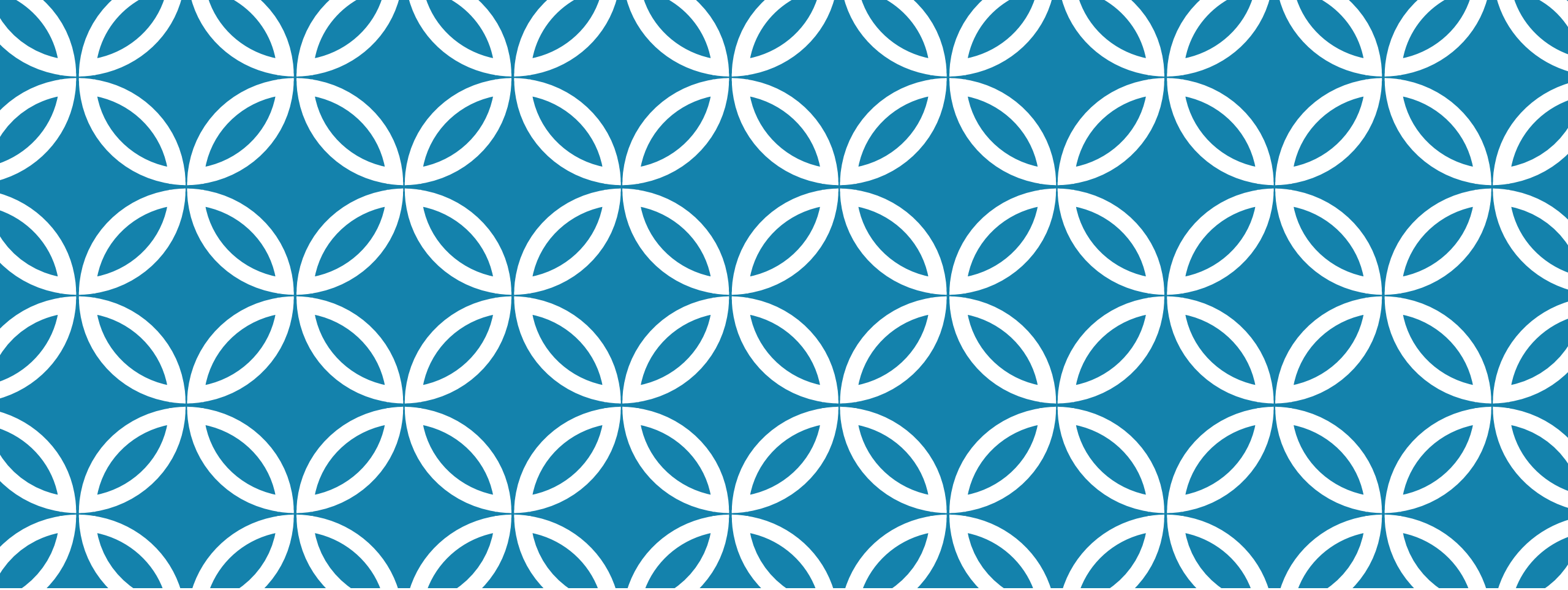


ОСНОВНИ ПРИНЦИПИ В ОБЕКТНО-ОРИЕНТИРАНОТО ПРОЕКТИРАНЕ

РАЗЛИКА МЕЖДУ ПРИНЦИПИ И ШАБЛОНИ ЗА ПРОЕКТИРАНЕ

- Принципите за проектиране осигуряват насоки на високо ниво за проектиране на по-добри софтуерни приложения. Те не предоставят насоки за реализация (имплементация) и не са обвързани с нито един език за програмиране. Принципите SOLID (SRP, OCP, LSP, ISP, DIP) са един от най-популярните набори от принципи за проектиране.
- Например Принципът за единствена отговорност (SRP) предполага, че даден клас трябва да има само една отговорност, а от там и само една причина за промяна. Това е твърдение на високо (абстрактно) ниво, което трябва да имаме предвид при проектирането или създаването на класове за нашето приложение. SRP не предоставя конкретни стъпки за реалзиция (имплементация), но от нас зависи как ще използваме SRP в приложението си.

РАЗЛИКА МЕЖДУ ПРИНЦИПИ И ШАБЛОНИ ЗА ПРОЕКТИРАНЕ

- Шаблоните за проектиране от друга страна предоставят решения на по-ниско ниво, свързани с реализацията на често срещани проблеми в обектно-ориентираното проектиране.
- Шаблоните за проектиране предлагат конкретно решение за конкретна ситуация или проблем в обектно-ориентираното програмиране
- Например, ако е нужно да имаме само един обект от даден клас, използваме Singleton шаблона, който ни предлага най-добрия подход да създадем клас, от който да има само един обект в приложението ни
- В заключение, шаблоните за проектиране ни дават конкретни решения в обектно-ориетираното проектиране, принципите ни дават “добрите практики” на високо ниво, които е добре да следваме

SOLID ПРИНЦИПИ

- SRP – Single Responsibility Principle
- OCP – Open/Closed Principle
- LSP – Liskov Substitution Principle
- ISP – Interface Segregation Principle
- DIP – Dependency Inversion Principle

Основно правило е, че за да постигнем желания резултат, софтуерът трябва да бъде написан възможно най-просто. Въпреки това, ако в даден момент развитието на софтуера стане сложно и болезнено, дизайнът му трябва да се коригира. Често тези принципи, в допълнение към по-общия принцип Don’t Repeat Yourself, могат да бъдат използвани като напътствия в процеса по рефакториране на софтуера с цел получаване на по-добър дизайн.

SINGLE-RESPONSIBILITY PRINCIPLE

- Един клас трябва да бъде отговорен само за една конкретна функционалност, предоставяна от софтуера
- “Един клас трябва да има само една единствена причина да бъде променян” – Робърт Мартин
- Определя отговорностите, които притежава един клас като причина за неговата промяна
- Пример: Представате си програмен клас, който е натоварен със задачата да създава и отпечатва справка (report). Промяна в този клас може да се наложи при две ситуации: промяна в съдържанието на справката или промяна в оформлението при отпечатване на тази справка. Причините водещи до тези промени са различни : едната е съществена, а другата козметична. Според SRP, тези две изменения са свързани с проблеми от различно естество, откъдето следва, че имаме две различни отговорности, с които е обременен нашия клас. От тук можем да направим извода, че функционалността (отговорностите) на този клас трябва да бъде разделена в два отделни класа. Лоша идея е да обвържем две изпълнявани действия, които се променят в следствие на различни причини в различен момент от времето. По този начин, когато имаме клас, натоварен само с една отговорност, значително по-рядко ще правим промени в него и той ще бъде много по устойчив на проблеми

SINGLE-RESPONSIBILITY PRINCIPLE

- SRP се свързва с концепцията за “обвързване” (coupling) и кохезия (cohesion)
- Coupling (low, high) – доколко отделните класове зависят един от друг – доколко промяна в един клас ще доведе до промяна в друг
- Cohesion (low, high) – доколко кодът, отговорен за конкретна функционалност е съсредоточен в даден клас или модул от класове
- Добре проектираните приложения се характеризират с ниско ниво на “обвързване” и високо ниво на кохезия. Това означава, че когато се наложи да се направи промяна в дадена функционалност на приложението, то тази промяна ще засегне минимален брой класове (в най-добрия случай само един) и тази промяна няма до доведе до други индиректни промени в други класове
- Целта е да създаваме “self-contained” (самодостатъчни), т.е. независими класове с една единствена и добре дефинирана отговорност

ПРИМЕРИ ЗА РАЗЛИЧНИ ОТГОВОРНОСТИ

- Persistence
- Validation
- Notification
- Error Handling
- Logging
- Class Selection / Instantiation
- Formatting
- Parsing
- Mapping

SRP ИЛИ НЕ-SRP?

This class violates both Single Responsibility Principle and Open/Closed Principle. Not surprisingly, I've run into issue modifying it.

OPEN-CLOSED PRINCIPLE (OCP)

- Според OCP софтуерните елементи (класове, модули, методи) трябва да бъдат отворени за разширения, но затворени за модификация.
- Напрактика това означава да създаваме класове, модули или методи чието поведение може да бъде променяно без да е необходимо да коригираме техния код. Пример : Представате си, че имаме метод, който съхранява данни във файл и името под което се записва файла е определено (“хардкоднато”) в метода. Ако изискванията се променят и това име трябва да е различно в определени ситуации, ние трябва да променим кода на този метод. Ако обаче името на файла се подава като параметър, ние можем да променим поведението на този метод, без да променяме неговия код
- OCP може да бъде постигнат по много начини включително чрез наследяване или използване на шаблон за проектиране като Strategy

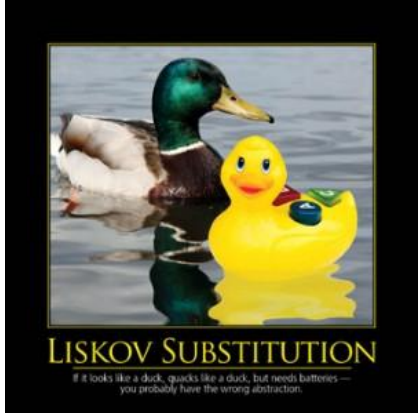
OPEN-CLOSED PRINCIPLE (OCP)

- Един клас е “затворен”, тъй като той има дефинирани данни и интерфейс, използван от други класове-клиенти
- От друга страна един клас е и “отворен”, тъй като може да бъде наследен, като по този начин се добавят нови възможности
- Когато добавим клас-наследник не е необходимо да променяме класа-родител (базовия клас)

LISKOV SUBSTITUTION PRINCIPLE (LSP)

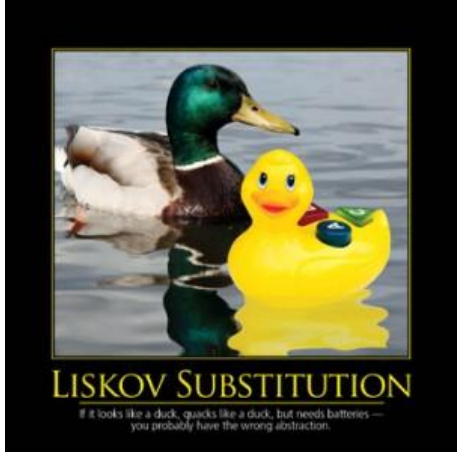
- Според LSP класовете наследници (subtypes) трябва да могат да бъдат използвани като заместители на техните базови класове (base types)
- Нарушаването на този принцип води до това, че на множество места в кода ще трябва да се прави проверка за реалния тип на даден обект
- Обикновено наследяването се обяснява така : ако един обект Y “е” (“IS–A”) обект “X”, то обект Y може да наследи обект X (Лисицате е животно, следователно класът Лисица може да наследи класът Животно) Това означава ли, че навсякъде където трябва да се използва обект Животно можем да използваме обект Лисица?
- Да вземем например Правоъгълник и Квадрат, да наистина Квадратът е Правоъгълник. Но правоъгълник има две страни x и y. Как ще осигурим в квадрата двете страни да са равни? Ще направим така, че когато X се промени, да се промени и Y със същата стойност

LISKOV SUBSTITUTION PRINCIPLE (LSP)



- По този начин при последователността от операции Square.X = 4, Square.Y = 3, страната на квадрата ще бъде 3;
- Ако в Правоъгълник класа имаме метод, който намира лицето на фигурата като return X * Y, то на теория всичко е наред. За правоъгълник обект с X = 3 и Y = 4 ще получим 12, а за квадрата от горния пример ще получим 9.
- Нека имаме парче код в което на обект Правоъгълник задаваме стойности за страните 3 и 4 и извикваме метода за намиране на лице. -> Резултат 12. Ако в същото парче код заменим Правоъгълник обекта с обект квадрат и зададем същите стойности за страните 3 и 4, ще получим резултат 16, което няма да е вярно, ако продължаваме да считаме квадрата за правоъгълник

LISKOV SUBSTITUTION PRINCIPLE (LSP)



- От гореописаното можем да заключим, че един клас наследява друг клас, ако без проблеми можем да докажем, че първият клас може да замени (“IS-SUBSTITUTABLE-FOR”) втория клас.
- Често нарушение на принципа за заместване на Лишков е – да НЕ направим реално имплементиране всички методи на даден интерфейс или базов клас, а в тяхното тяло да поставим създаване на Exception (NotImplementedException). Когато този клас се използва само в един проект или има само един клас-клиент, това не е толкова голям проблем. Но ако този клас стане част от библиотека, която ще се използва от множество други класове и модули, това вече е проблем.
- **Liskov Substitution Principle "Is-Substitutable-For" Not Just "Is-A"**

INTERFACE SEGREGATION PRINCIPLE (ISP)

- Според ISP (Принцип на разделяне на интерфейса) ползвателите на даден клас не трябва да бъдат принуждавани да зависят от методи, предоставящи функционалност, която те не използват
- Разработчиците трябва да се предпочитат малки “thin” интерфейси, пред големи “fat” интерфейси, които предлагат повече функционалност, от която един клас или метод се нуждае.
- Повечето езици за програмиране предоставят възможност за наследяване на интерфейси. По този начин, ако се нуждаем от по-обширен интерфейс в дадена част от приложението, можем да конструираме такъв, обединявайки няколко по-малки

INTERFACE SEGREGATION PRINCIPLE (ISP)


```
public interface IMembership
{
  bool Login(string username, string password);
  void Logout(string username);
  Guid Register(string username, string password, string email);
  void ForgotPassword(string username);
} //  По-добрия вариант е да разделим тези 4 метода в 2 отделни интерфейса

public interface ILogin
{
  bool Login(string username, string password);
  void Logout(string username);
}

public interface IMembership : ILogin
{
  Guid Register(string username, string password, string email);
  void ForgotPassword(string username);
}
```

INTERFACE SEGREGATION PRINCIPLE (ISP)


```
public interface ILogin
{
  bool Login(string username, string password);
  void Logout(string username);
}
public interface IRegister
{
  Guid Register(string username, string password, string email);
}
public interface IForgotPassword
{
  void ForgotPassword(string username);
}

public interface IMembership : ILogin, IRegister, IForgotPassword
{
}
```

INTERFACE SEGREGATION PRINCIPLE (ISP)

- В идеалния случай трябва да постигнем високо ниво на кохезия, което означава интерфейсите да групират само логически свързани операции.
- Интерфейси с по-малко операции са по-лесни за пълно имплементиране от класовете, в резултат на което не се нарушава принципът на Лишков

DEPENDENCY INVERSION PRINCIPLE (DIP)

- Според принципа на инверсия на зависимостта модулите на системата от по-високо ниво не трябва да зависят от модулите от по-ниско ниво
- И двата вида модули трябва да зависят от абктракции
- Абстракциите не трябва да зависят от детайли (те скриват детайлите).
- Детайлите трябва да зависят от абстаркцията
- Често когато създаваме софтуер правим грешката всеки клас или метод да реферира директно класовете с които си сътрудничи. Този начин на програмиране води до липса на важните слоеве на абстракция, което от своя страна води до силно обвързани класове, зависими един от друг (tightly coupled system). Проблемът е още по-голям, когато тези класове са от различните слоеве на системата.

DEPENDENCY INVERSION PRINCIPLE (DIP)

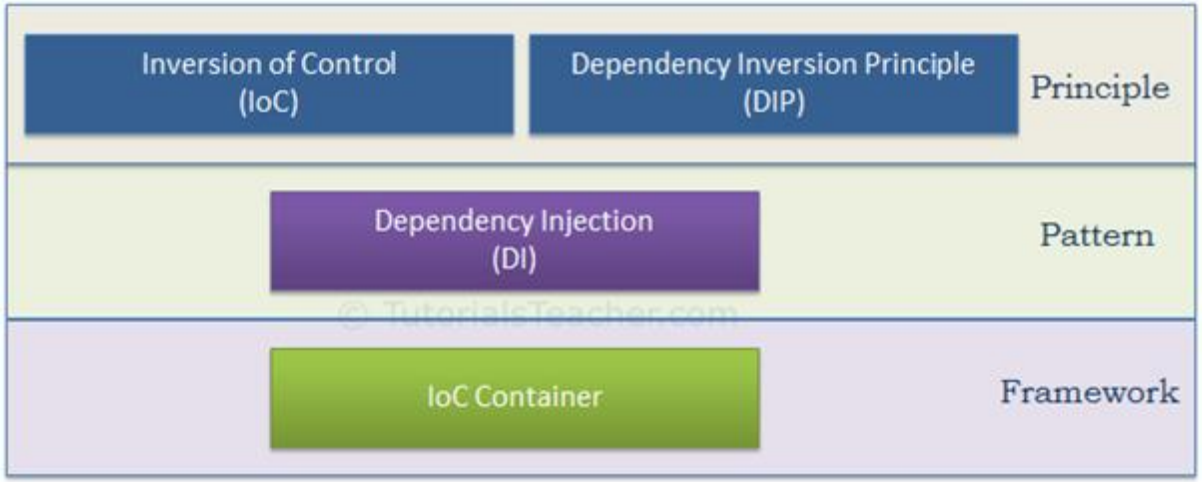
- Представете си, че имаме прозорец от потребителския интерфейс (GUI) на приложение, в който при натискане на бутон се извиква събитие, в което се създава обект от слоя с бизнес логиката (BLL) и се извиква негов метод.
- В този метод на обекта от BLL се създава обект от слоя за запис на данни в база данни (DAL) и се извиква негов метод, който изпълнява заявка към базата данни (DB)
- Получаваме силно обвързана верига от обекти. Зависимостта е следната GUI>BLL>DAL>DB и тя е транзитивна.
- Начинът да премахнем тези зависимости е да използваме DIP, което обикновенно започва с деклариране на необходимите интерфейси

DEPENDENCY INVERSION PRINCIPLE (DIP)

- Отново имаме бутон от потребителския интерфейс, който се натиска. Този път вместо да се създаде обект от BLL, се използва интерфейсна променлива, която предварително е инициализирана при създаване на прозореца чрез метод наречен **Dependency Injection.** Ключовата дума “new” няма да бъде използвана, както и няма да се извика статичен метод на друг клас.
- Извиква се метод на интерфейса
- Ако този метод реализира ключова бизнес логика, но изисква достъп до DAL, обектът имплементиращ метода ще има интерфейсно поле, инициализирано с подходящия DAL обект.
- По този начин чрез Dependency injection се избягва пряка зависимост между отделните слоеве на една система.

INVERSION OF CONTROL (IOC)

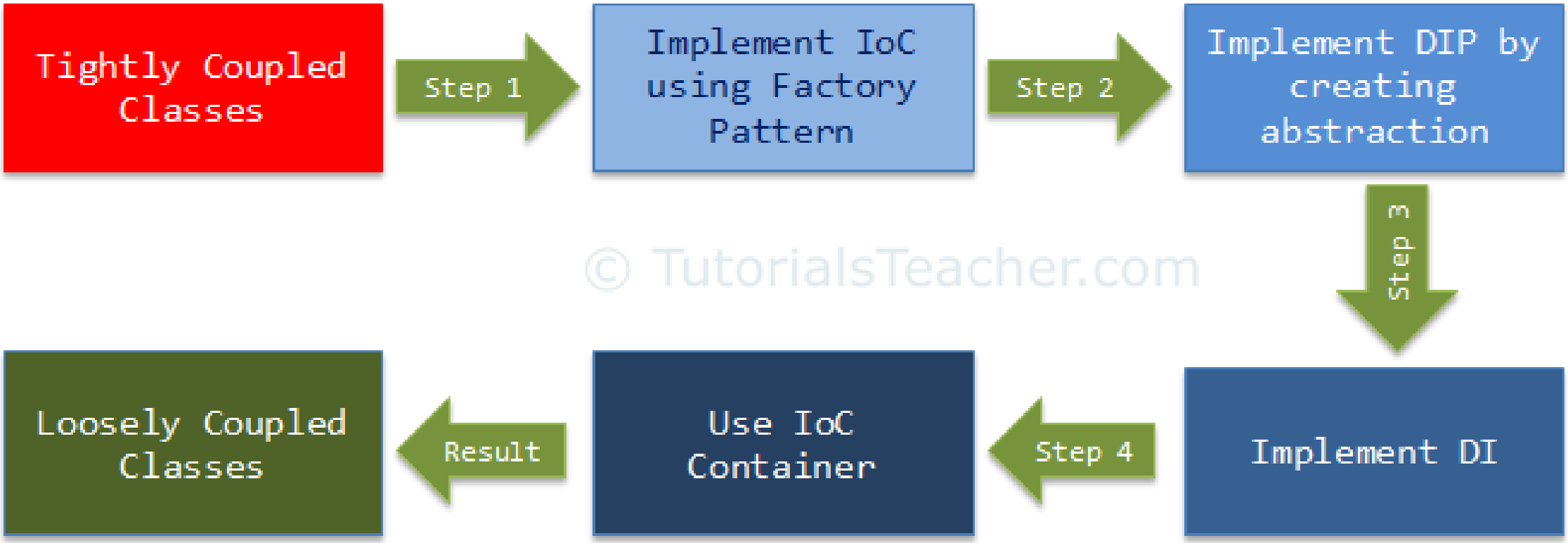
- IoC е принцип на проектиране, който препоръчва инвертирането (обръщането) на контрола (управлението, отговорностите) в обектно-ориентирания дизайн, за да се постигне слабо обвързване между класовете на дадено приложение. В този случай контролът се отнася до всякакви допълнителни отговорности, които даден клас има, освен основната му отговорност, като например контрол върху потока на приложение или контрол върху създаването на зависими обекти



- DIP също помага за постигане на слаба зависимост между класовете. Силно препоръчително е да използвате DIP и IoC заедно, за да постигнете максимално добра слаба обвързаност между класовете

DIP предполага, че модулите от високо ниво не трябва да зависят от модулите от ниско ниво. И двата вида модули трябва да зависят от абстракцията.

ИЗПОЛЗВАНЕ НА IOC И DIP



https://www.tutorialsteacher.com/ioc/introduction

IOC

- Инвертиране на контрол, може да се представи по следния начин чрез делегиране на отговорности - представате си, че шофирате автомобил. Тогава IoC би могъл да се представи с наемането на такси, където друг човек ще бъде отговорен за управлението на автомобила, т.е. за вашето придвижване. Има два вида контрол:

- Контрол по управлението на потока на изпълнение на приложението - в конзолните приложения потокът на изпълнение се управлява от Main метода на приложението. Пример за IoC е създаването на Windows Form приложение, в което потока на изпълнение се управлява от .NET Framework посредстово генерирането на събития

- Контрол върху създаването на зависими обекти

КОНТРОЛ ВЪРХУ СЪЗДАВАНЕТО НА ЗАВИСИМИ ОБЕКТИ

```
public class A
{
    B b;
    public A()
    {
        b = new B();
    }

    public void Task1()
    {
        // do something here..
        b.SomeMethod();
        // do something here..
    }
}

public class B
{
    public void SomeMethod()
    {
        //doing something..
    }
}
```

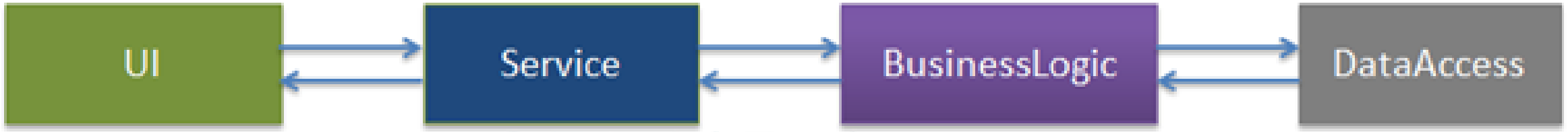
- Класът A създава обект от клас B за да изпълни метода си Task1(), без него това няма да е възможно. Клас A е зависим от клас B или клас B е dependency (зависимост) на клас A. В случая обект на A управлява кога се създава обект от B и неговия живот.
- IoC принципът пропагандира инвертираннето на контрола, т.е. делегира управлението на създаването на обект от клас B на друг клас – най-често чрез използването на Factory шаблон

КОНТРОЛ ВЪРХУ СЪЗДАВАНЕТО НА ЗАВИСИМИ ОБЕКТИ

```
public class A
{
    B b;
    public A()
    {
        b = Factory.GetObjectOfB();
    }
    public void Task1()
    {
        // do something here..
        b.SomeMethod();
        // do something here..
    }
}
public class Factory
{
    public static B GetObjectOfB()
    {
        return new B();
    }
}
```

- След прилагането на IoC Класът A НЕ създава директно обект от клас B, а делегира това действие на Factory класа
- За да премахнем напълно зависимостта на клас A от клас B, трябва да приложим и DIP като създадем Interface, който се имплементира от класа B
- Тогава Factory методът ще връща интерфейса имплементиран от класа B, а не обект от B

ЗАВИСИМОСТИ В N СЛОЙНА АРХИТЕКТУРА



В типичната n-слойна архитектура, Потребителският интерфейс (UI) използва “Слой на услугите” за извличане или запазване на данни. Слоя на услугите използва клас BusinessLogic, за да прилага бизнес правила върху данните. Класът BusinessLogic зависи от класа DataAccess, който извлича или запазва данните в базата данни. Това е прост n-слой архитектурен модел. Нека се съсредоточим върху класовете BusinessLogic и DataAccess, за да разберем IoC.

https://www.tutorialsteacher.com/ioc/inversion-of-control

```
public class CustomerBusinessLogic
        {
            DataAccess _dataAccess;
            public CustomerBusinessLogic()
            {
                _dataAccess = new DataAccess();
            }
            public string GetCustomerName(int id)
            {
                return _dataAccess.GetCustomerName(id);
            }
        }

        public class DataAccess
        {
            public DataAccess()
            {
            }

            public string GetCustomerName(int id)
            {
                return "Dummy Customer Name"; // get it from DB in real app
            }
        }
```

ПРОБЛЕМИ НА ТЕЗИ ДВА КЛАСА

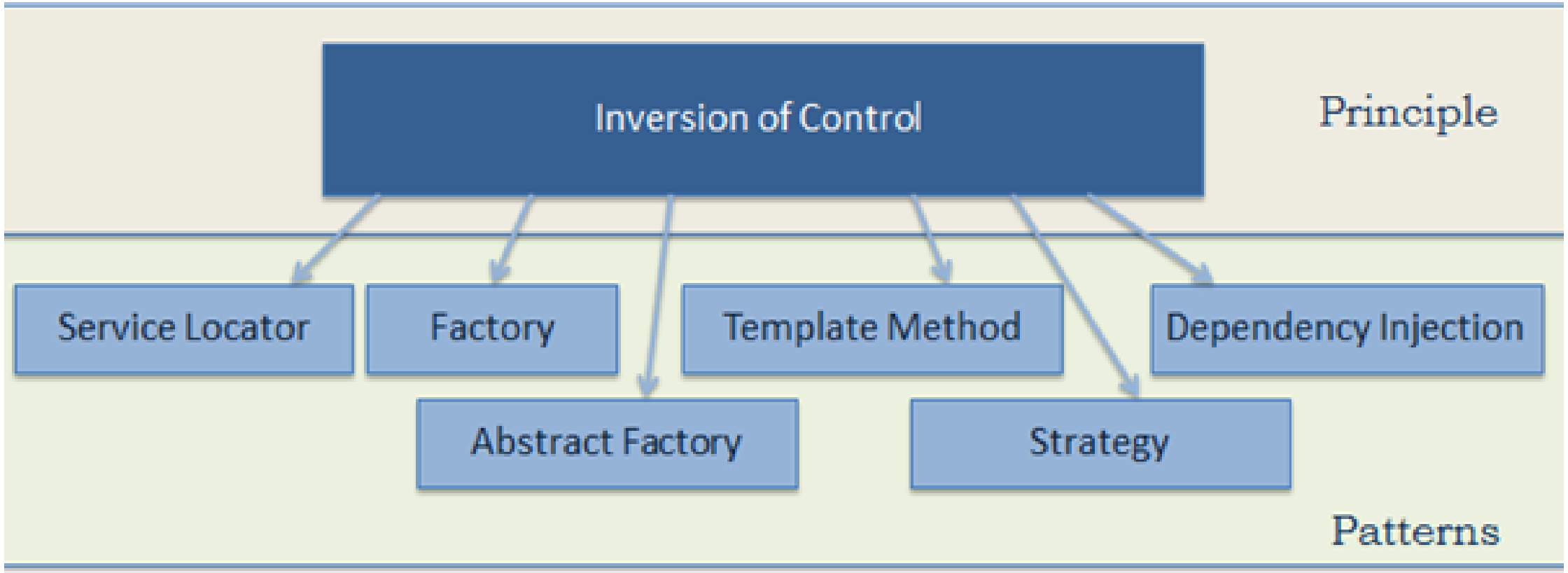
- Класовете CustomerBusinessLogic и DataAccess са тясно свързани класове. Това означава, че промените в класа DataAccess ще доведат до промени в класа CustomerBusinessLogic. Например, ако добавим, премахнем или преименуваме всеки метод в класа DataAccess, тогава трябва да променим съответно класа CustomerBusinessLogic.

- Да предположим, че данните за клиенти идват от различни бази данни или уеб услуги и в бъдеще може да се наложи да създадем различни класове, така че това ще доведе до промени в класа на CustomerBusinessLogic.

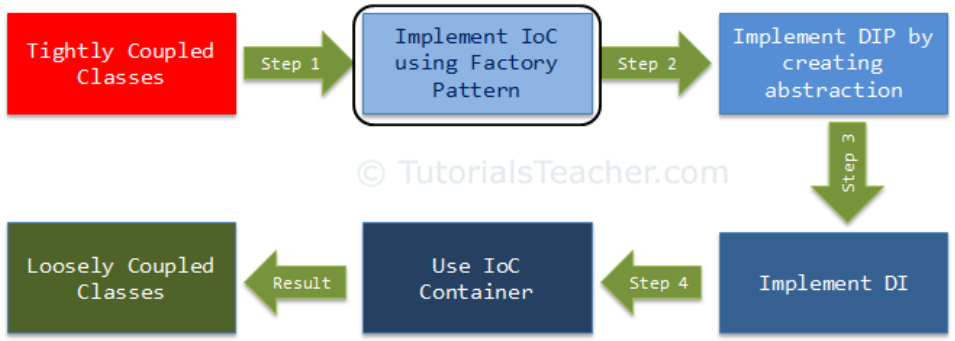
- Класът CustomerBusinessLogic създава обект от класа DataAccess, използвайки ключова дума new. Възможно е да има няколко класа, които използват клас DataAccess и създават неговите обекти. Това означава че, ако променим името на класа, тогава трябва да намерим всички места в сорс кода си, където сме създали обекти на DataAccess, и да извършим промените в целия код. Това е повтарящ се код за създаване на обекти от същия клас и поддържане на техните зависимости.

- Тъй като класът CustomerBusinessLogic създава обект от конкретния клас DataAccess, той не може да бъде тестван независимо (TDD). Класът DataAccess не може да бъде заменен с от друг mock (макет) клас .

ПРИЛАГАНЕ НА INVERSION OF CONTROL

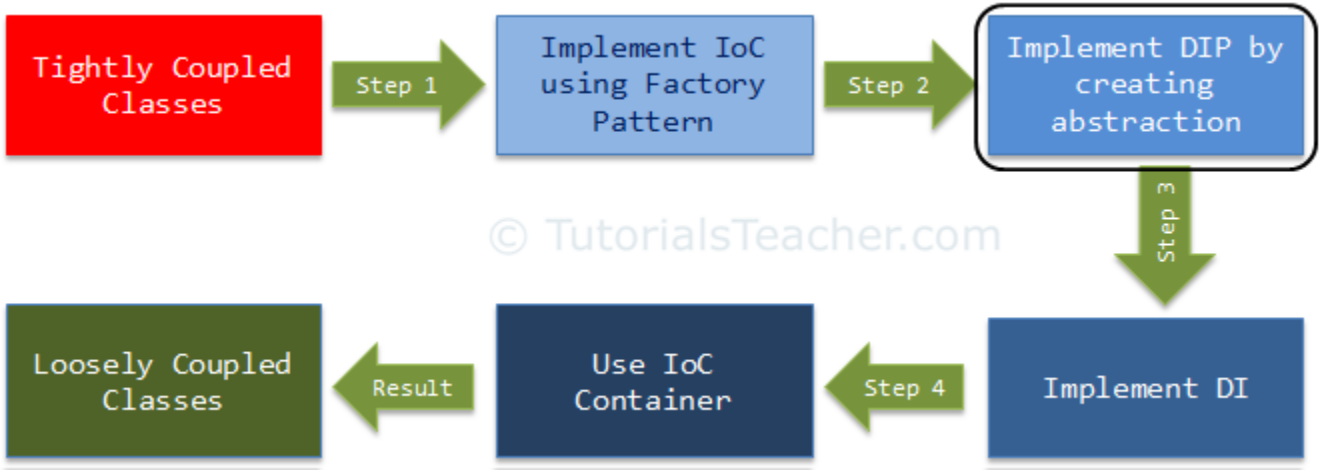


ПЪРВА СТЪПКА - IOC




```
public class DataAccessFactory
        {
            public static DataAccess GetDataAccessObj()
            {
                return new DataAccess();
            }
        }

        public class CustomerBusinessLogic
        {

            public CustomerBusinessLogic()
            {
            }

            public string GetCustomerName(int id)
            {
                DataAccess _dataAccess = DataAccessFactory.GetDataAccessObj();

                return _dataAccess.GetCustomerName(id);
            }
        }
```

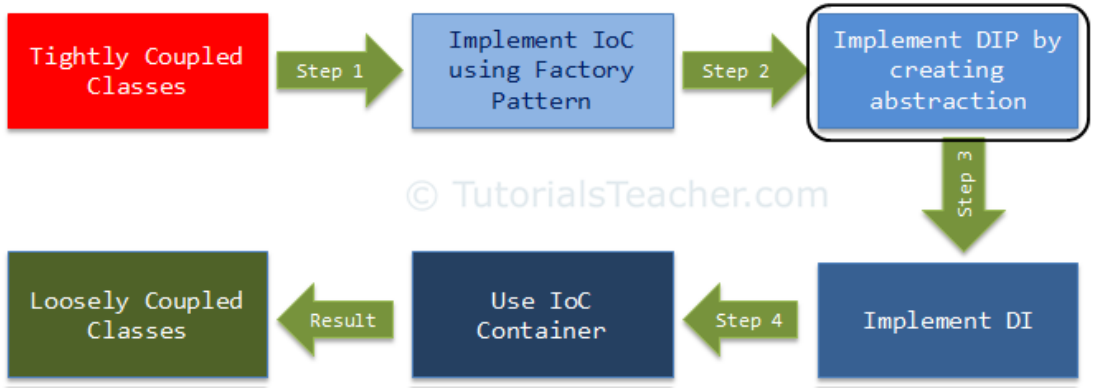
ВТОРА СТЪПКА - DIP



- Модулите от високо ниво не трябва да зависят от модулите с ниско ниво. И двете трябва да зависят от абстракцията.
- Абстракциите не трябва да зависят от подробности. Детайлите трябва да зависят от абстракциите.
- Абстракция и капсулиране са важни принципи за обектно-ориентираното програмиране. Има много различни определения от различни хора, но нека да разберем абстракцията, използвайки горния пример.
- Абстракция означава нещо, което не е конкретно. По отношение на програмирането, горепосочените CustomerBusinessLogic и DataAccess са конкретни класове, което означава, че можем да създаваме обекти от тях. И така, абстракцията в програмирането означава да се създаде интерфейс или абстрактен клас, който не е конкретен. Това означава, че не можем да създадем обект от интерфейс или абстрактен клас. Според DIP, CustomerBusinessLogic (модул на високо ниво) не трябва да зависи от конкретния клас DataAccess (модул на ниско ниво). И двата класа трябва да зависят от абстракциите, което означава, че и двата класа трябва да зависят от интерфейса или абстрактния клас

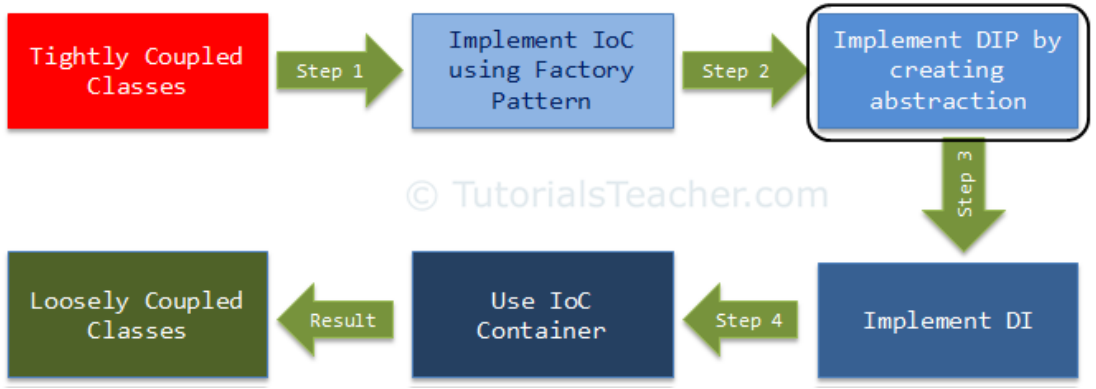
https://www.tutorialsteacher.com/ioc/dependency-inversion-principle

ВТОРА СТЪПКА - DIP



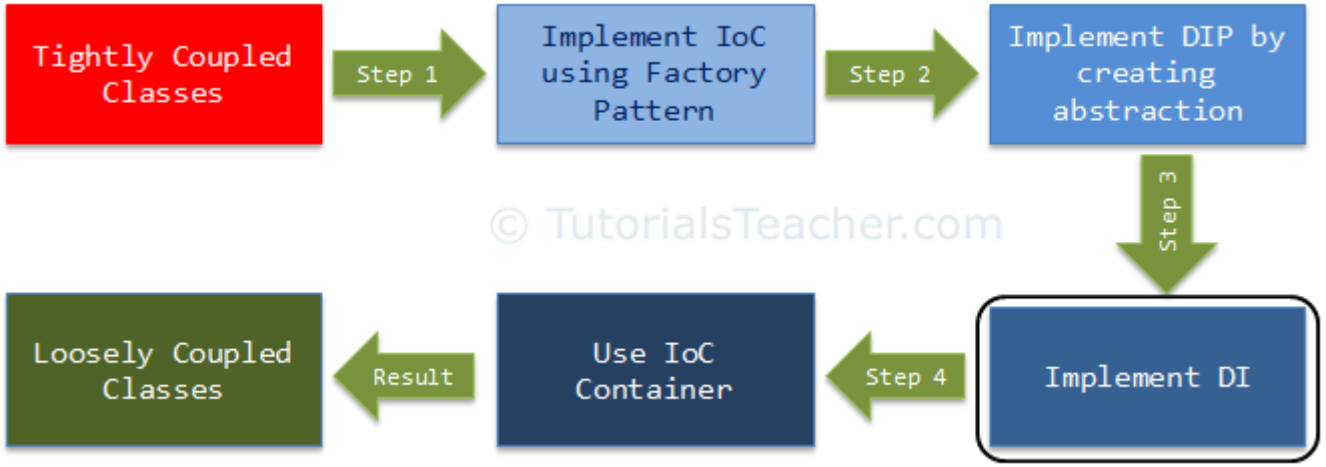

```
public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}
public class CustomerDataAccess : ICustomerDataAccess
{
    public CustomerDataAccess()  {}

    public string GetCustomerName(int id)
    {
        return "Dummy Customer Name";
    }
}
public class DataAccessFactory
{
    public static ICustomerDataAccess GetCustomerDataAccessObj()
    {
        return new CustomerDataAccess();
    }
}
```

ВТОРА СТЪПКА - DIP



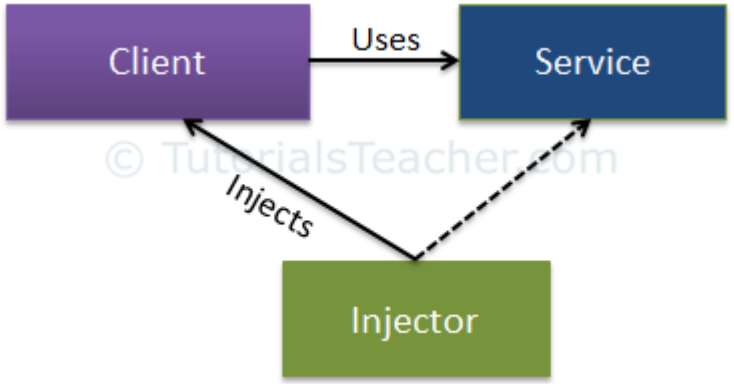

```
public class CustomerBusinessLogic
        {
            ICustomerDataAccess _custDataAccess;
            public CustomerBusinessLogic()
            {
                _custDataAccess =
DataAccessFactory.GetCustomerDataAccessObj();
            }
            public string GetCustomerName(int id)
            {
                return _custDataAccess.GetCustomerName(id);
            }
        }
```

DEPENDENCY INJECTION



- Dependency Injection (DI) е модел на проектиране, използван за прилагане на IoC. Той позволява създаването на обекти, от които даден клас има нужда извън него и му предоставя тези обекти по няколко различни начина. Използвайки DI, преместваме създаването и обвързването на зависимите обекти извън класа, който зависи от тях.

- Моделът на DI включва 3 типа класове.
  - Клас-клиент: Класът-клиент (зависимият клас) е клас, който зависи от класа на услугите
  - Клас на услугите : Класът на услугите (клас-зависимост) е клас, който предоставя функционалност на класа клиент.
  - Инжектиращ клас : Инжектира обекта на класът на услугите в класа-клиент.



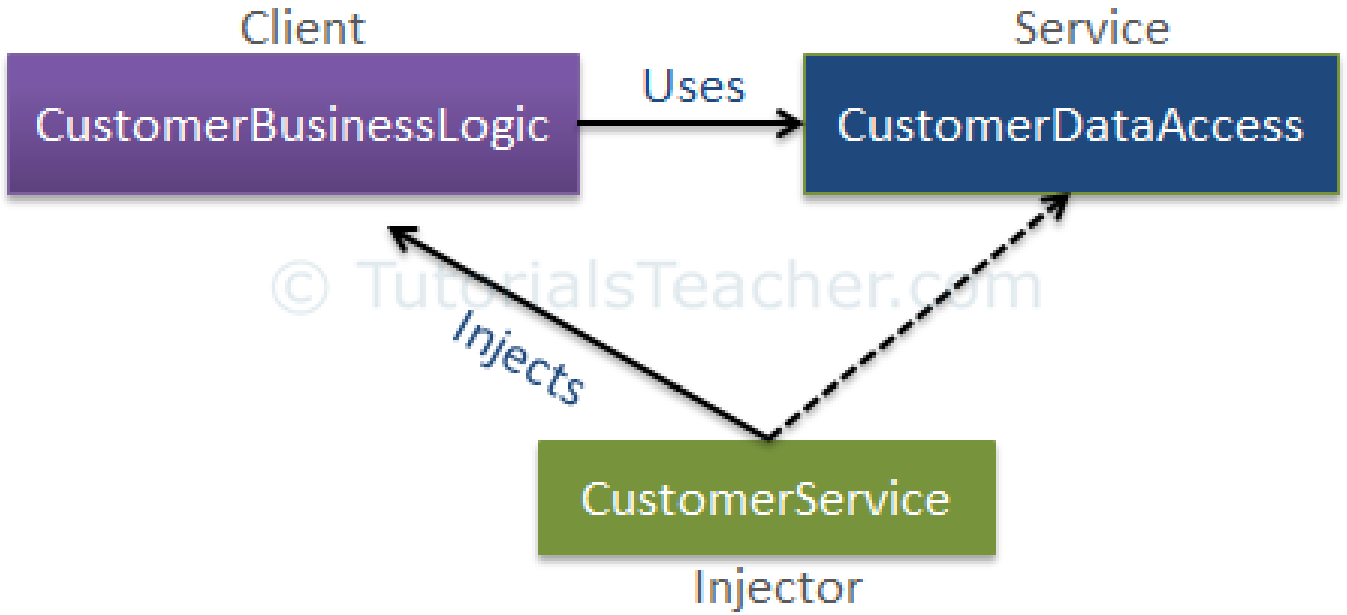
https://www.tutorialsteacher.com/ioc/dependency-injection

ВИДОВЕ DI

- Както видяхме по-горе, класът инжектор инжектира обект от класа услуга (зависимостта) на клиента (зависимия клас). Класът инжектор инжектира зависимостите по три начина: чрез конструктор, чрез свойство или чрез метод.
- Инжектиране в конструктор: При инжектиране в конструктора, инжекторът предоставя услугата (зависимостта) чрез конструктора на клиентски клас.
- Инжектиране на свойства: При инжектирането на свойство (известно също като Setter Injection) инжекторът доставя зависимостта чрез публично свойстно на класа-клиент.
- Инжектиране чрез метод: При този тип инжектиране клиентският клас имплементира интерфейс, който декларира метода (ите) за доставяне на зависимостта и инжекторът използва този интерфейс, за да достави зависимостта на клиентския клас.

ПРИЛАГАНЕ НА DI

- Проблемът с горния пример е, че използвахме DataAccessFactory вътре в класа CustomerBusinessLogic. Така че, ако предположим, че има друга реализация на ICustomerDataAccess и искаме да използваме този нов клас в CustomerBusinessLogic, то ще трябва да променим и сорс код на класа CustomerBusinessLogic. Моделът на DI решава този проблем чрез инжектиране на зависими обекти чрез конструктор, свойство или интерфейс.



DI С КОНСТРУКТОР

```
public class CustomerBusinessLogic
{   ICustomerDataAccess _dataAccess;
    public CustomerBusinessLogic(ICustomerDataAccess custDataAccess)
    {
        _dataAccess = custDataAccess;
    }
    public CustomerBusinessLogic()
    {
        _dataAccess = new CustomerDataAccess();
    }
    public string ProcessCustomerData(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
}
public class CustomerService
{  CustomerBusinessLogic _customerBL;
    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic(new CustomerDataAccess());
    }
    public string GetCustomerName(int id)
    {
        return _customerBL.GetCustomerName(id);
    }
```

DI C PROPERTY INJECTION

```
public class CustomerBusinessLogic
    {
      public CustomerBusinessLogic() {}
      public string GetCustomerName(int id)
        {
            return DataAccess.GetCustomerName(id);
        }
        public ICustomerDataAccess DataAccess { get; set; }
    }

    public class CustomerService
    {
        CustomerBusinessLogic _customerBL;
        public CustomerService()
        {
            _customerBL = new CustomerBusinessLogic();
            _customerBL.DataAccess = new CustomerDataAccess();
        }

        public string GetCustomerName(int id)
        {
            return _customerBL.GetCustomerName(id);
        }
    }
```

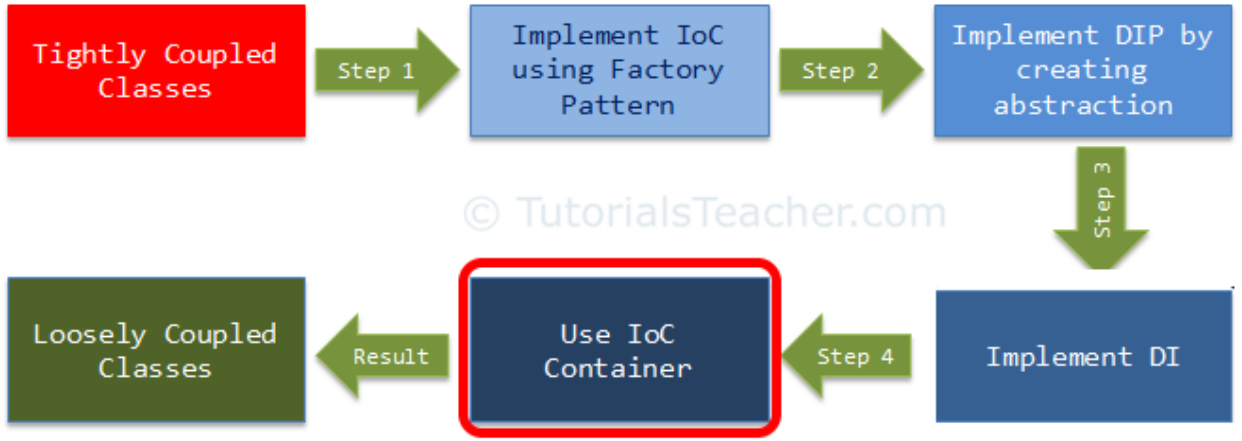
DI C METHOD INJECTION

```
interface IDataAccessDependency
 {
     void SetDependency(ICustomerDataAccess customerDataAccess);
 }
 public class CustomerBusinessLogic : IDataAccessDependency
 {   ICustomerDataAccess _dataAccess;
     public CustomerBusinessLogic() {}
     public string GetCustomerName(int id)
     {
         return _dataAccess.GetCustomerName(id);
     }

     public void SetDependency(ICustomerDataAccess customerDataAccess)
     {
         _dataAccess = customerDataAccess;
     }
 }
```

DI C METHOD INJECTION

```
public class CustomerService
        {   CustomerBusinessLogic _customerBL;
            public CustomerService()
            {
                _customerBL = new CustomerBusinessLogic();
                ((IDataAccessDependency)_customerBL).SetDependency(new
CustomerDataAccess());
            }

            public string GetCustomerName(int id)
            {
                return _customerBL.GetCustomerName(id);
            }
        }
```

IOC CONTAINER



- IoC контейерът е библиотека за автоматично инжектиране на зависимости. Тя управлява създаването на обекти, техния живот, а също и инжектирането им като зависимости в обекти.
- IoC контейнерът създава обект от определения клас и инжектира всички обекти, зависими от него чрез конструктор, свойство или метод по време на изпълнение на приложенинето. Инжекцията се изпълнява в подходящото време. Това се прави, за да не се налага ръчно да създаваме и управляваме обекти. Всички контейнери трябва да поддържат следните операции, необходими за имплементиране на DI.
  - Register: Контейнерът трябва да знае коя зависимост да инстанцира, когато срещне определен тип. Този процес се нарича регистрация. По принцип тя трябва да включва начин за създаване на съответствие между типове.
  - Resolve: Когато използваме IoC контейнер, не е необходимо да създаваме обекти ръчно. Контейнерът го прави вместо нас. Това се нарича резолюция. Контейнерът трябва да включва някои методи за определяне на посочения тип; контейнерът създава обект от посочения тип, инжектира необходимите зависимости, ако има такива и връща обекта.
  - Dispose: Контейнерът трябва да управлява живота на създаваните обекти. Повечето IoC контейнери включват различни времеви мениджъри, които управляват жизнения цикъл на обектите.

РЕЗУЛТАТ – РЕФАКТОРИРАН И ПО-ЛЕСЕН ЗА ПОДДРЪЖКА СОРС КОД