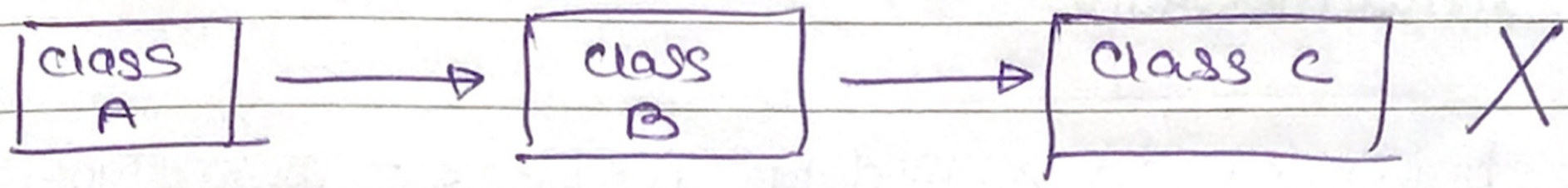
Dependency Injection

Dependency Inversion

Resolve 2 things ① Your High level modules should not depend on low level modules.



⇒ new B(); ⇒ new C();

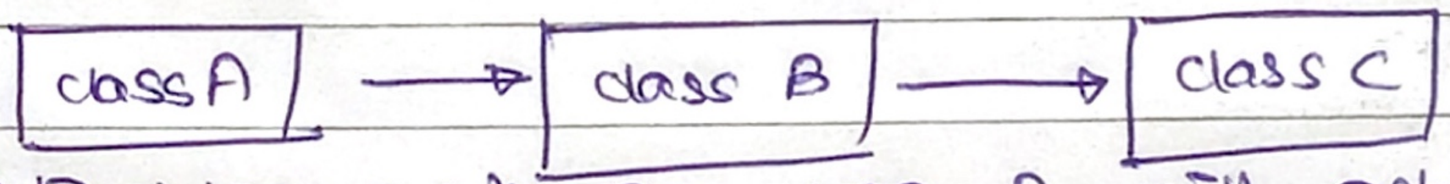
#) Here we should add interface & instantiate it.

#) This will resolve the dependency.

#) Problem : If class B changes class A will also get change & when class C changes class C will get change. ⇒ Solution ⇒ Add Interface & fix it.

Inversion of control

A framework controls which code is executed next, not your code.



#) Problem : When class A will get executed it will instantiate class B & class B will instantiate class C.

Resolution : There should be framework which will inject the dependency & will resolve it.

eg. ASP.net framework ⇒ The framework should decide which to execute.

Life Time Management

A) Scoped B) Transient C) Singleton

A) Scoped : A new instance is created once per scope and reused in the scope.

B) Transient : A new instance is created every time a type is requested.

Microsoft Dependecy Injection package ⇒ Microsoft Extension.

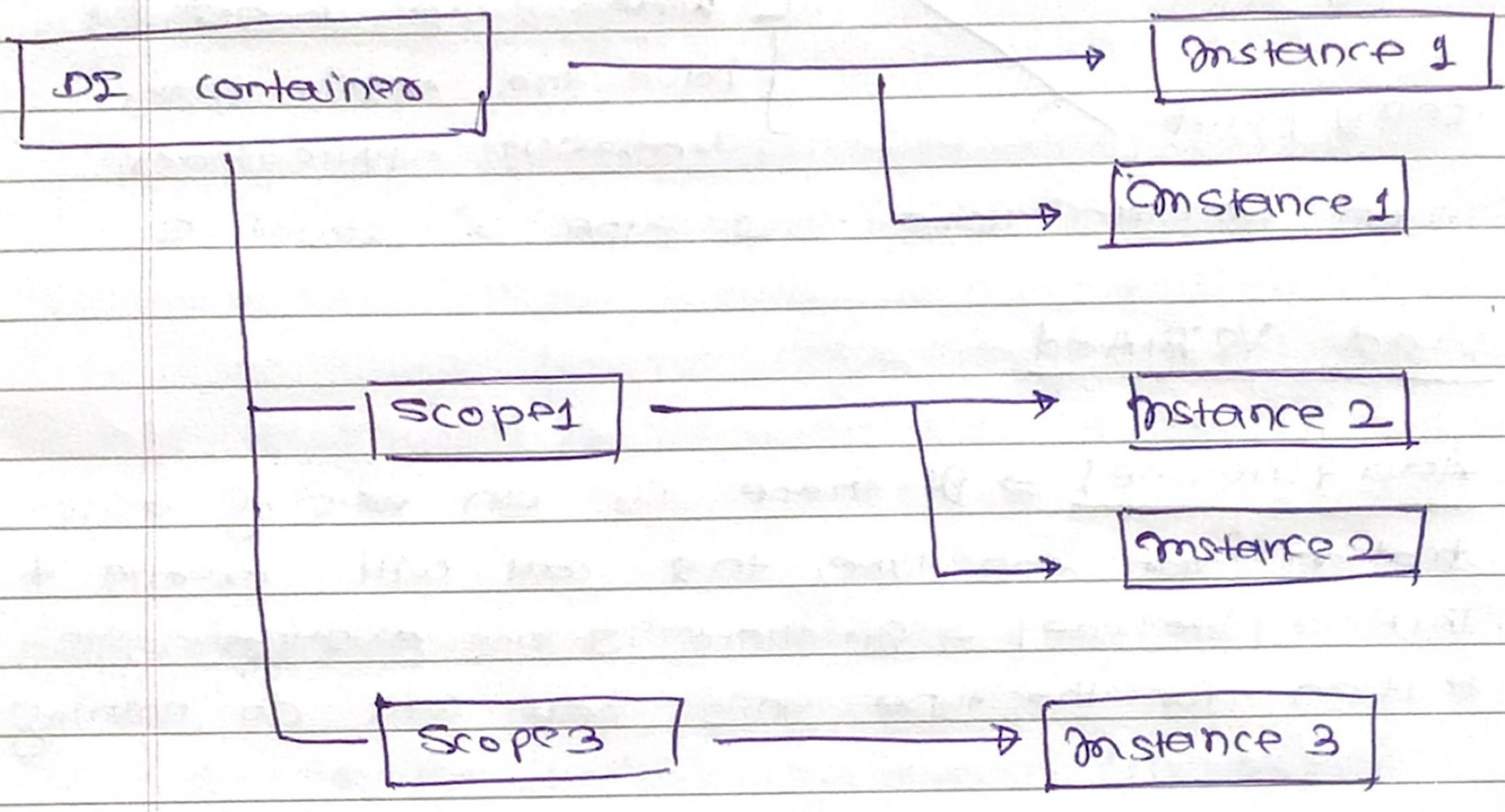
Hosting

Hosting ⇒ Microsoft.Extension.Dependency Injection.

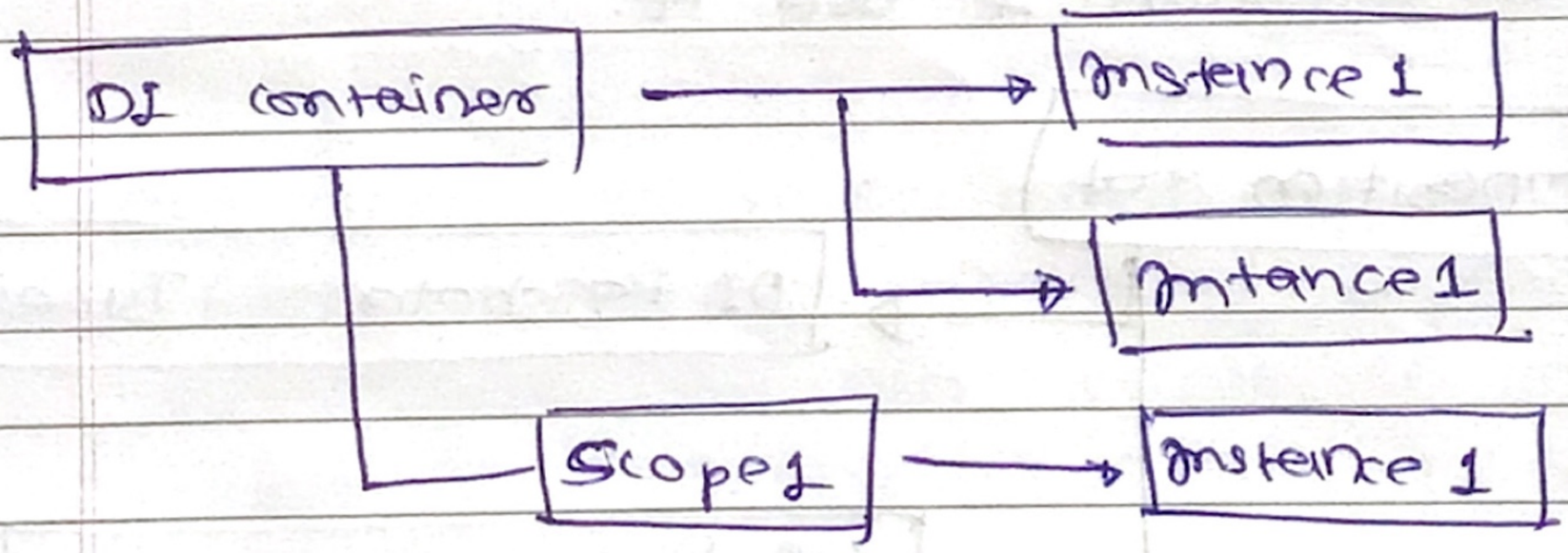
c) Scoped : A new instance is created once per scope, and then reused in the scope.

d) Singleton : A new instance is created once & reused from then onwards.

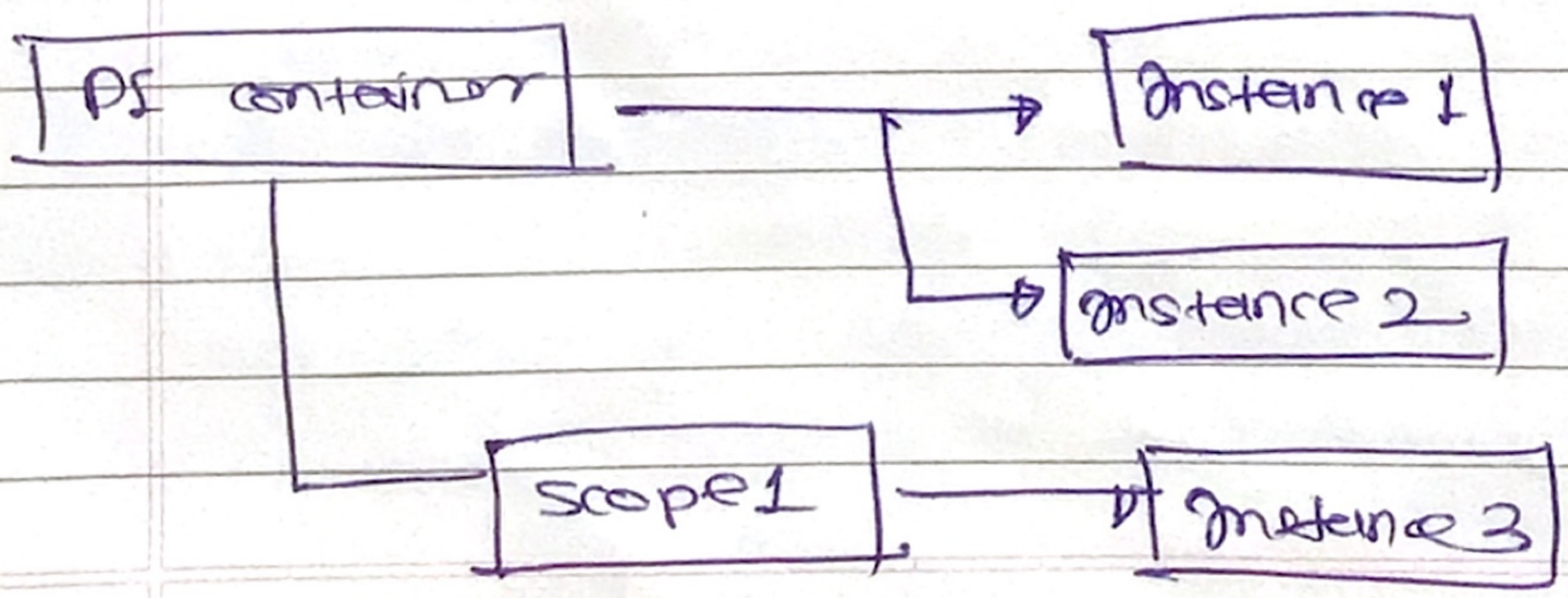
Scoped



Singleton



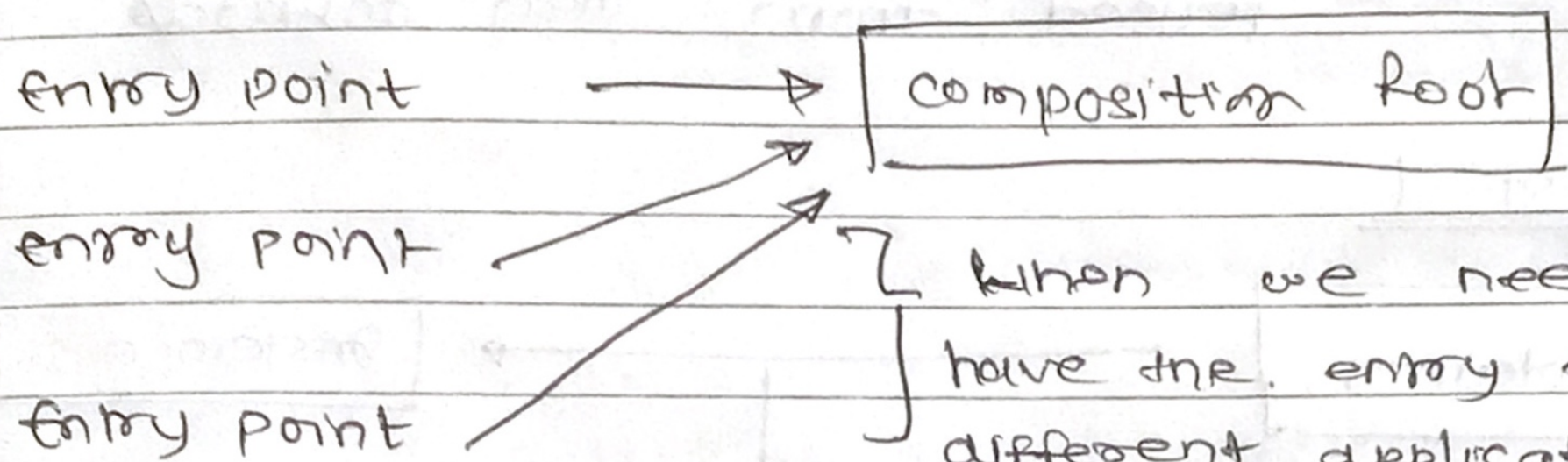
Transient



Page No.
Date

Composition Root >> entry point

>> The place where we keep the dependency Injection.



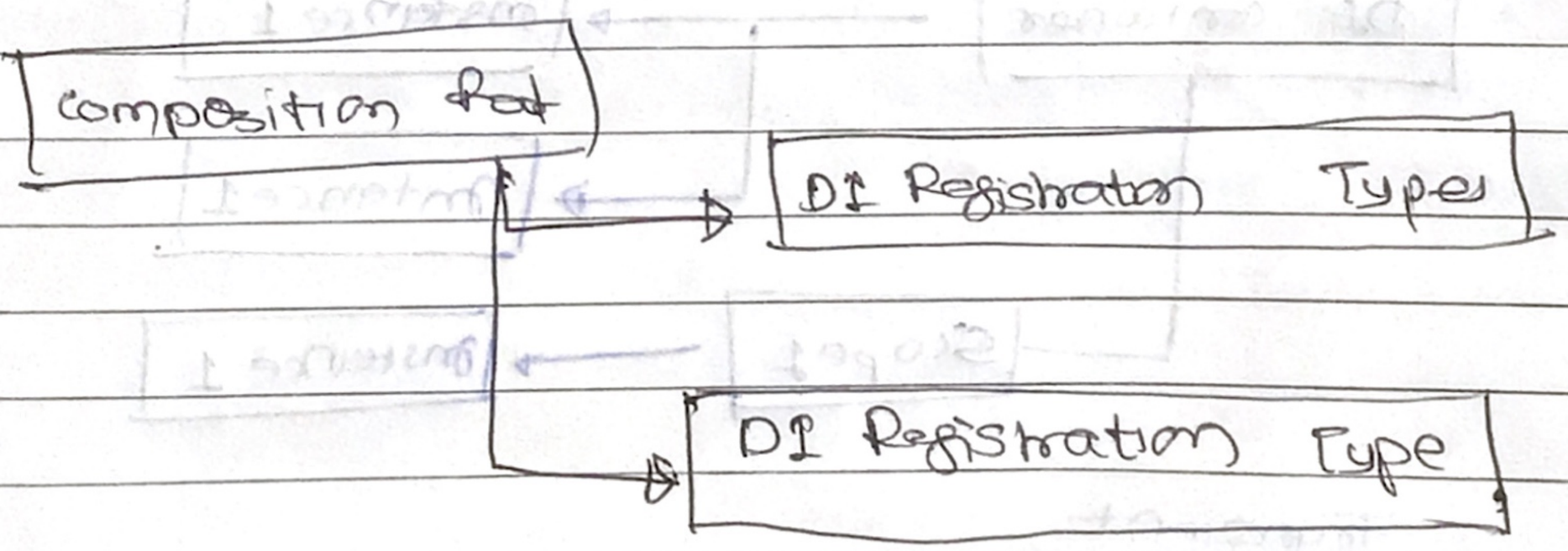
then we can have this type of structure.

Add VS TryAdd

☆) **Add {LifeTime}** ⇒ If there is an existing registration for the type, this call will overwrite it.

☆) **TryAdd {LifeTime}** ⇒ If there is an existing registration for the type this call will do nothing.

☆) If there are many composition root, then its good practice to create a extension method of IServiceCollection & use it.



HttpClient

```
services.AddHttpClient<IProductSource, HttpProductSource>()
  .ConfigureHttpClient(client =>
      {
          client.BaseAddress = new Uri("...");
      });
```

AppSetting Binding

```
services.AddOptions<CSVProductSource>()
  .Configure<IConfiguration>((option, configuration) =>
      {
          configuration.GetSection("sectionName").Bind
                                                  (options);
      });
```

Pass Parameter in dependency Injection

```
services.RegisterProductTransformation((options) => {
    options.EnableCurrency = false;
});
```

(extension method.)

Note a) If you have constructor that needs to have something hardcoded dependencies then in that case you can create a new wrapper class & (inject the) create a instance of it in it & return the object.

b) Also you can implement Lazy loading in initialization.

```
services.AddTransient<Lazy<IProductTransformer>>((service
                                                  Provider) =>
{
    return new Lazy<IProductTransformer>(() =>
    {
        var abc = serviceProvider.GetRequiredService<
                                                IServiceScopeFactory>();
        var xyz = ——————— <IWriteImproverStatic>();
```

Tata new ⇒ domestic & international.

Page No.
Date

```
        return new ProductTransformer(abc, xyz);
    });
});
```

ABTesting ⇐ Switching in two dependencies

```
services.AddTransient<IProductTarget>((serviceProvide) =>
{
    var configuration = serviceProvider.GetRequired
        Services<IConfiguration>();
    if(configuration.GetValue<int>("EnableCSVImport")
        == 1)
    { return serviceProvider.GetRequiredService<
        CSVProductTarget>();
    }

    return serviceProvider.GetRequiredService<
        GoogleProductTarget>();
});
```