Tasarım Şablonları 5. hafta

24.03.2025

SOLID PRINCIPLES

S : single responsibility principle

O : open/closed principle

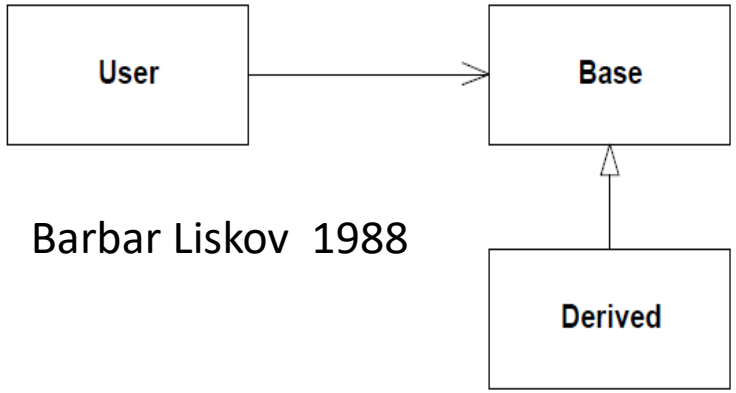
L : Liskov Substitution Principle

I: Interface Segregation

D: Dependency Inversion

Liskov Substitution Principle
Subtypes must be substitutable for their base types

- Derived sınıflar Base sınıfları ile yer değiştirebilir olmalıdır.
- Base sınıfının kullanıcısı, Base sınıfa ait Derived sınıfı kendisine geçirildiğinde düzgün bir şekilde çalışmaya devam etmelidir.



Barbar Liskov 1988

LSP Schema

Mathematical Definition (Liskov 1988)

- If for each object **o1** of type S there is an object **o2** of type T such that for all programs **P** defined in terms of **T,** the behavior of **P** is unchanged when **o1** is substituted for **o2** then **S** is a subtype of **T.**

```
void User(Base& b);
Derived d;
User(d);
```

- ❑Preconditions cannot be strengthened in the subtype.
- ❑Postconditions cannot be weakened in the subtype.
- ❑Invariants cannot be weakened in the subtype.

When Is a Subtype Substitutable for Its Supertype?

- Bir alt tip (subtype), otomatik olarak üst tipinin (süper type) yerine kullanılamaz.
  - Alt tipin yerine kullanılabilir olması için, alt tipin üst tip gibi davranması gerekir.
- Bir nesnenin davranışı (behavior), client tarafından güvenilebilir bir sözleşmedir.
  - The behavior is specified by the **public methods**
  - The behavior is specified by any **constraints** placed on their inputs
  - The behavior is specified by any **state changes** that the object goes through

- Java'da alt tipleme (subtyping), temel sınıfın özelliklerinin ve yöntemlerinin alt sınıfta da olmasını gerektirir
- Davranışsal alt tipleme (behavior subtyping), bir alt tipin yalnızca üst tipteki tüm yöntemleri sağlaması değil, aynı zamanda üst tipin davranışsal belirtimlerine de uyması gerektiği anlamına gelir.
- Bu, istemcilerin (client) üst tip davranışı hakkında yaptığı tüm varsayımların alt tip tarafından karşılanmasını sağlar.
- Bu, Liskov Substitution Prensibinin nesne yönelimli tasarıma getirdiği ek kısıtlamadır.

```
class Bird {          // LSP  ihlali örneği
    public void fly() {
        System.out.println("Bird is flying");        }   }
class Sparrow extends Bird {
    @Override
    public void fly() {
        System.out.println("Sparrow is flying");           }      }
class Penguin extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Penguins can't fly");          }     }
// throw programın akışında değişimler  oluşturur.
public class Main {
    public static void main(String[] args) {
        Bird bird = new Sparrow();
        bird.fly(); // Output: Sparrow is flying
        bird = new Penguin();
        bird.fly(); // Throws UnsupportedOperationException      }      }
```

The Penguin sınıfı LSP bozar.
Çünkü throw ile, fly metodunun davranışını değiştirir.
Bu, Bird sınıfının beklenen davranışını bozar ve run time hatası olur.

throw ve throws farkı

```
public class Main {
 static void checkAge(int age) throws ArithmeticException {
if (age < 18) {
            throw new ArithmeticException("Access denied - You must be at least 18 years
old.");     }
      else {      System.out.println("Access granted - You are old enough!");
   }   }

 public static void main(String[] args) {
  checkAge(15);      // Set age to 15 (which is below 18...)
 }     }
```

throw bir metoda ait kod bloğunun içerisinde yani metodun gövdesinde yazılır.
throws ise metod bildiriminde metodun isminden sonra yazılır.
throw sözcüğünden sonra bir **nesne** oluşturulur. throws sözcüğünden sonra ise **Exception sınıf** ismi yazılır.

LSP içeren kod

```
abstract class Bird {
   public abstract void eat();        }

class Sparrow extends Bird {
   @Override
   public void eat() {
      System.out.println("Sparrow is eating");       }
   public void fly() {
      System.out.println("Sparrow is flying");    }
                                        }
class Penguin extends Bird {
   @Override
   public void eat() {
      System.out.println("Penguin is eating");        }
   public void swim() {
      System.out.println("Penguin is swimming");      }
                                        }
```

«Liskov Substitution Ppinciple» Bağımlılığı

```
public class Main {
    public static void main(String[] args) {
        Bird sparrow = new Sparrow();
        sparrow.eat();  // Output: Sparrow is eating
        ((Sparrow) sparrow).fly();  // Output: Sparrow is flying

        Bird penguin = new Penguin();
        penguin.eat(); // Output: Penguin is eating
        ((Penguin) penguin).swim(); // Output: Penguin is swimming
    }
}
```

Bird: Tüm kuş türleri için ortak bir arayüz sağlayan soyut bir sınıf.

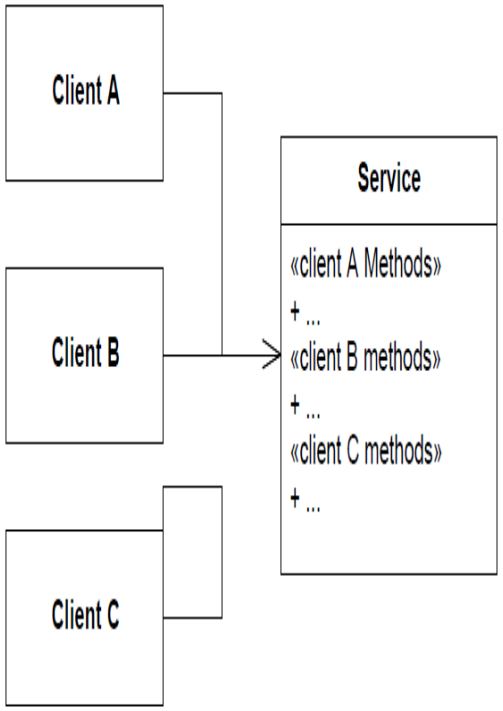
Sparrow ve Penguin: Bird sınıfının davranışını değiştirmeden onu genişleten alt sınıflar.

Main: Alt sınıfların parent sınıfla dönüşümlü olarak kullanılmasına izin vererek LSP kullanımını gösterir.

- LSP, nesne yönelimli tasarımda, programın doğruluğu değişmeden alt sınıfların parent sınıfları değiştirebilmesini sağlayan temel bir kavramdır.
- LSP'ye bağlı kalarak daha esnek (flexible) , yeniden kullanılabilir (reusable) ve sürdürülebilir (maintainable) kodlar oluşturulur.

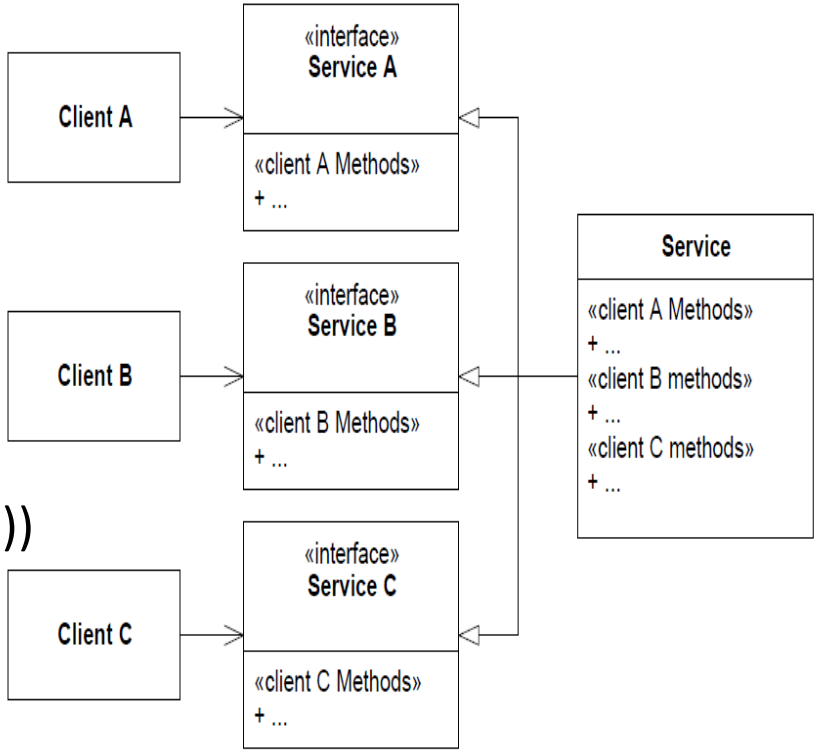
Interface Segregation Principle (ISP)

- Pek çok client (kullanıcı) için farklı arayüzler, tek bir genel amaçlı arayüzden daha iyidir
- ISP, bileşenlerin alt katmanlarını destekleyen etkin teknolojilerden bir diğeridir (COM gibi)
- ISP olmasa bileşenler ve sınıfların çok daha az kullanışlılığı ve taşınabilirliği düşer. olacaktı.
- ISP'ye göre birden fazla client olan bir sınıfı varsa, sınıfı client ların tümüne ait yöntemleri yüklemek yerine, her client için özel arayüz oluşturulur ve bunların sınıfları oluşturulur



Whenever a change is made to one of the methods that ClientA calls, ClientB and ClientC may be affected.
It may be necessary to recompile and redeploy them.

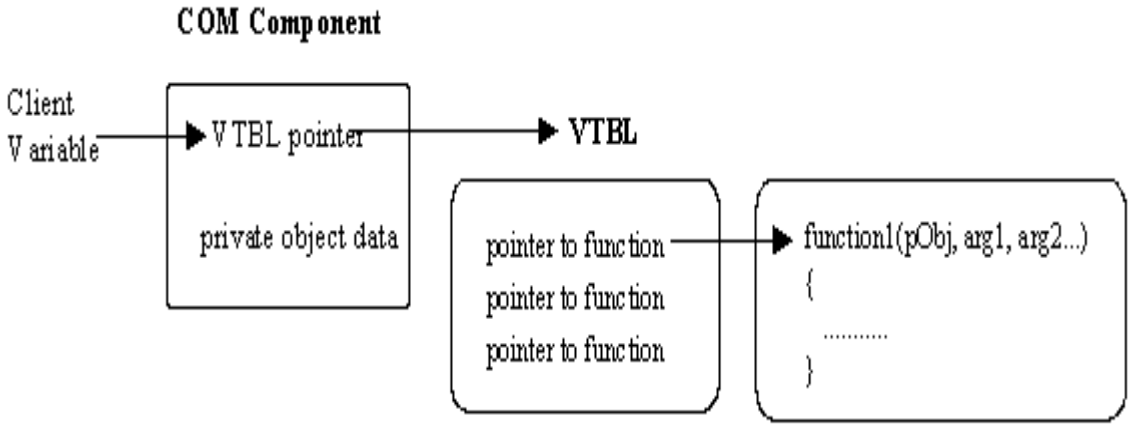
```
void Client(Service* s)
{
if (NewService* ns = dynamic_cast<NewService*>(s))
{
// use the new service interface
}                    }
```



COM -Component Object Model (Microsoft 1993)

- Inter Process Communication (IPC) için çeşitli programlama dilleri arasında kullanılır.
- COM platform bağımsız, dağıtık, nesneye yönelik bir sistemdir ve birbirleri ile ilişkide olan yazılım bileşenleri oluşturur.
- COM, nesnelerin içindekileri bilinmeden yeniden kullanımlarını sağlar.
- Implementasyondan bağımsız iyi tanımlanmış arayüzler verir.
- COM, Page Object Model (POM) un gelişmiş halidir.
  - Tüm bir sayfayı, tüm sayfa öğeleriyle etkileşimdeki yöntemlerle bir nesne olarak modellemek yerine, COM kullanıcı arayüzünü Buttons, Text fields, Dropdown menus, Search bars, Tables gibi farklı bileşenlere ayırır.
  - Her bileşen bir Java sınıfında kapsüllenir (encapsulated) ve etkileşimleri (davranışları) belirli yöntemlerle yönetilir; bu da her bir öğenin bağımsız olarak test edilmesine imkan sağlar (ve de maintainance).

Component Object Model -COM



- COM defines a standard way to lay out **virtual function tables** (vtables) in memory, and a standard way to call functions through the **vtables.**
- Any language that can call functions via pointers (C, C++, Smalltalk, Ada, and even BASIC) all can be used to write components that can interoperate with other components written to **the same binary standard**.
- Indirection (the client holds a pointer to a vtable) allows for **vtable** sharing among **multiple instances of the same object** class.
- On a system with hundreds of object instances, **vtable** sharing can reduce memory requirements

Button Component

```
public class ButtonComponent {

private WebDriver driver;
public ButtonComponent(WebDriver driver) {
this.driver = driver;                                    }

public void clickButton(String buttonText) {
WebElement button = driver.findElement(By.xpath("//button[text()='" + buttonText +
                                                                                "']"));
button.click();
}
}
```

Runner
An example of `TestRunner`

- The runner class is responsible for running the tests using JUnit or TestNG.
- The runner class configures Cucumber to load the test scenarios defined in the `.feature` files and run them using the step definitions.

```
@RunWith(Cucumber.class)
@CucumberOptions(
features = “src/test/resources/features”,
glue = “com.componentObjectModel.stepDefinitions”,
plugin = {“pretty”, “html:target/cucumber-reports.html”}
)
public class TestRunner {

}
```

Writing and Explaining a Gherkin Scenario

- One of the essential elements of automation with Cucumber is using the Gherkin language to write test scenarios.
- Gherkin allows us to describe tests in a readable and understandable way
- Let’s test the interaction with a button, using the `ButtonComponent`

Hhow it might be written in Gherkin:

Scenario: User clicks on the "Submit" button

Given I am on the login page

When I click on the button "Submit"

Then I should be redirected to the homepage

Explanation of the Scenario

Scenario: This scenario describes the action where a user clicks the ‘Submit’ button on the login page and ensures they are redirected to the homepage after clicking.

Given I am on the login page: The initial state of the test is that the user is on the login page.

When I click on the button “Submit”: The action performed in the test is clicking the ‘Submit’ button.

Then I should be redirected to the homepage: The expected verification is that the user is redirected to the homepage after clicking the button.

Interface Segregation (tekrar)

- When we define an interface that provides multiple methods, we generally break it down into multiple ones
  - Therefore, each one contains fewer methods
  - The basic idea of abstract base classes define a basic behavior or interface that some derived classes are responsible for implementing.
    - ✓Therefore certain methods are overridden
  - This principle also contains a virtual subclass.
- Interface Segregation:

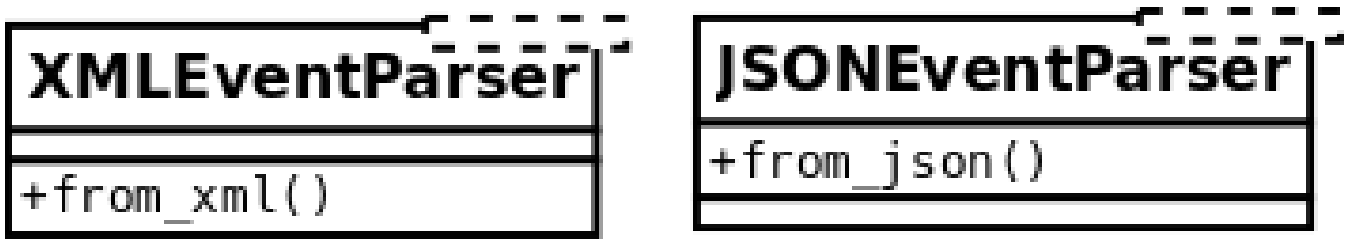
By separating interfaces into the smallest possible units, each class that wants to implement one of these interfaces will most likely be highly cohesive

- Therefore the code reusability is successed
- The new interface has a quite definite behavior and set of responsibilities.

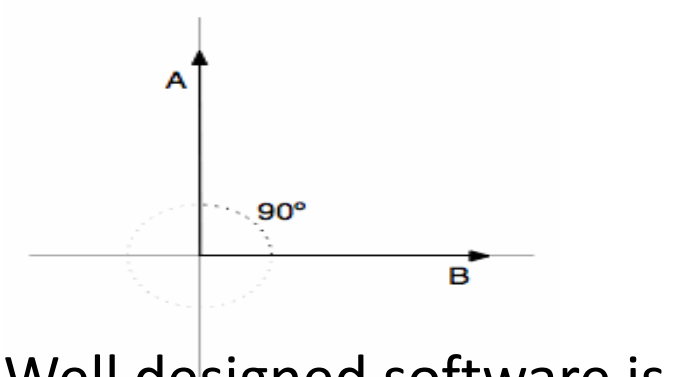
An Interface Providing too much

- We want to parse an event from several data sources, in different formats (XML and JSON, for instance).
  - we design an interface as our dependency instead of a concrete class
- What happens using this interface?
  - It is required to use an abstract base class and define the methods (from_xml() and from_json()) as abstract
    - ✓ The derived classes can implement them.
- But, there is a force while implementation
  - Events that derive from this abstract base class and implement these methods would be able to work with their corresponding types.
- if a particular class does not need the XML method, and can only be constructed from a JSON
  - It would still carry the from_xml() method from the interface
    - ✓ since it does not need it, it will have to pass.
  - This is not a flexible case
    - ✓ it creates coupling and forces clients of the interface to work with methods that they do not need.

The Smaller the Interface, the Better



- These are two independent functions, preserve the flexibility of the system
- Functionality can still be achieved by composing new smaller objects.
- Does this resemblance to the **Single Responsibility?**
  - No, these are interfaces
  - These is is an abstract definition of behavior.
  - There is no reason to change
    - ✓ There is nothing there until the interface is actually implemented.
  - Failure to comply with this principle will create an interface that will be coupled with **orthogonal functionality** (cohesion coupling)
    - ✓ This derived class will also fail to comply with the Single Responsibility (it will have more than one reason to change).



Cohesion and Coupling: Principles of Orthogonal, Object-Oriented Programming

- Well designed software is orthogonal.
- Each of its components can be altered without effecting other components
- Cohesion describes the focus of an individual software component.
  - When a component has only one responsibility, there is only one reason to change,
    - ✓ it has high cohesion
  - When a component has many responsibilities, a therefore many reasons to change,
    - ✓ it has low cohesion
- Coupling is the extent to which a component of your software depends on other components.
  - Loosely (low) coupled components have fewer dependencies than tightly (high)coupled components.
  - Modifying a tightly coupled component is difficult.
    - ✓ If a car object, boat object, and plane object are all tightly (high) coupled to an engine object, then a modification to the engine object will necessitate changes to the car, boat, and plane objects.

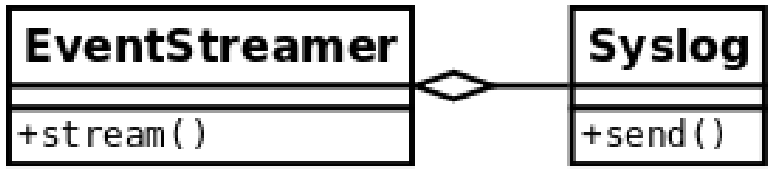
Dependency Inversion Principle (differents terms can be used)

- Dependency Inversion—Bağımlılıkların dönüşümü ilkesi

Bu ilkenin amacı herhangi bir şeyi somut (concrete) olmaktan ziyade soyut (abstract) olarak bağlamaktır (couple)

- Dependency Inversion Principle gerçekleşmesi için nesneler run time sırasında seçilir compile time sırasında gerçekleşmez.
  - Bir programın ya da program parçasının davranışı (behaviour) run time da değişir.
  - Somut bir nesnenin (concrete object) oluşunu new anahtar sözcüğü ile mümkündür (bir zincir içerisinde main() metodunda gerçekleşecektir).
- Dependency Injection—Bağımlılıkları tersine çevirme eylemi
- Constructor Injection— Constructor (yapıcı) aracılığıyla bağımlılık enjeksiyonu
- Parameter Injection- Performing dependency injection via the parameter of a method.

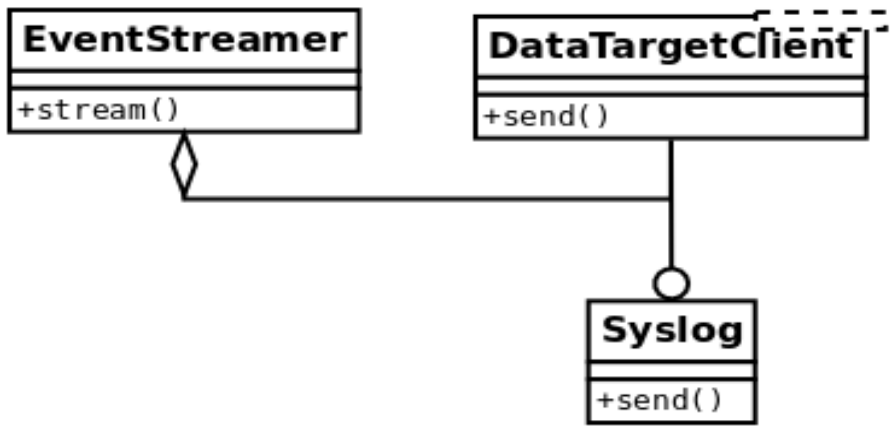
Dependency Inversion



- An event monitoring system delivers the identified events to a data collector to be further analyzed.
- An implementation of this idea would consist of having an event streamer class that interacts with a data destination, for example, Syslog
- This design is not very good.
  - we have a **high-level class** (EventStreamer) depending on a **low-level one** (Syslog is an implementation detail)
    - ✓ Since the composition rule and it is a concrete class.!!
  - If something changes in the way we want to send data to Syslog, EventStreamer will have to be modified.
  - If we want to change the data destination with a different one or add new ones at runtime, we are also in trouble
    - ✓ Because this change will constantly modify the stream() method to adapt it to these requirement

Inverting the Dependencies

- The solution to these problems is to make EventStreamer work **with an interface, rather than a concrete class.**
- This way, implementing this interface is up to the low-level classes that contain the implementation details



- There is an interface that represents a generic data target where data is going to be sent to.
- The dependencies have now been inverted
- EventStreamer does not depend on a concrete implementation of a particular data target,
  - it does not have to change in line with changes on this one
  - it is up (open) to every particular data target
  - it **implements** the interface **correctly** and **adapt to changes** if necessary.

The Comparison of Two Implementations

- First Implementation(before inverting Dependency: EventStreamer only worked with objects of type Syslog
  - This was not very flexible.
  - It was possible to work with any object that could respond to a send() message
    - ✓With the identification this method as the interface that it needed to comply with
- Second Implemention(after Inverting Dependency) : Syslog is actually extending the abstract base class named DataTargetClient
  - It defines the send() method.
  - it is possible to update every new type of data target (email, for instance) to extend this abstract base class and implement the send() method.

Dependency Injection ??

- Dependency Injection protect the code by making it independent of things that are out of our control (fragile).

Therefore: The code should not adapt to details or concrete implementation

it is required to force whatever the implementation

- Abstractions have to be organized in such a way that they do not depend on details.
  - Instead, details (concrete implementations) should depend on abstractions.

Dependency Injection (Tekrar)

- Dependency Injection (DI) is a concept in which objects receive their required dependencies from external sources rather than creating them internally.
- Dependency Injection (DI) is a design pattern used to implement IoC (Inversion of Control)
- DI allows a program design to follow the Dependency Inversion Principle.
- The primary purpose of DI is to manage dependencies between objects, making your code more modular, testable, and maintainable.

```
public class MessageSender {
    private MessageService messageService;
    public MessageSender() {  //tightly coupled with EmailMessageService
        this.messageService = new EmailMessageService();    // Direct instantiation       }
    public void sendMessage(String message) {
        messageService.send(message);     }
                                    }
//Changing to a different message service, such as an SMS service, would require modifying the MessageSender class itself.
```

Dependency Injection Approach

```
public class MessageSender {
    private MessageService messageService;

    // Dependency is injected through the constructor
    public MessageSender(MessageService messageService) {
        this.messageService = messageService;
    }
    public void sendMessage(String message) {
        messageService.send(message);
    }
}
```

The Importance of Dependency Injection

- **Decoupling of Components**: DI leads to **less tightly coupled code**. This decoupling means that components can be developed, tested, and maintained more **independently of each other**.
- **Enhanced Testability**:, it's more easy to swap out real services for mock objects in **unit tests**, which can lead to more thorough tests.
- **Flexibility and Scalability:** Applications tend to be more flexible and easier to scale. Changing or adding new functionality becomes simpler since the system's components are less interdependent.
- **Simplified Configuration and Management:** As dependencies can be centrally managed and injected at runtime, allowing for more dynamic behavior of applications.
- **Better Code Reusability**: By decoupling components, DI facilitates code reuse, allowing developers to leverage existing functionalities across different parts of an application or across multiple projects.