependency).

Unfortunately the bad part is that some people have assumed you need a specialized framework to do dependency obfuscation and that you are somehow a 'lesser' programmer if you choose not to use a particular framework to do it. Another, extremely disturbing myth, believed by many, is that dependency injection is the only way of achieving dependency obfuscation. This is demonstrably and historically and obviously 100% wrong but you will have trouble convincing some people that there are alternatives to dependency injection for your dependency obfuscation requirements.

Programmers have understood the dependency obfuscation requirement for years and many alternative solutions have evolved both before and after dependency injection was conceived. There are Factory patterns but there are also many options using ThreadLocal where no injection to a particular instance is needed - the dependency is effectively injected into the thread which has the benefit of making the object available (via convenience static getter methods) to *any* class that requires it without having to add annotations to the classes that require it and set up intricate XML 'glue' to make it happen. When your dependencies are required for persistence (JPA/JDO or whatever) it allows you to achieve 'tranaparent persistence' much easier and with domain model and business model classes made up purely of POJOs (i.e. no framework specific/locked in annotations).

Follow

answered Apr 1, 2014 at 22:21

Volksman

2,048 ● 1 ● 24 ● 18

6

Dependency Injection for 5 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 be even 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.

Share Improve this answer

Follow

answered Sep 7, 2019 at 15:16

Nithin Prasad

584 ● 1 ● 9 ● 22

I'm not sure whether this explanation is misleading, as the unfavourable *service locator antipattern* seems to be a valid solution the described problem. – Codor Sep 10, 2020 at 11:52

5

In simple words dependency injection (DI) is the way to remove dependencies or tight coupling between different object. Dependency Injection gives a cohesive behavior to each object.

DI is the implementation of IOC principal of Spring which says "Don't call us we will call you". Using dependency injection programmer doesn't need to create object using the new keyword.

Objects are once loaded in Spring container and then we reuse them whenever we need them by fetching those objects from Spring container using getBean(String beanName) method.

Share Improve this answer Follow

edited May 8, 2014 at 19:57
Marius Waldal
9,912 ● 4 ● 31 ● 45

answered Nov 12, 2013 at 5:19
Waqas Ahmed
5,119 ● 3 ● 41 ● 45

5

from Book **Apress.Spring.Persistence.with.Hibernate.Oct.2010**

> The purpose of dependency injection is to decouple the work of resolving external software components from your application business logic.Without dependency injection, the details of how a component accesses required services can get muddled in with the component’s code. This not only increases the potential for errors,

> adds code bloat, and magnifies maintenance complexities; it couples components together more closely, making it difficult to modify dependencies when refactoring or testing.

Share Improve this answer Follow

answered Aug 28, 2015 at 10:00

BERGUIGA Mohamed Amine

6,280 ● 3 ● 42 ● 41

5

Dependency Injection (DI) is part of Dependency Inversion Principle (DIP) practice, which is also called Inversion of Control (IoC). Basically you need to do DIP because you want to make your code more modular and unit testable, instead of just one monolithic system. So you start identifying parts of the code that can be separated from the class and abstracted away. Now the implementation of the abstraction need to be injected from outside of the class. Normally this can be done via constructor. So you create a constructor that accepts the abstraction as a parameter, and this is called dependency injection (via constructor). For more explanation about DIP, DI, and IoC container you can read Here

Share Improve this answer Follow

edited Jun 3, 2016 at 23:41

answered Jun 3, 2016 at 23:34

kusnaditjung tjung

339 ● 3 ● 3

5

Dependency Injection (DI) is one from Design Patterns, which uses the basic feature of OOP - the relationship in one object with another object. While inheritance inherits one object to do more complex and specific another object, relationship or association simply creates a pointer to another object from one object using attribute. The power of DI is in combination with other features of OOP as are interfaces and hiding code. Suppose, we have a customer (subscriber) in the library, which can borrow only one book for simplicity.

Interface of book:

```
package com.deepam.hidden;

public interface BookInterface {

public BookInterface setHeight(int height);
public BookInterface setPages(int pages);
public int getHeight();
public int getPages();

public String toString();
}
```

Next we can have many kind of books; one of type is fiction:

```
package com.deepam.hidden;

public class FictionBook implements BookInterface
{
int height = 0; // height in cm
int pages = 0; // number of pages

/** constructor */
public FictionBook() {
    // TODO Auto-generated constructor stub
}

@Override
public FictionBook setHeight(int height) {
  this.height = height;
  return this;
}

@Override
public FictionBook setPages(int pages) {
  this.pages = pages;
  return this;
}

@Override
public int getHeight() {
    // TODO Auto-generated method stub
    return height;
}

@Override
public int getPages() {
    // TODO Auto-generated method stub
    return pages;
}

@Override
```

Now subscriber can have association to the book:

```
package com.deepam.hidden;

import java.lang.reflect.Constructor;
import
java.lang.reflect.InvocationTargetException;

public class Subscriber {
BookInterface book;
```

```
/** constructor*/
public Subscriber() {
    // TODO Auto-generated constructor stub
}

// injection I
public void setBook(BookInterface book) {
    this.book = book;
}

// injection II
public BookInterface setBook(String bookName) {
    try {
        Class<?> cl = Class.forName(bookName);
        Constructor<?> constructor =
cl.getConstructor(); // use it for parameters in
constructor
        BookInterface book = (BookInterface)
constructor.newInstance();
        //book = (BookInterface)
Class.forName(bookName).newInstance();
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (NoSuchMethodException e) {
```

All the three classes can be hidden for it's own implementation. Now we can use this code for DI:

```
package com.deepam.implement;

import com.deepam.hidden.Subscriber;
import com.deepam.hidden.FictionBook;

public class CallHiddenImplBook {

public CallHiddenImplBook() {
    // TODO Auto-generated constructor stub
}

public void doIt() {
    Subscriber ab = new Subscriber();

    // injection I
    FictionBook bookI = new FictionBook();
    bookI.setHeight(30); // cm
    bookI.setPages(250);
    ab.setBook(bookI); // inject
    System.out.println("injection I " +
ab.getBook().toString());

    // injection II
    FictionBook bookII = ((FictionBook)
ab.setBook("com.deepam.hidden.FictionBook")).setHeig
// inject and set
    System.out.println("injection II " +
ab.getBook().toString());
}

public static void main(String[] args) {
    CallHiddenImplBook kh = new
CallHiddenImplBook();
    kh.doIt();
}
}
```

There are many different ways how to use dependency injection. It is possible to combine it with Singleton, etc., but still in basic it is only association realized by creating attribute of object type inside another object. The usefulness is only and only in feature, that code, which we should write again and again is always prepared and done for us forward. This is why DI so closely binded with Inversion of Control (IoC) which means, that our program passes control another running module, which does injections of beans to our code. (Each object, which can be injected can be signed or considered as a Bean.) For example in Spring it is done by creating and initialization *ApplicationContext* container, which does this work for us. We simply in our code create the Context and invoke initialization the beans. In that moment injection has been done automatically.

Share Improve this answer Follow

edited May 19, 2017 at 11:28

answered Sep 20, 2015 at 12:33

hariprasad
585 ● 11 ● 21

5

I would propose a slightly different, short and precise definition of what Dependency Injection is, focusing on the primary goal, not on the technical means (following along from here):

> Dependency Injection is the process of creating the static, stateless graph of service objects, where each service is parametrised by its dependencies.

The objects that we create in our applications (regardless if we use Java, C# or other object-oriented language) usually fall into one of two categories: stateless, static and global “service objects” (modules), and stateful, dynamic and local “data objects”.

The module graph - the graph of service objects - is typically created on application startup. This can be done using a container, such as Spring, but can also be done manually, by passing parameters to object constructors. Both ways have their pros and cons, but a framework definitely isn't necessary to use DI in your application.

One requirement is that the services must be parametrised by their dependencies. What this means exactly depends on the language and approach taken in a given system. Usually, this takes the form of constructor parameters, but using setters is also an option. This also means that the dependencies of a service are hidden (when invoking a service method) from the users of the service.

When to use? I would say whenever the application is large enough that encapsulating logic into separate modules, with a dependency graph between the modules gives a gain in readability and explorability of the code.

Share Improve this answer Follow

answered Sep 21, 2018 at 9:42

adamw
8,576 ● 4 ● 31 ● 34