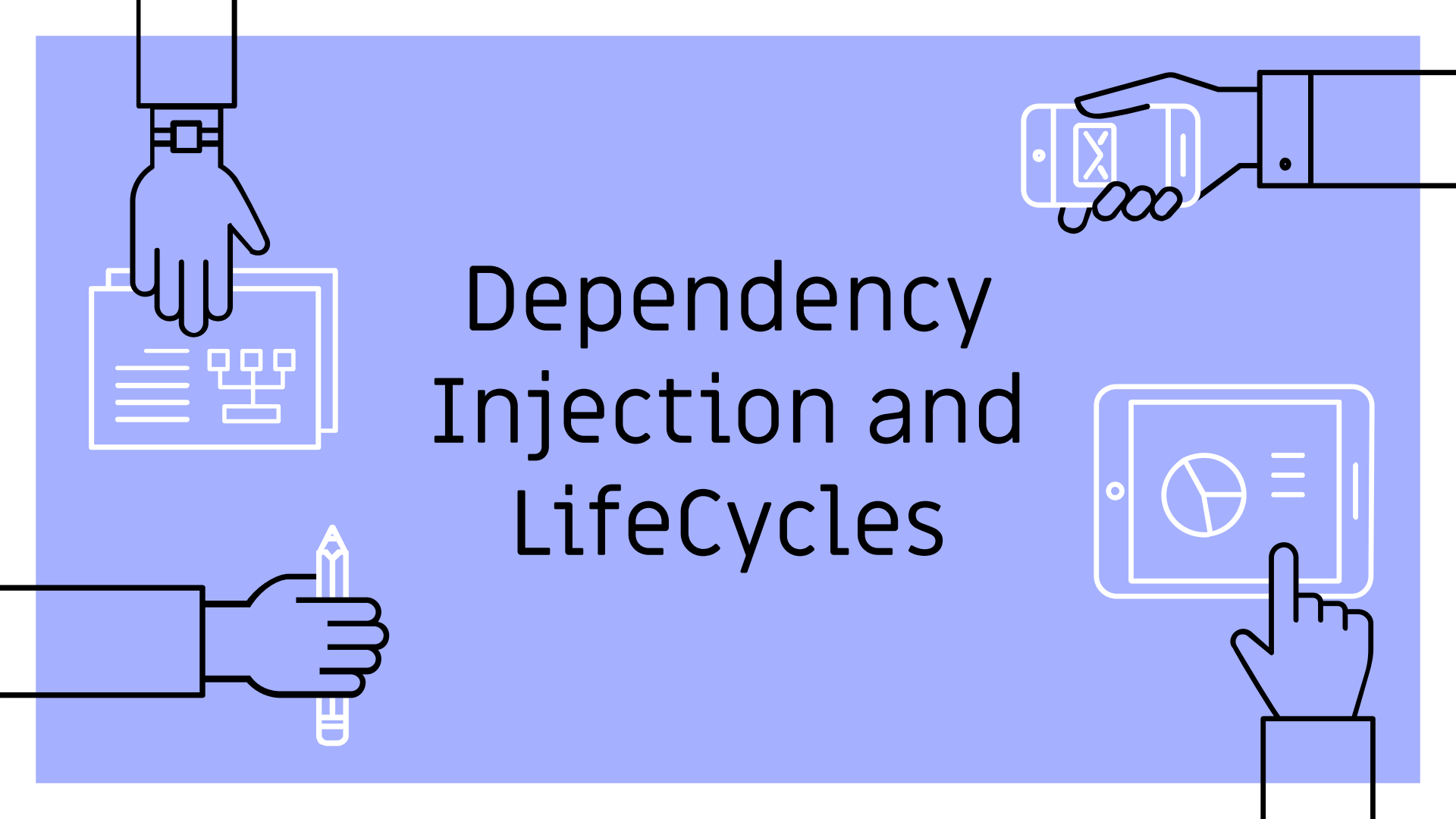
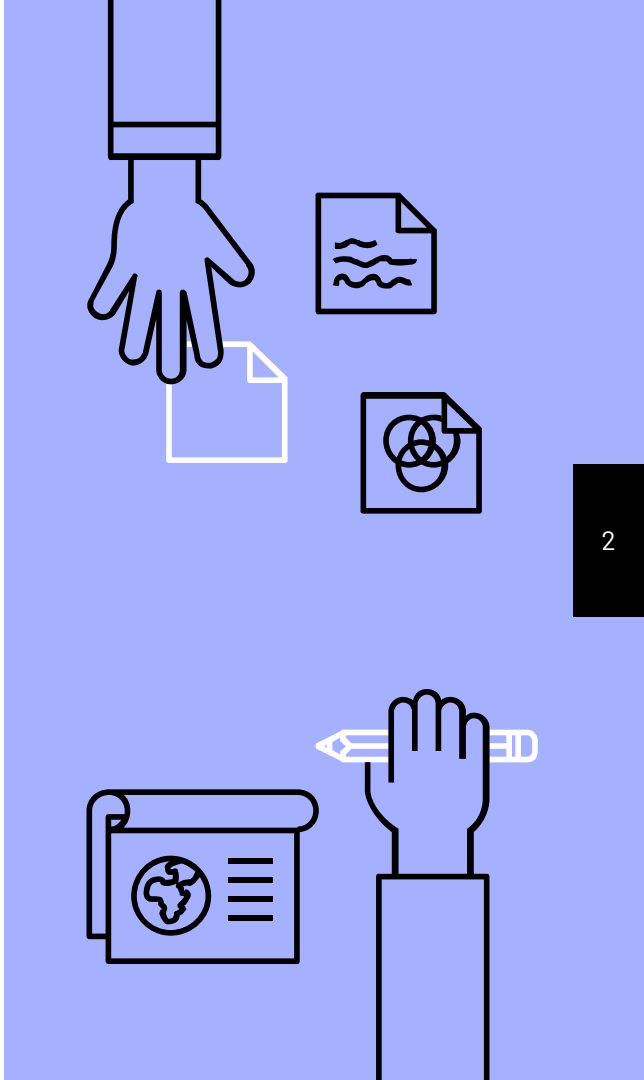
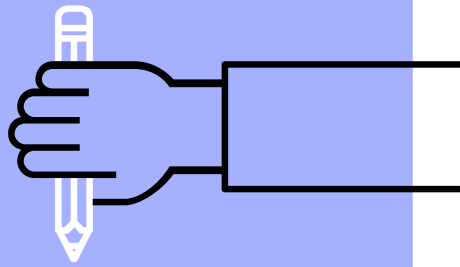

Dependency
Injection and
LifeCycles

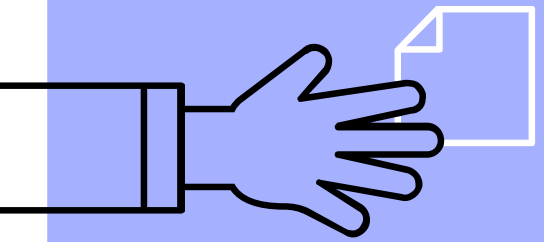
Index

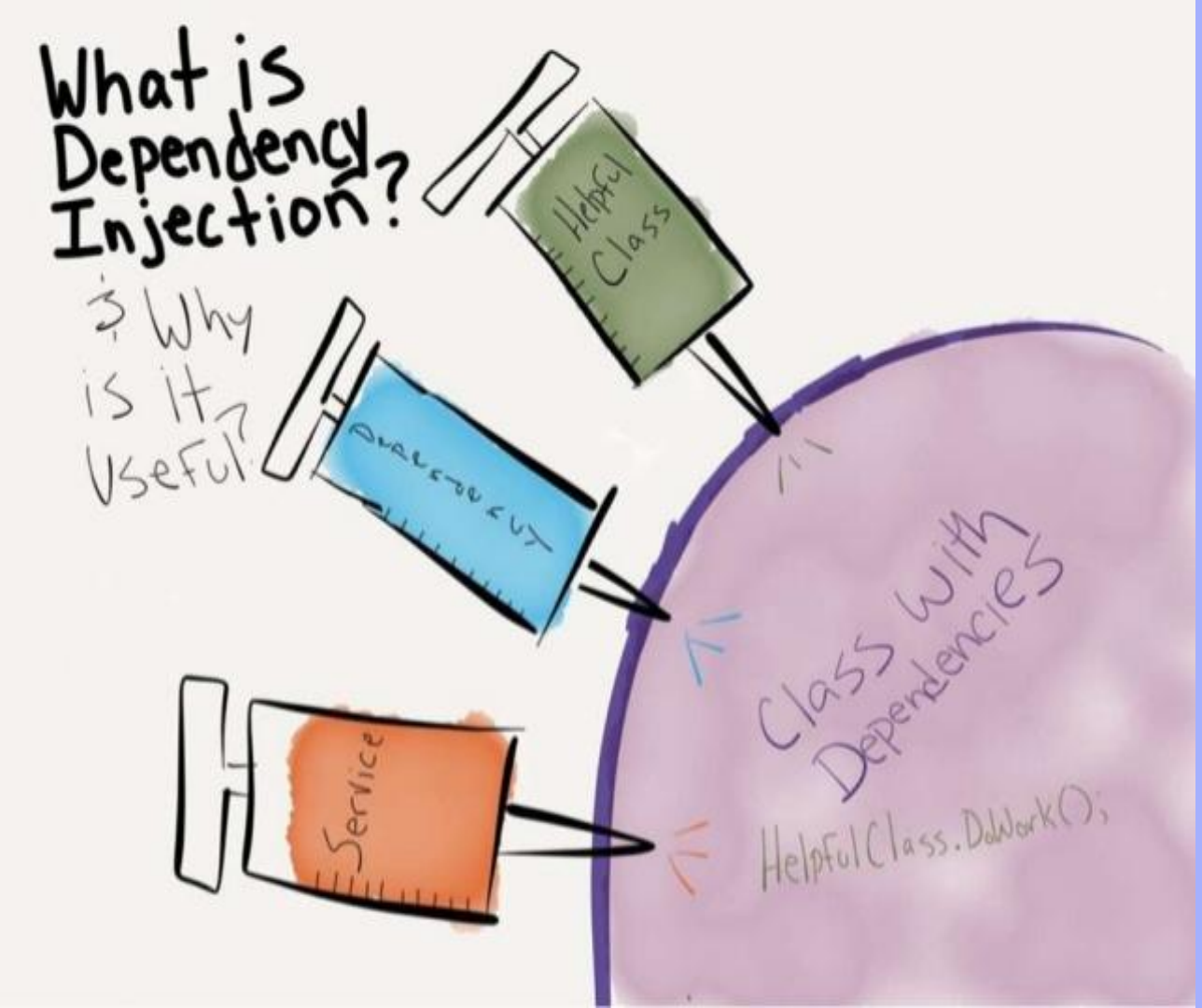




1. Dependency Injection (DI)

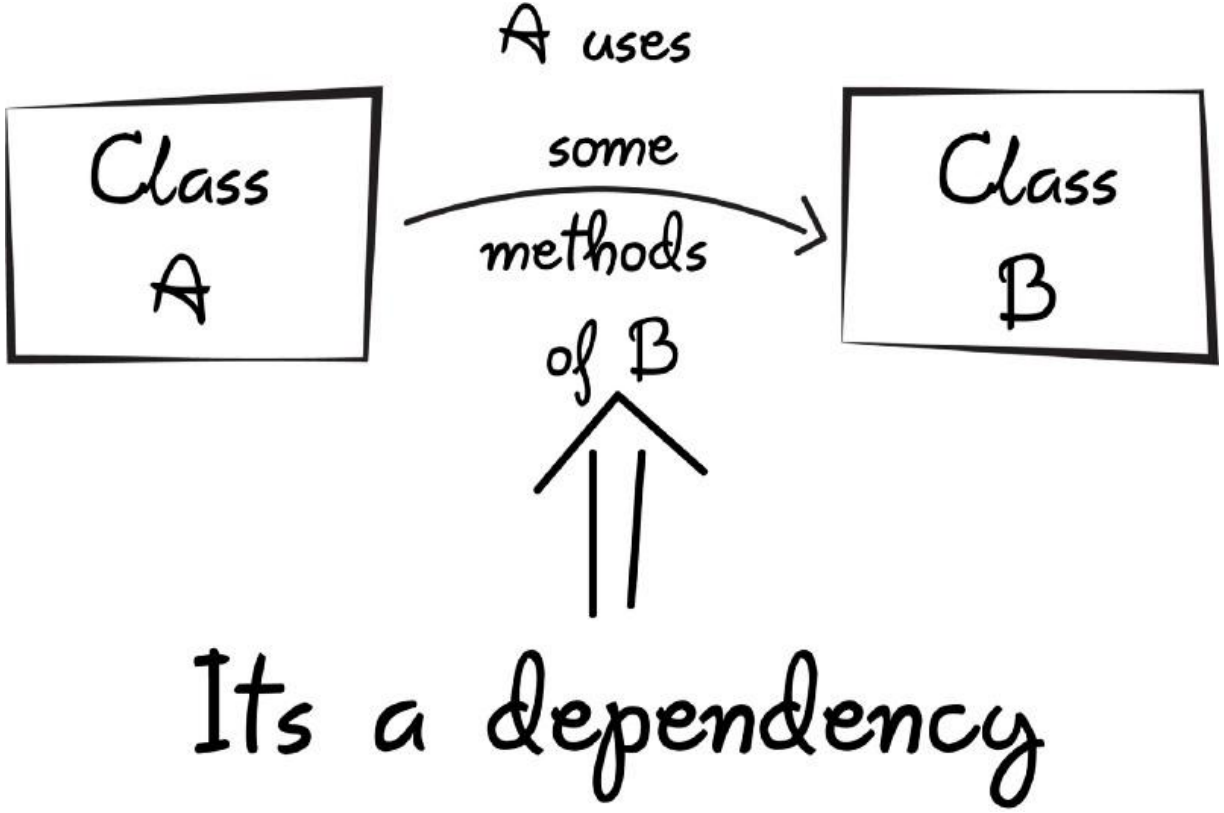
Initial Introduction





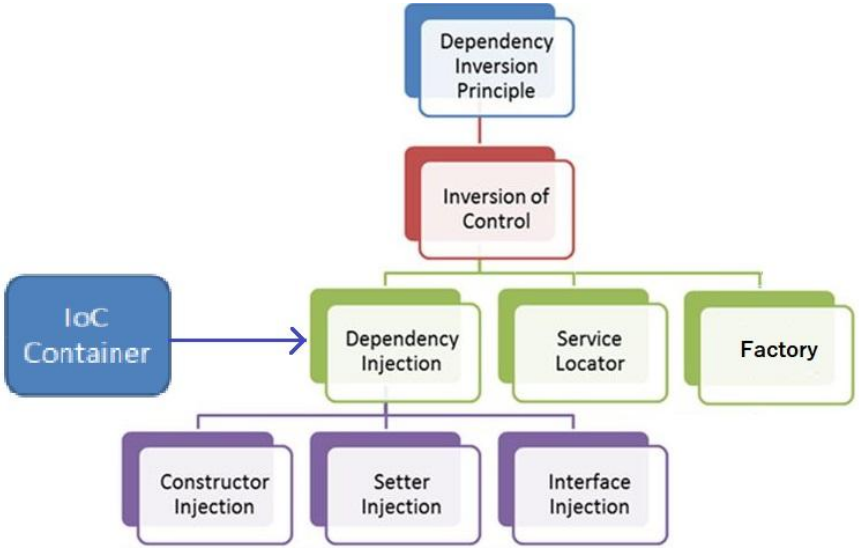
What?

In software engineering, dependency injection **is** a technique in which an object receives other objects that it depends on.



Why?

This enable an object to access another object's functions and properties.

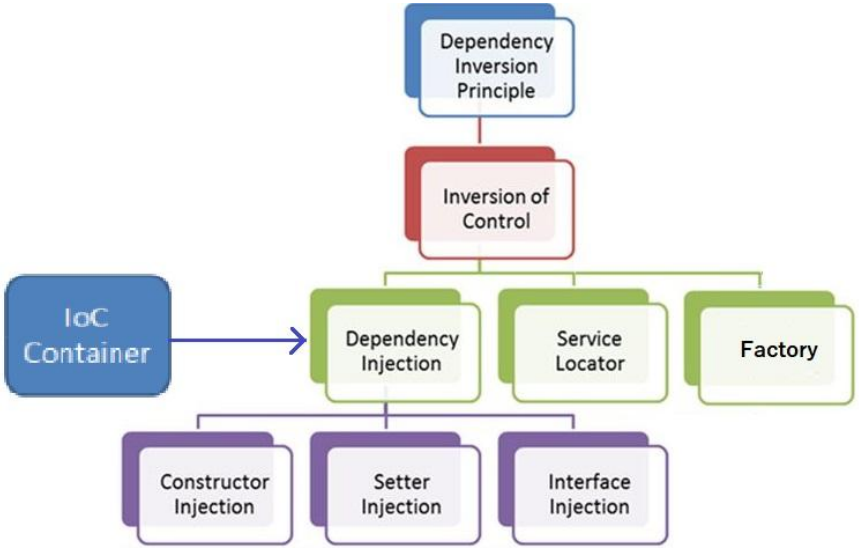


But How?

In order to be able to apply dependency injection, it is first necessary to apply tools that deal with dependency resolution.

DI's Tools: the most famous are:

- constructor Injection
- Property Injection
- Method Injection



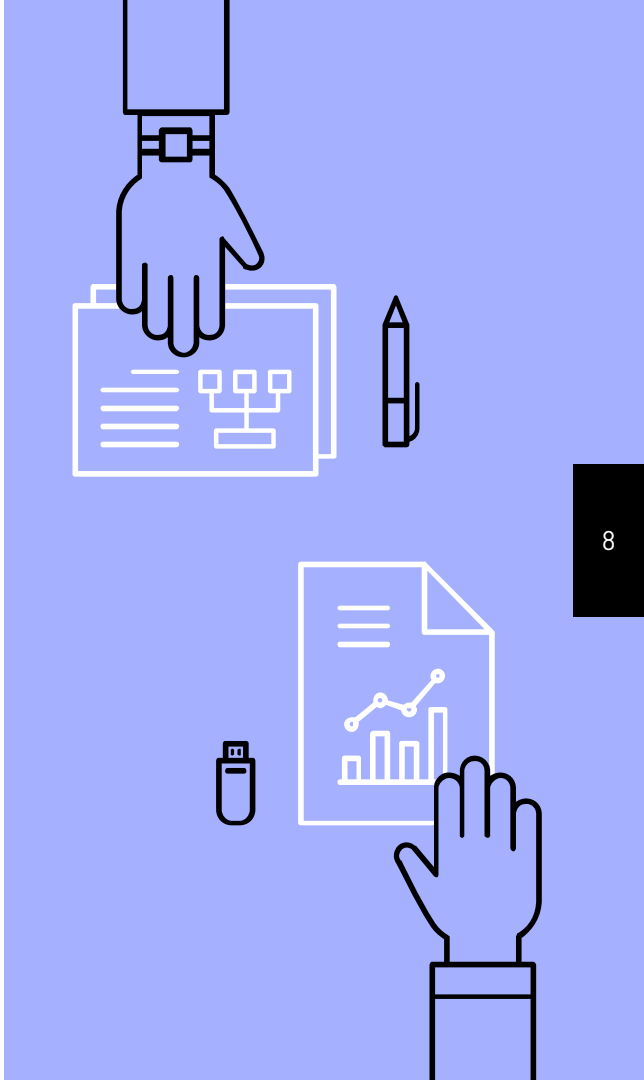
But How?

Containers: After we made our DI estructure we need to use it in different places or layers of our application. For that we could use containers, for example IoC.

Which?

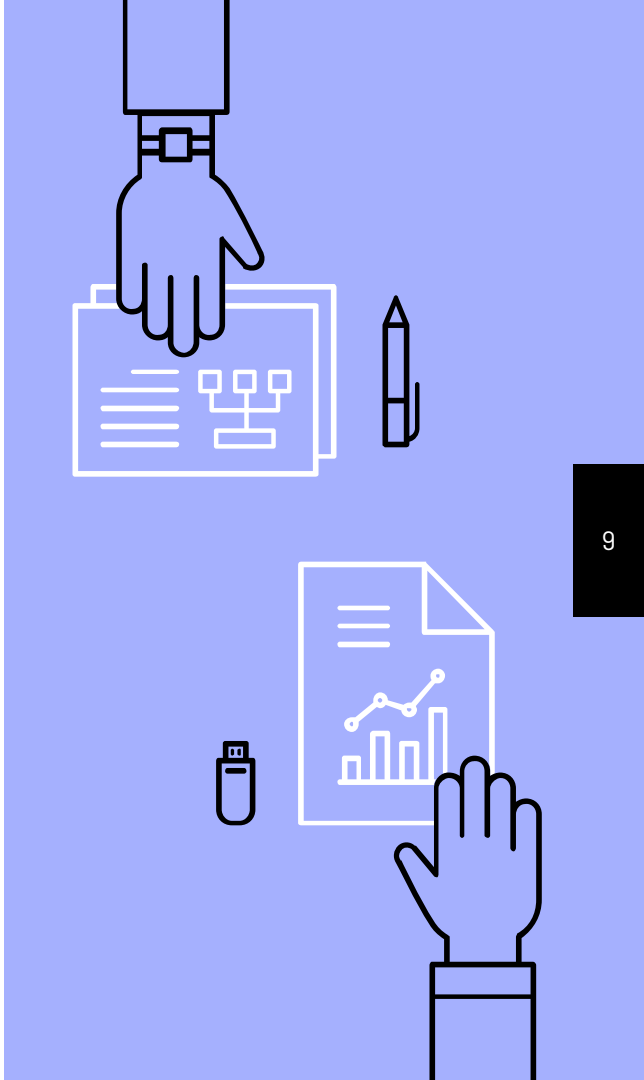
It depends. DI is an architectural pattern, so its implementation and tools will depend on the objectives of the project. But there are some

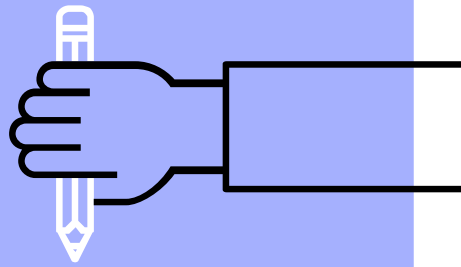
good practices that we should try to follow which we gonna check in this presentation.



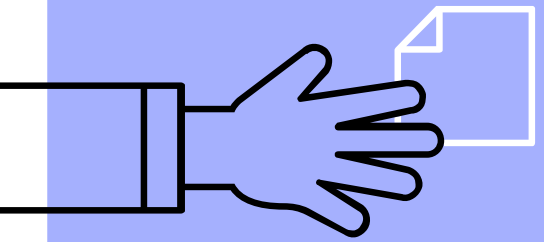
When?

DI is generally recommended for classes which contain methods performing operations with side effects, such as writing to a database. In that case you want to know what the dependencies are so you are not surprised when a method work suddenly wipes your entire database.



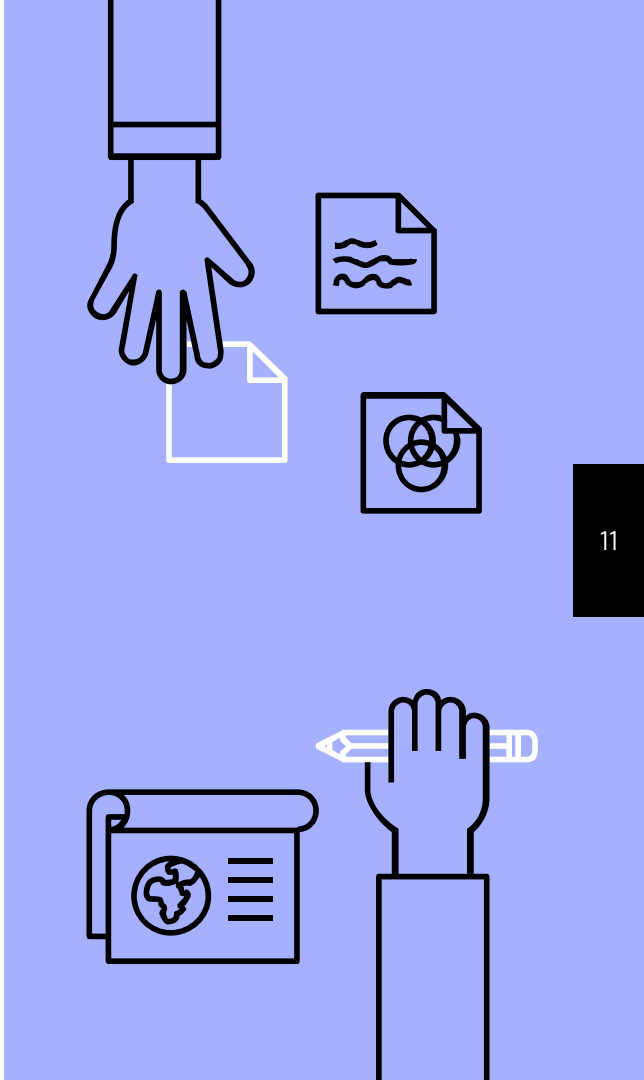


2. Anti-Pattern



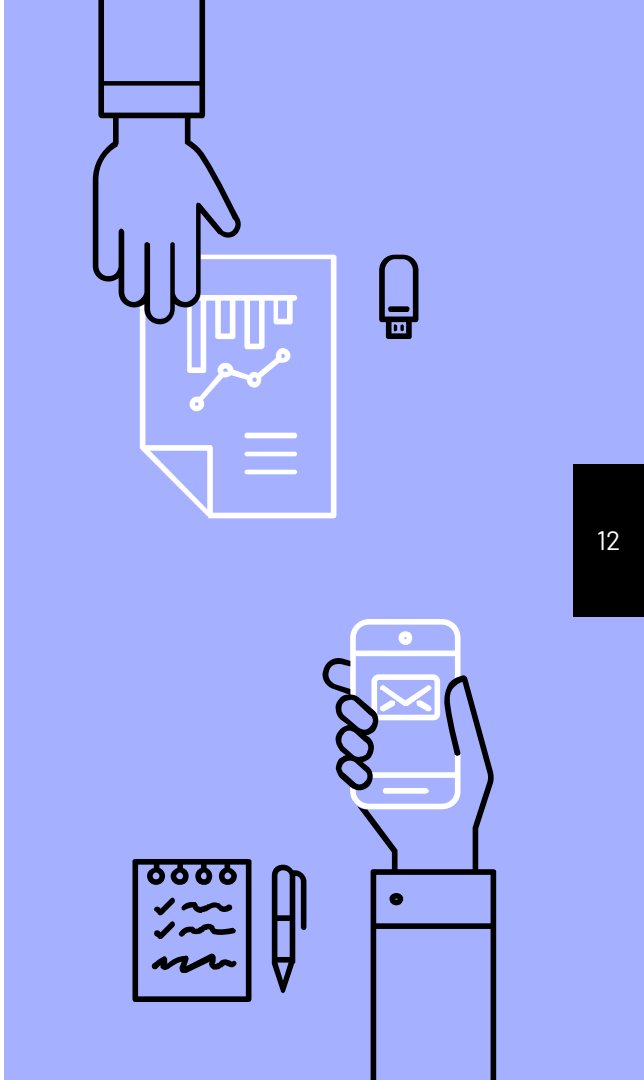
Others patterns

There are other patterns for resolution of dependencies. However in nowdays we used them to specific situations. Because diferente from DI, they dont handle well multiple dependencies in the same class.



Lets start with a problem

Problem: Make a international e-mail!



```
1 reference
public InternationalEmailService()
{
    this.configs = config.GetSection("Email").Get<EmailConfigurations>();
    this.translater = new EnglishTranslaterService();
}

1 reference
public async Task Send(string subject, string message, bool isBodyHtml = false)
{
    message = await translater.Translate(message);

    MailMessage msg = new MailMessage
```

```
var email = new InternationalEmailService();

await email.Send("Email internacional", "Bom dia! Como vai você?");
```

Without constructor

It is a class without any other restrictions. It is easier for the unit test. However since it encapsulates the creation of its dependencies, then we can't translate to another language, there is no flexibility!

```
2 references
public TranslaterService translater;

1 reference
public InternationalEmailService(IConfiguration config, TranslaterService translater)
{
    this.configs = config.GetSection("Email").Get<EmailConfigurations>();
    this.translater = translater;
}

1 reference
public async Task Send(string subject, string message, bool isBodyHtml = false)
{
    message = await translater.Translate(message);

    MailMessage msg = new MailMessage
```

```
var translater  = new TranslaterService("it");
var email = new InternationalEmailService(config, translater);

await email.Send("Email internacional", "Bom dia! Como vai você?");
```

Constructor by hand

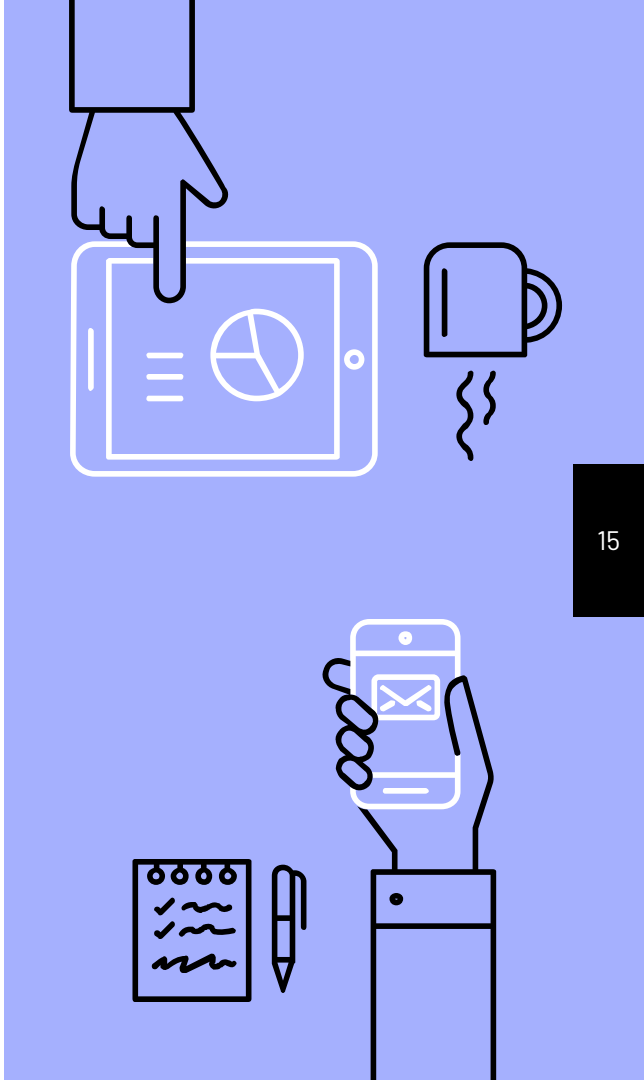
Now we don't encapsulate the creation of dependencies, so we can construct our own international email for differents languages.

Construction by Hand Review

This technique is also called construction injection and has the advantage of being explicit about its contract, that's why it definitely helps with testing.

disadvantages: the most obvious it is necessary knowing of the all object graphs. If you use the same object in many places, you must repeat code for wiring objects in all of those places. Fixing even a small bug can mean changing vast amount of code.

The grave problem is the fact that users need to know how object graphs are wird internally. This violates the principle of encapsulation and becomes problematic when dealing with code that is used by many clients, who really should not have to care about the internals of their dependencies in order to use them.



```
2 references
public TranslaterService translater;

1 reference
public InternationalEmailService(IConfiguration config, TranslaterService translater)
{
    this.configs = config.GetSection("Email").Get<EmailConfigurations>();
    this.translater = translater;
}

1 reference
public async Task Send(string subject, string message, bool isBodyHtml = false)
{
    message = await translater.Translate(message);

    MailMessage msg = new MailMessage
```

```
var factory = new LanguageFactory();
var translater  = factory.GetInstance("Dutch");

var email = new InternationalEmailService(config, translater);

await email.Send("Email internacional", "Bom dia! Como vai você?");
```

Factory Pattern

Factory Method is a Design Pattern which defines an interface for creating an object but lets the classes that implement the interface decide which class to instantiate.

```
1 reference
public class LanguageFactory
{
    1 reference
    public TranslaterService GetInstance(string language)
    {
        switch (language)
        {
            case "English":
                return new TranslaterService("en");
            case "Portuguese":
                return new TranslaterService("pt");
            case "Italy":
                return new TranslaterService("it");
            case "Dutch":
                return new TranslaterService("nl");
            case "Korean":
                return new TranslaterService("ko");
            case "Spanish":
                return new TranslaterService("sp");
            case "Afrikaans":
                return new TranslaterService("af");
            case "Russian":
                return new TranslaterService("ru");
            case "Romanian":
                return new TranslaterService("ro");
            case "Vietnamese":
                return new TranslaterService("vi");
            case "Filipino":
```

Factory Pattern

The idea behind the Factory pattern is to afflowad the burden of creating dependencies to a third-party object called a Factory.

Factory Pattern Review

It allows loose-coupling. It's called a factory because it creates various types of objects without necessarily knowing what kind of object it creates or how to create it. That's why

it is used when we have a superclass with multiple sub-classes and based on the input, we need to return one of the sub-class.

The Factory Method pattern is generally used in the following situations:

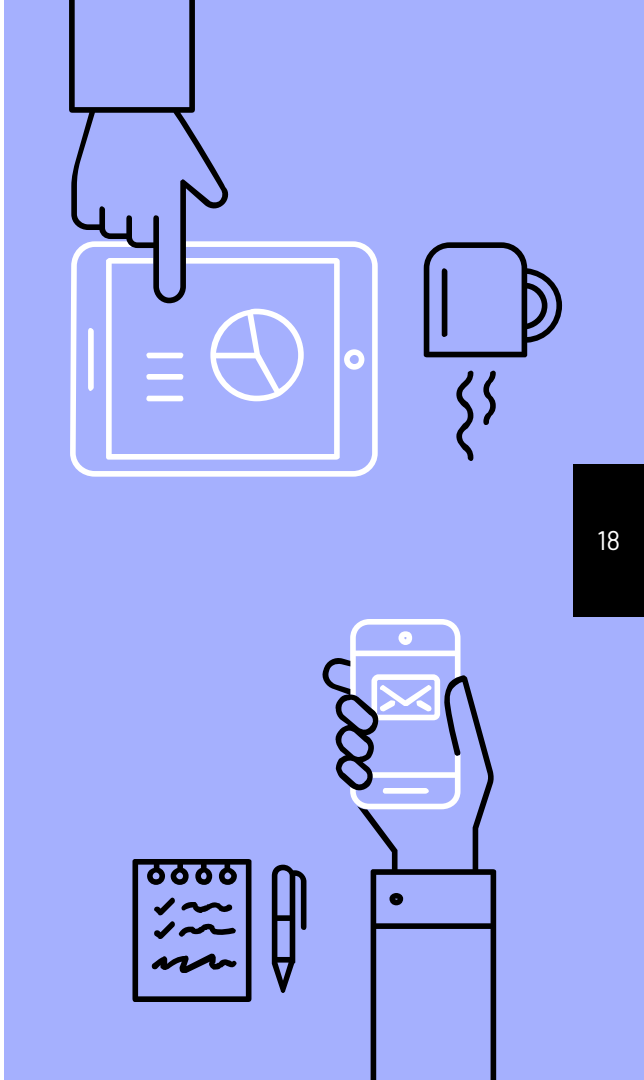
A class cannot anticipate the type of objects it needs to create beforehand.

A class requires its subclasses to specify the objects it creates.

You want to localize the logic to instantiate a complex object.

But, it makes code more difficult to read as all of your code is behind an abstraction that may in turn hide abstractions.

Can be classed as an anti-pattern when it is incorrectly used, for example some people use it to wire up a whole application when using an IOC container, instead use Dependency Injection



```
public class TranslaterFactory
{
    public ITranslater GetInstance(string translater)
    {
        switch (translater)
        {
            case "GoogleApi":
                return new GoogleApiService();
            case "Yandex":
                return new YandexTranslaterService();
            case "Dapplo":
                return new DapploService();
            default:
                return new CustomTranslaterService();
        }
    }
}
```

```
var factory = new TranslaterFactory();
var translater = factory.GetInstance("GoogleApi");

var msg = translater.GetHelloWorld("pt");

Console.WriteLine(msg);
```

Service Locator

A service locator pattern is a kind of Factory. It is a third-party object responsible for producing a fully constructed object graph. For example, if further we also want choose which whole translator API use, we would have a Service locator for that.

```
1 reference
public class TranslaterServiceLocator : IService
{

    6 references
    public Dictionary<object, object> servicecontainer = null;
    1 reference
    public ServiceLocator()
    {
        servicecontainer = new Dictionary<object, object>();
        servicecontainer.Add(typeof(ITranslater), new GoogleApiService());
        servicecontainer.Add(typeof(ITranslater), new YandexTranslaterService());
        servicecontainer.Add(typeof(ITranslater), new DapploService());
        servicecontainer.Add(typeof(ITranslater), new DapploService());
    }
    1 reference
    public ITranslater GetService<ITranslater>()
    {
        try
        {
            return (ITranslater)servicecontainer[typeof(ITranslater)];
        }
        catch (Exception ex)
        {
            throw new NotImplementedException("Service not available.");
        }
    }
}
```

Service Locator

Another popular implementation

```
var serviceLocator = new TranslaterServiceLocator();
ITranslater googleTranslater = serviceLocator.GetService<GoogleApiService>();
googleTranslater.SetDefaultLanguage("Spanish");

var email = new InternationalEmailService(config, googleTranslater);

await email.Send("Email internacional", "Bom dia! Como vai você?");
```

```
1 reference
public class InternationalEmailService
{
    9 references
    private readonly EmailConfigurations configs;
    2 references
    public ITranslater translater;

    1 reference
    public InternationalEmailService(IConfiguration config, ITranslater translater)
    {
        this.configs = config.GetSection("Email").Get<EmailConfigurations>();
        this.translater = translater;
    }

    1 reference
    public async Task Send(string subject, string message, bool isBodyHtml = false)
    {
        message = await translater.Translate(message);

        MailMessage msg = new MailMessage
```

Service Locator

P.S: Since there is only a single instance of the Locator there can only be one resolution and because you are REQUESTING a service, it makes it impossible to have two different behaviors for the same code executing simultaneously.

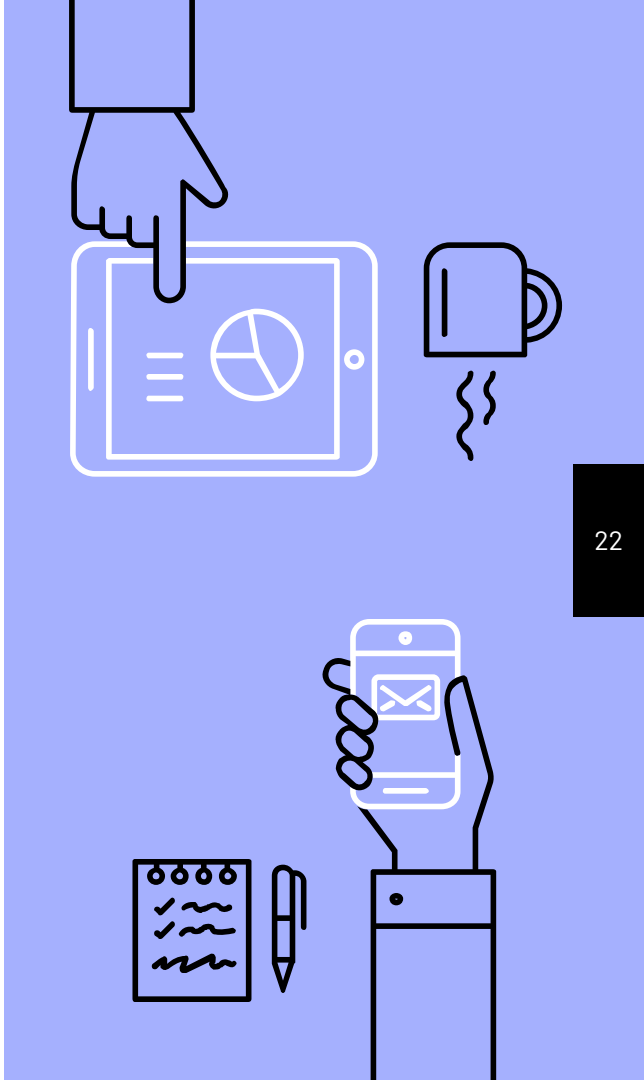
Service Locator Pattern Review

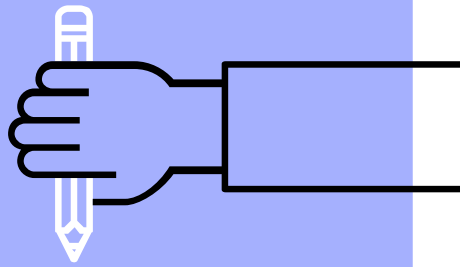
This is significantly different from a Factory that produces only one kind of service.A service Locator is, therefore, a Factory that can produce any kind of service. Right away this helps reduce a huge amount of repetitive Factory code, in factory that can produce any kind of service.

However, being a kind of Factory, Service Locators suffer from the same problems of testability and shared state.

If a key is bound improperly, the wrong type of object may be created, and this error is found out only at runtime.

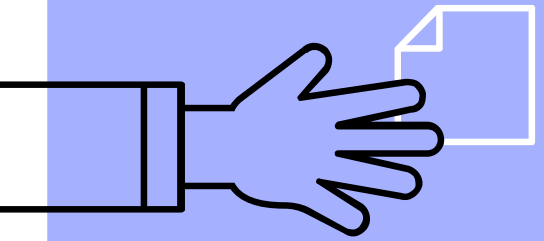
Not so great for APIs: Because dependencies are hidden inside your class and because they are specified at runtime, formal knowledge of the dependencies is required. Again though, for simple or internal projects this is acceptable.



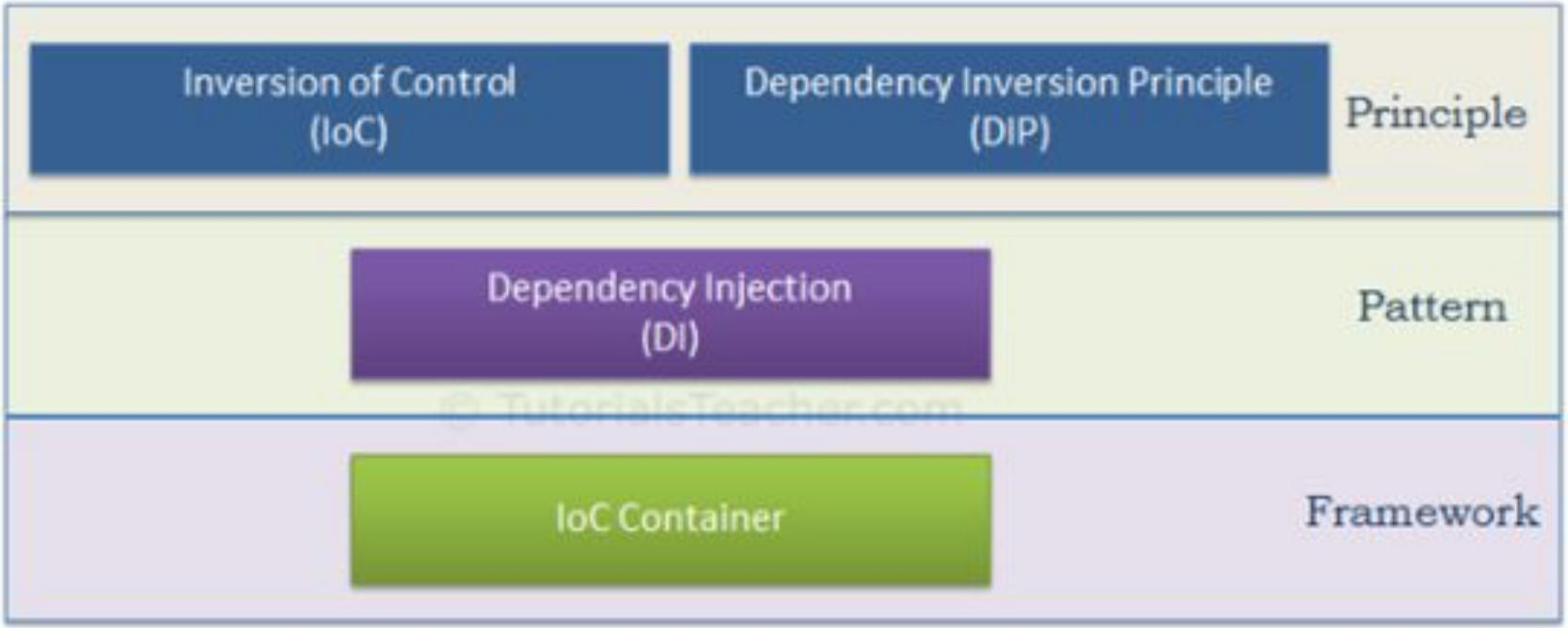


3. Patterns

Tools we should use



Concepts


Inversion of Control
(IoC)
Dependency Inversion Principle
(DIP)
Principle
Dependency Injection
(DI)
Pattern
IoC Container
Framework

Principle Vs Pattern

Principle

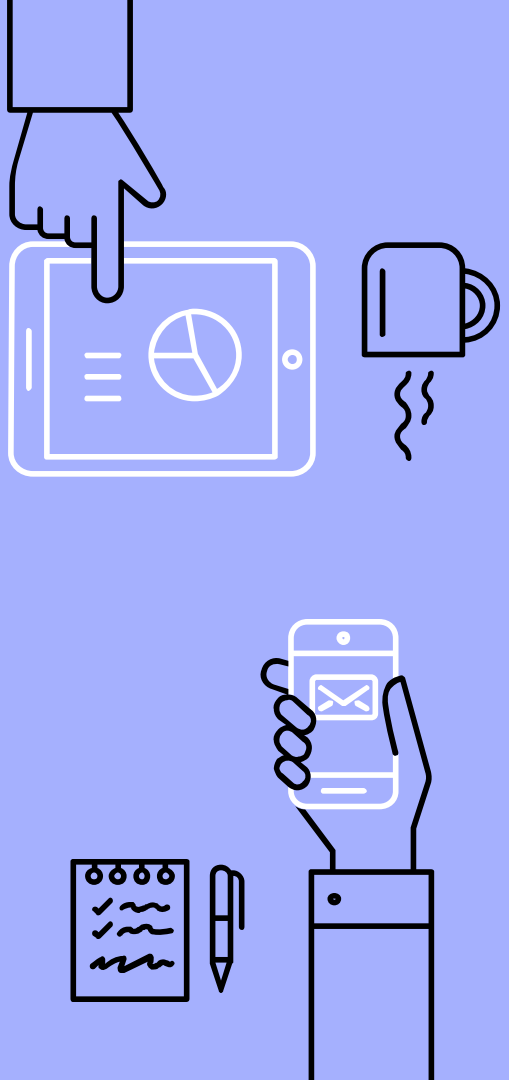
Design principles provide high level guidelines to design better software applications. They do not provide implementation guidelines and are not bound to any programming language. Ex: SOLID.

Pattern

Design Pattern provides low-level solutions related to implementation, of commonly occurring object-oriented problems. In other words, design pattern suggests a specific implementation for the specific object-oriented programming problem.

Framework

Framework is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software.



1. Dependency inversion Principle

Dependence not inverted: High level depends on low level interface.

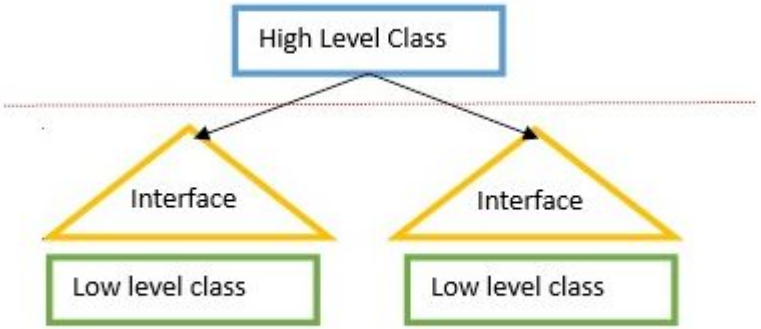


fig: When Dependency was not inverted

Dependency inversion: Higher level class defines the interface and higher level class doesn't depend on lower level class directly.

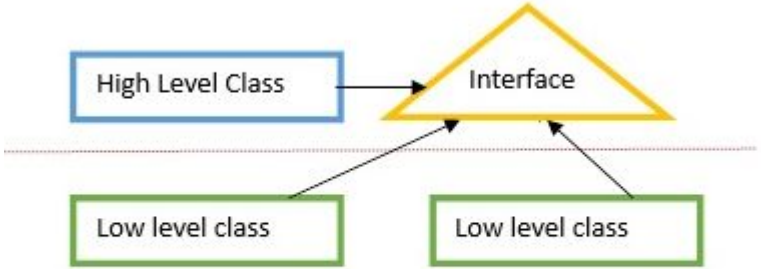
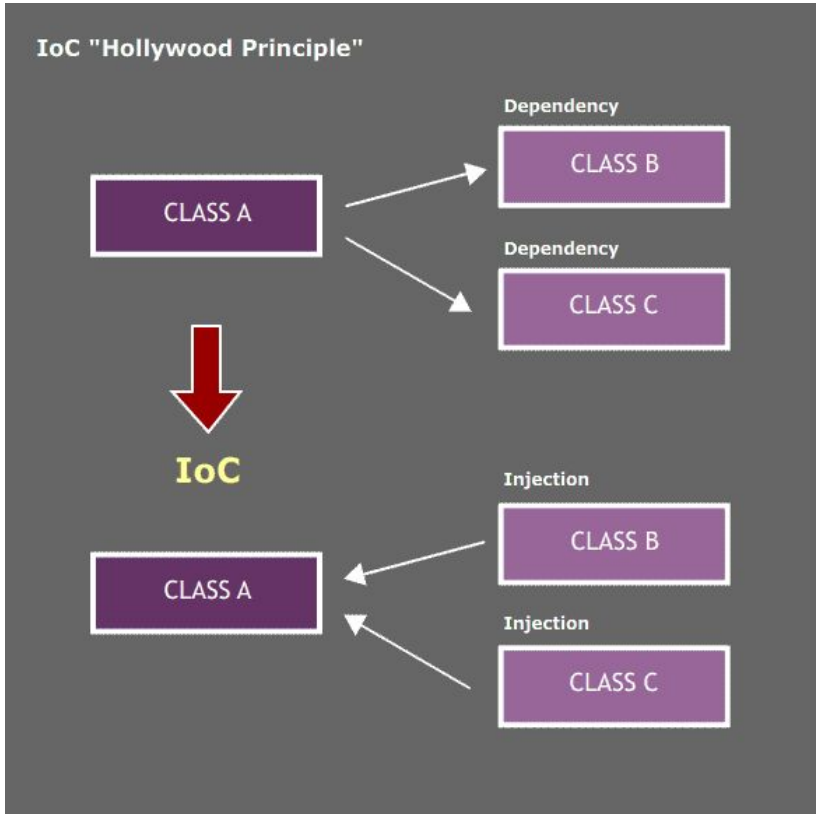
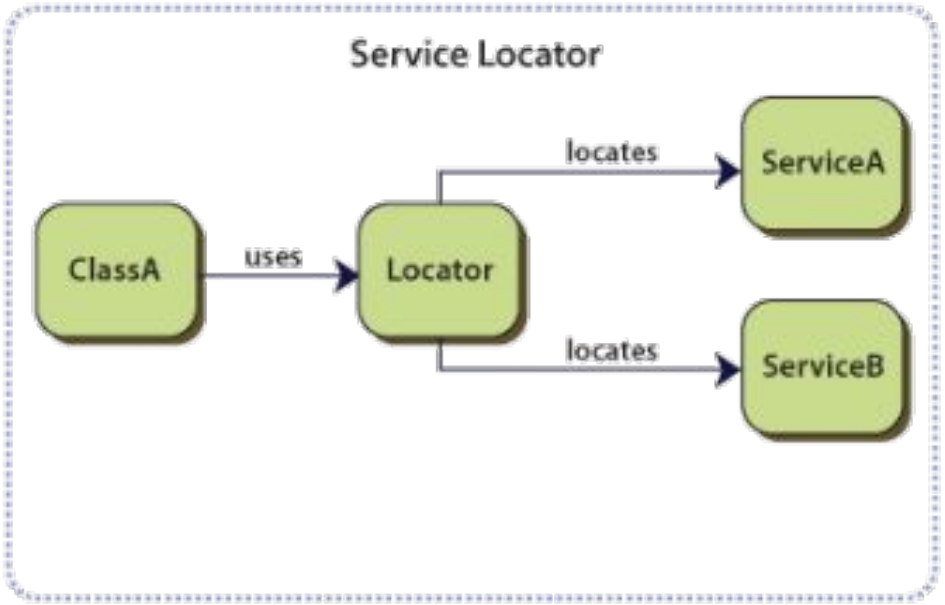


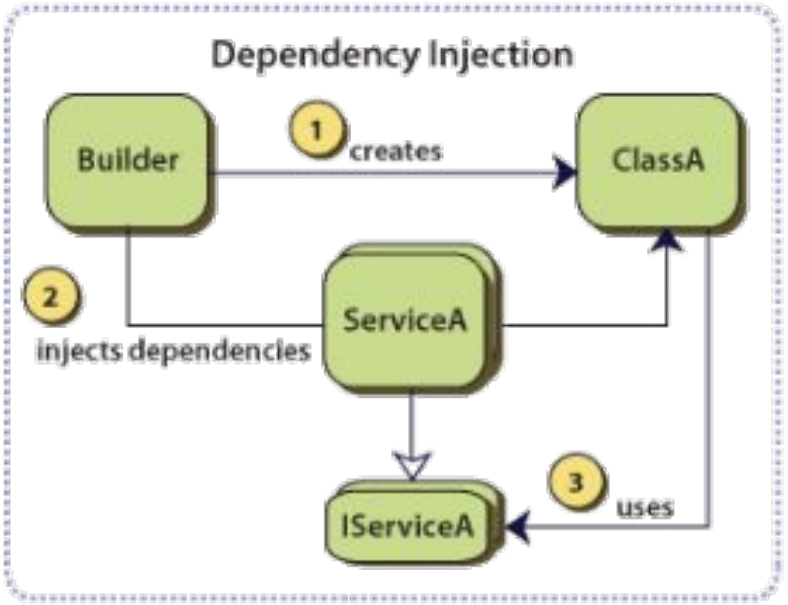
fig: Interface was defined by high level class

2. Inversion of Control (IoC)



3. Dependency Injection





```
1 reference
public class InternationalEmailService
{
    9 references
    private readonly EmailConfigurations configs;
    2 references
    public ITranslater translater;

    1 reference
    public InternationalEmailService(IConfiguration config, ITranslater translater)
    {
        this.configs = config.GetSection("Email").Get<EmailConfigurations>();
        this.translater = translater;
    }

    1 reference
    public async Task Send(string subject, string message, string language)
    {
        message = await translater.Translate(message, language);

        MailMessage msg = new MailMessage
```

```
ITranslater googleTranslater = new GoogleApiService();
var email = new InternationalEmailService(config, googleTranslater);

await email.Send("Email internacional", "Bom dia! Como vai você?", "Spanish");
```

Constructor Injection

Dependency Injection is done by supplying the DEPENDENCY through the class's constructor when creating the instance of that class.

```
1 reference
public class InternationalEmailService
{
    9 references
    private readonly EmailConfigurations configs;
    2 references
    private ITranslater translater;
    1 reference
    public ITranslater Translater { set { this.translater = value; } }

    1 reference
    public InternationalEmailService(IConfiguration config)
    {
        this.configs = config.GetSection("Email").Get<EmailConfigurations>();
    }

    1 reference
    public async Task<string> Translate(string text, string language)
    {
        return await this.translater.Translate(text, language);
    }

    1 reference
    public async Task Send(string subject, string message, string language)
    {
        message = await this.Translate(message, language);

        MailMessage msg = new MailMessage
        {
```

Property Injection

Recommended using when a class has optional dependencies, or where the implementations may need to be swapped

```
var email = new InternationalEmailService(config);
email.Translater = new GoogleApiService();

await email.Send("Email internacional", "Bom dia! Como vai você?", "Spanish");
```

Property or Setter Injection

Recommended using when a class has optional dependencies, or where the implementations may need to be swapped

```
2 references
private ITranslater translater;

1 reference
public InternationalEmailService(IConfiguration config)
{
    this.configs = config.GetSection("Email").Get<EmailConfigurations>();
}

1 reference
public void SetTranslater(ITranslater translater)
{
    this.translater = translater;
}

1 reference
public async Task Send(string subject, string message, string language)
{
    message = await this.translater.Translate(message, language);

    MailMessage msg = new MailMessage
```

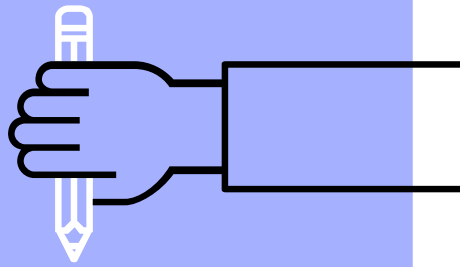
```
var email = new InternationalEmailService(config);
email.SetTranslater(new GoogleApiService());

await email.Send("Email internacional", "Bom dia! Como vai você?", "Spanish");
```

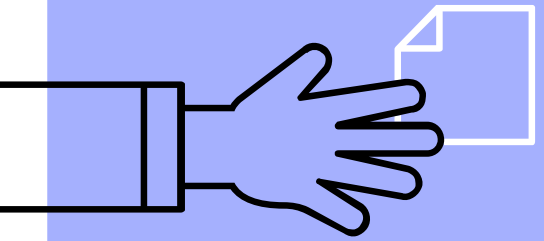
Method Injection

Inject the dependency into a single method and generally for the use of that method.

It could be useful, where the whole class does not need the dependency, only one method having that dependency.

DI Containers

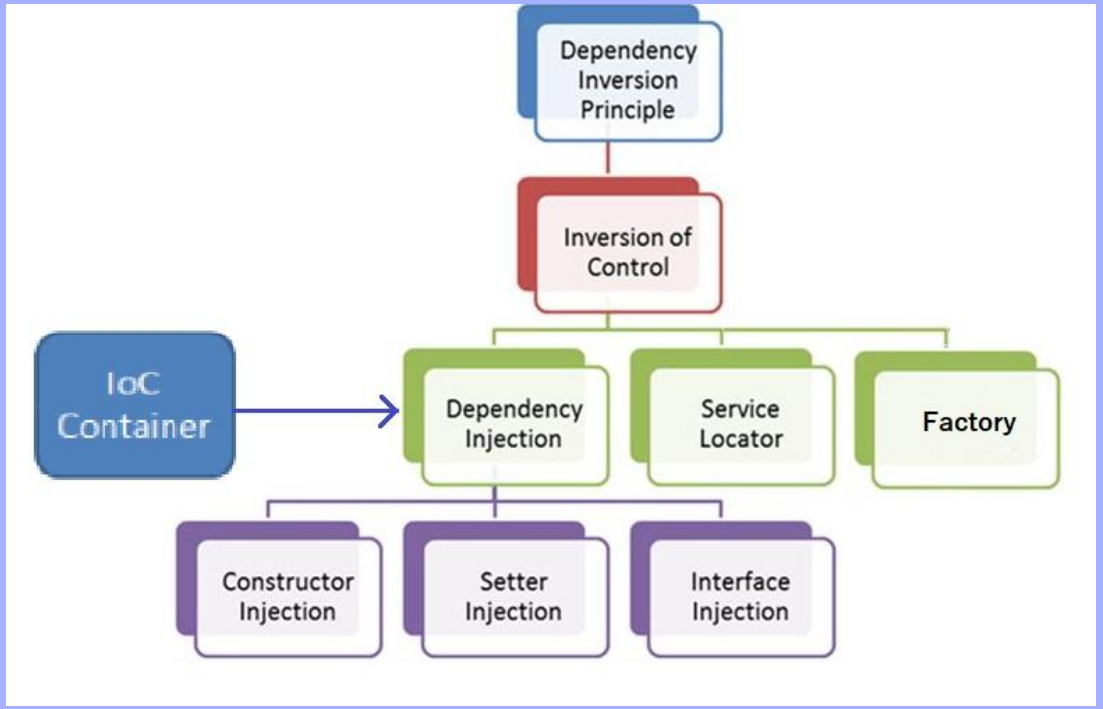




BIG

CONCEPT

The recommended way to implement DI is, you should use DI containers.



```
public static class NativeDotNetInjector
{
    1 reference
    public static void RegisterServices(IServiceCollection serviceCollection)
    {
        AddRepositories(serviceCollection);
        AddServices(serviceCollection);
        AddContextAdministrators(serviceCollection);
    }

    1 reference
    private static void AddServices(IServiceCollection serviceCollection)
    {
        serviceCollection.AddTransient<JwtTokenGeneratorService>();
        serviceCollection.AddTransient<IUserService, UserService>();
        serviceCollection.AddTransient<IPropertyService, PropertyService>();
        serviceCollection.AddTransient<IPropertyTypeService, PropertyTypeService>();
        serviceCollection.AddTransient<ICustomerService, CustomerService>();
        serviceCollection.AddTransient<IImageService, ImageService>();
        serviceCollection.AddTransient<IResourceService, ResourceService>();
        serviceCollection.AddTransient<IResourceTypeService, ResourceTypeService>();
        serviceCollection.AddTransient<IActionTypeService, ActionTypeService>();
        serviceCollection.AddTransient<IProfileService, ProfileService>();
        serviceCollection.AddTransient<IEmailService, EmailService>();
        serviceCollection.AddTransient<ICityService, CityService>();
        serviceCollection.AddTransient<IStateService, StateService>();
```

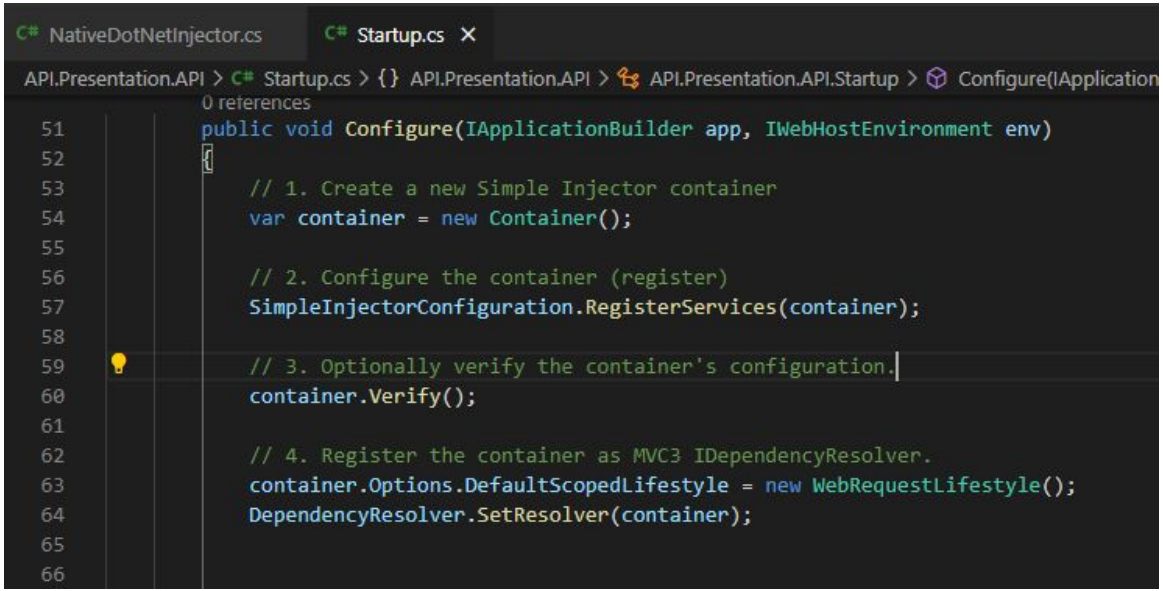
Native dotnet Injector

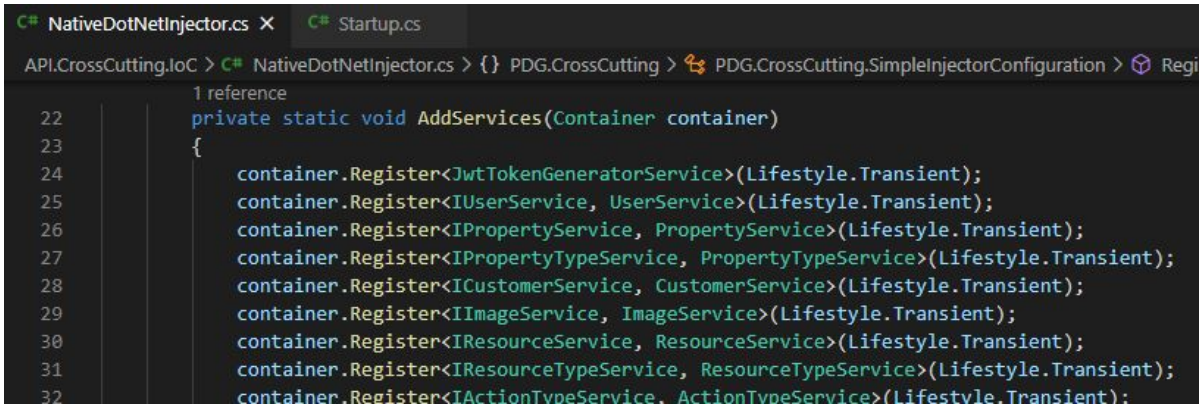
```
public override void Load()
{
    Bind<JwtTokenGeneratorService>();
    Bind<IUserService>().To<UserService>();
    Bind<IPropertyService>().To<PropertyService>();
    Bind<IPropertyTypeService>().To<PropertyTypeService>();
    Bind<ICustomerService>().To<CustomerService>();
    Bind<IImageService>().To<ImageService>();
    Bind<IResourceService>().To<ResourceService>();
    Bind<IResourceTypeService>().To<ResourceTypeService>();
    Bind<IActionTypeService>().To<ActionTypeService>();
    Bind<IProfileService>().To<ProfileService>();
    Bind<IEmailService>().To<EmailService>();
    Bind<ICityService>().To<CityService>();
    Bind<IStateService>().To<StateService>();
```

```
    Bind<NotificationContext>();
    Bind<IUnitOfWork>().To<UnitOfWork>().InSingletonScope();
}
```

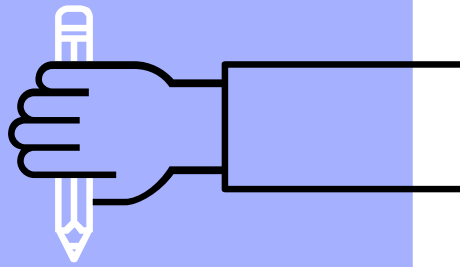
```
public void Configure (IApplicationBuilder app, IWebHostEnvironment env) {
    var kernel = CreateKernel ();

    var webApiConfiguration = new HttpConfiguration();
    webApiConfiguration.MapHttpAttributeRoutes();
    app.UseNinjectMiddleware(CreateKernel).UseNinjectWebApi(webApiConfiguration);
```

Ninject Injector



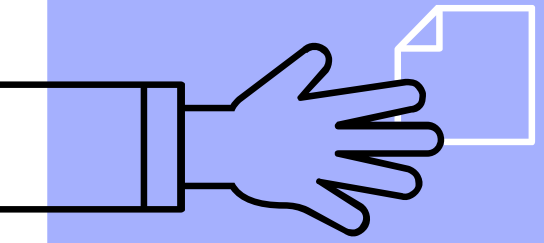


Simple Injector



LifeCycle

Lifetime of the objects



Life-Time

Trasient

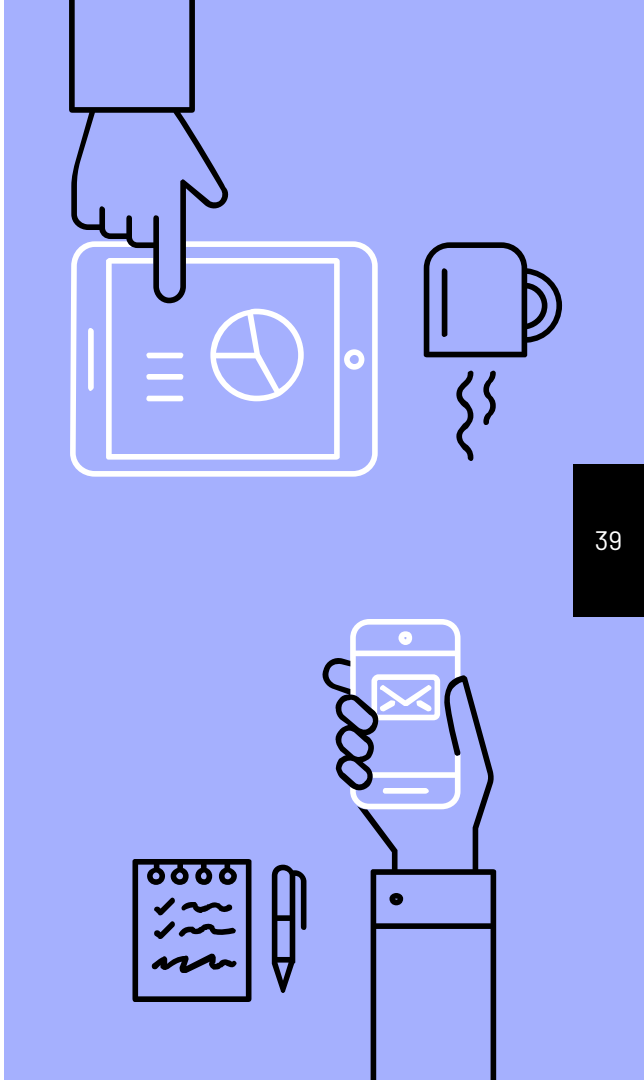
A new instance of the component will be created each time the service is requested from the container. If multiple consumers depend on the service within the same graph, each consumer will get its own new instance of the given service.

Scopped

For every request within an implicitly or explicitly defined scope.

Singleton

There will be at most one instance of the registered service type and the container will hold on to that instance until the container is disposed or goes out of scope. Clients will always receive that same instance from the container.



Good practices

Trasient:

Register your services as transient wherever possible. Because it's simple to design transient services. You generally don't care about multi-threading and memory leaks and you know the service has a short life

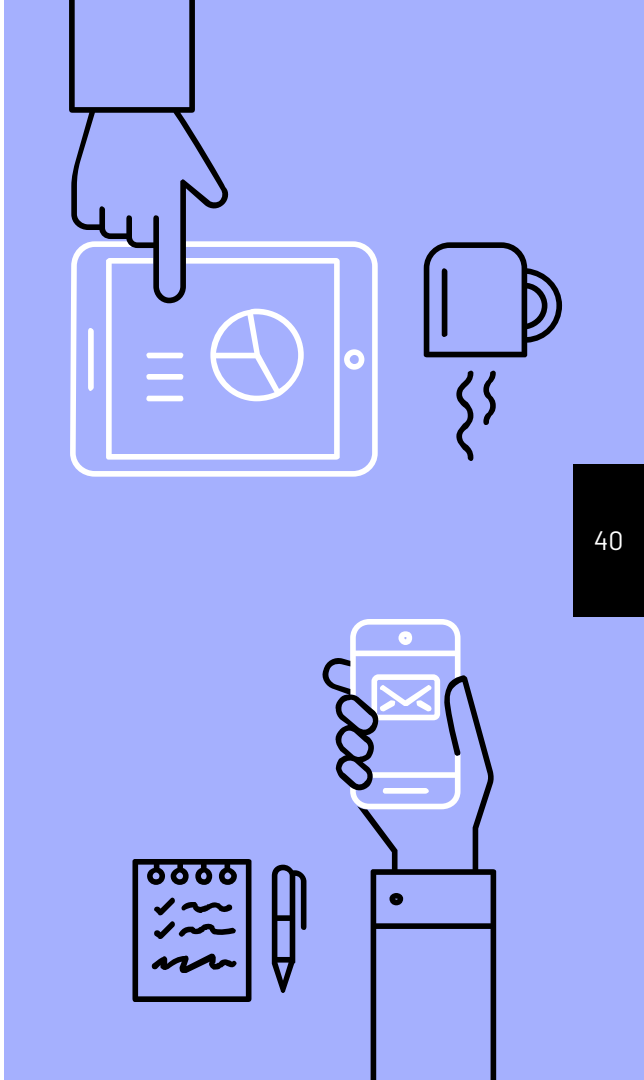
Scoped:

Duration of Scope. Asp.Net uses web request as scope Useful for **UnitOfWork** objects, e.g. Db context. See also ServiceScopeFactory;

Singleton:

Singleton services are generally designed to keep an application state. A **cache** is a good example of application states.

Use singleton lifetime carefully since then you need to deal with multi-threading and potential memory leak problems.

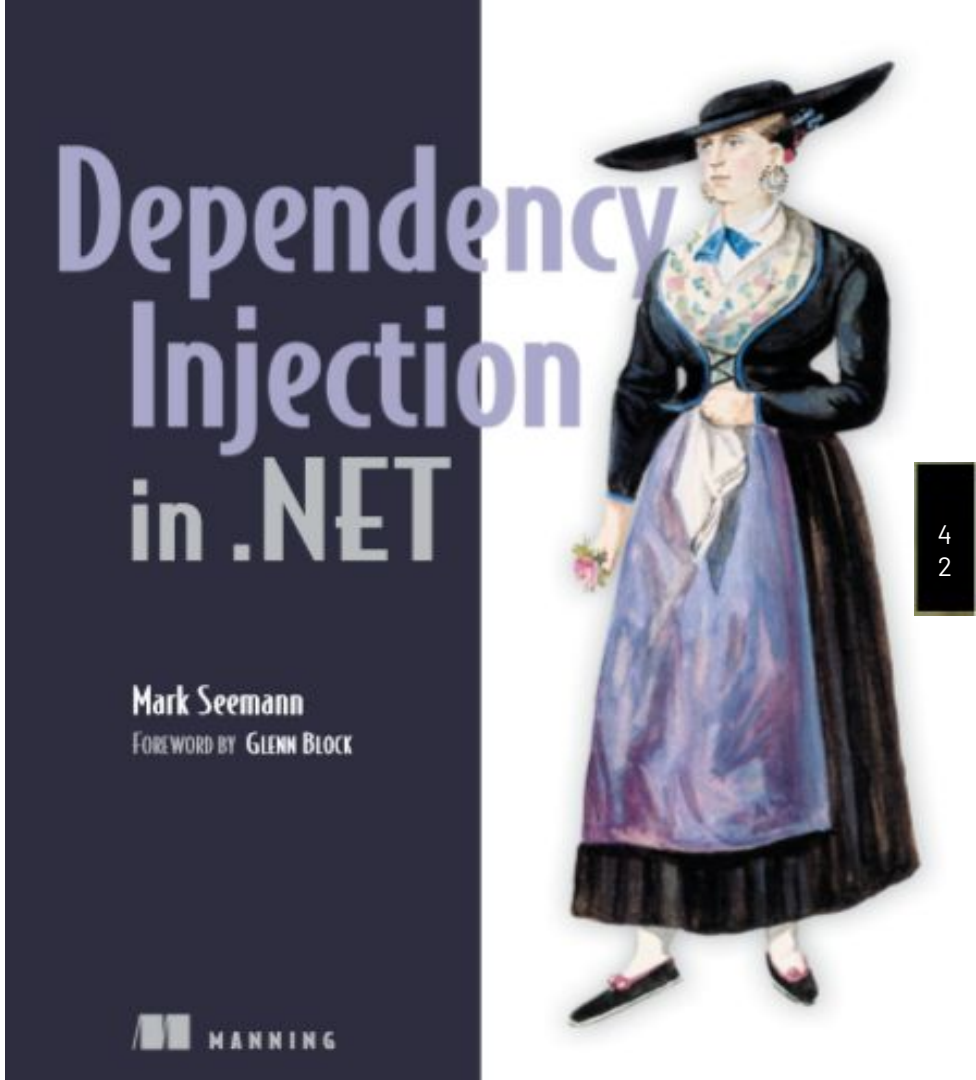


THANKS!

Any questions?

Inspired by

This presentation is based in the book “Dependency Injection in .NET” - Mark Seemann



Inspired by

"Dependency Injection"
- Dhanji R. Prasanna