Thiết kế phần mềm -
KTPM3

DEPENDENCIES INJECTION

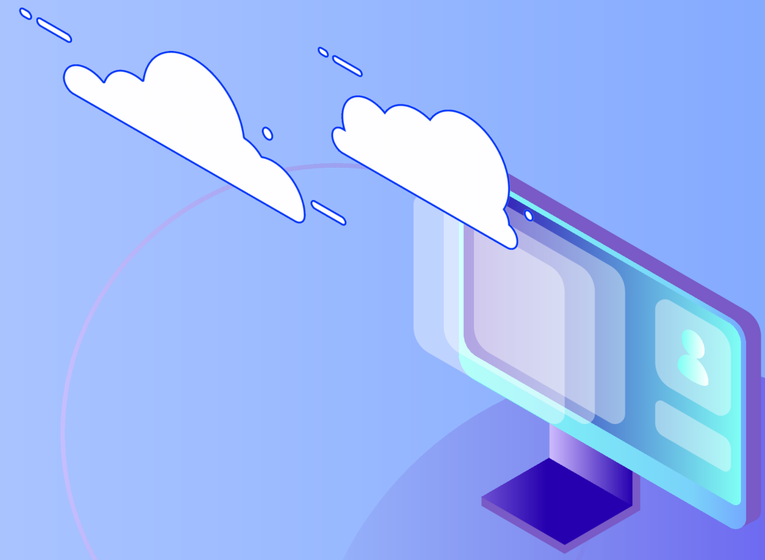
Đại học Khoa Học Tự Nhiên
Khoa Công nghệ Thông tin

Trần Nhật Huy
Nguyễn Quang Minh

1. Khái niệm DI
2. Các roles chính của DI
3. Phân loại
4. DI & SOLID Principle
5. Các loại framework phổ biến
6. Ưu điểm và nhược điểm

DEPENDENCIES INJECTION

Thiết kế phần mềm -
KTPM3

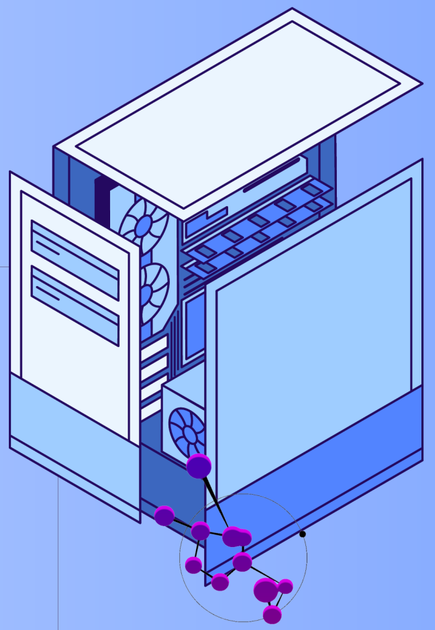


DEPENDENCIES INJECTION

LÀ GÌ ?
ĐƯỢC DÙNG LÀM GÌ ?

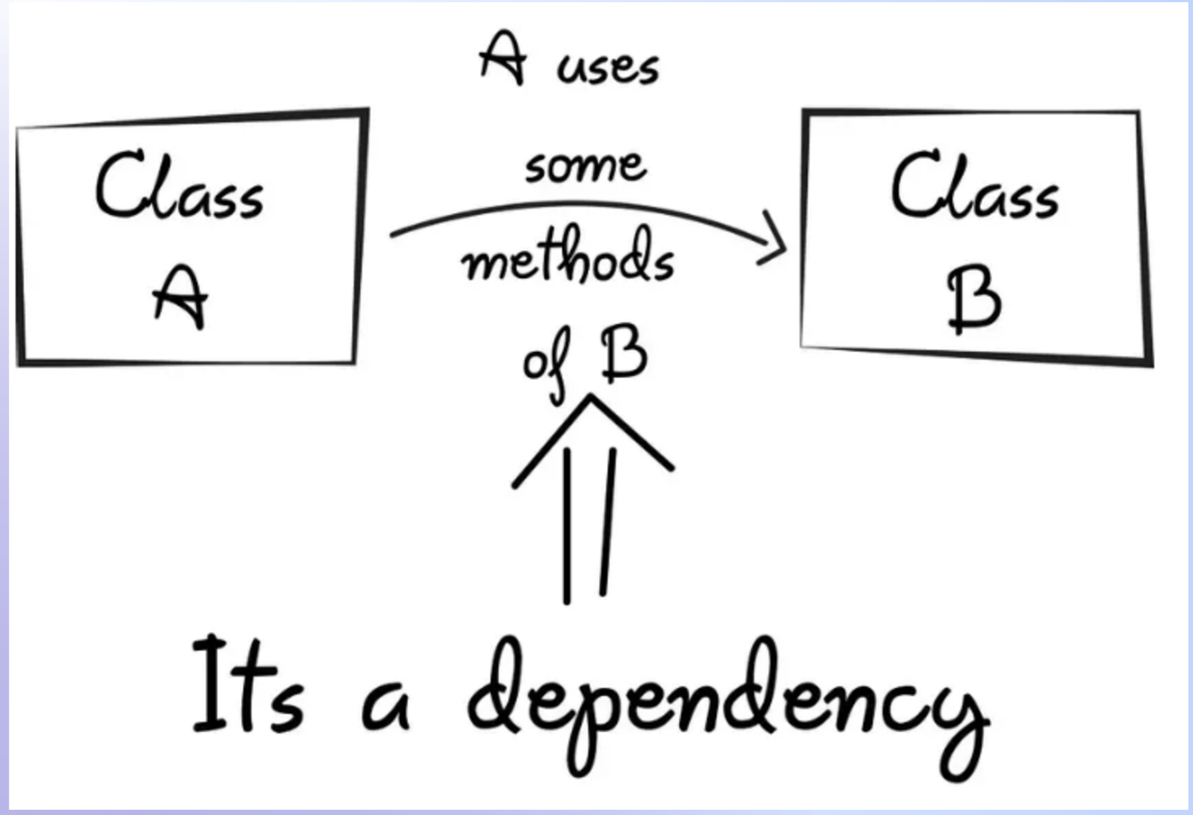
KHÁI NIỆM

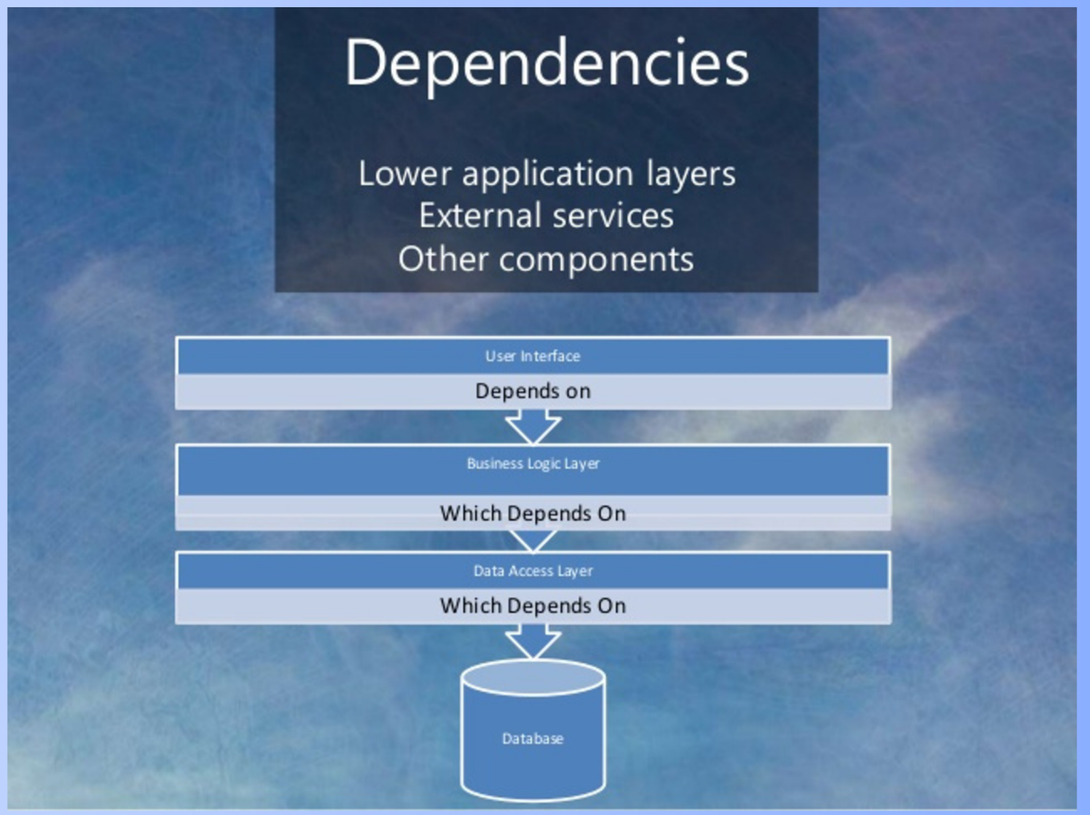
DEPENDENCIES INJECTION (DI)



Dendencies Injection là một **kỹ thuật** trong đó **đối tượng (object)** hoặc **hàm (function)** nhận và sử dụng các **objects và functions** khác **(services)**, không quan tâm đến việc chúng được khởi tạo như thế nào.

DEPENDENCY LÀ GÌ





VÍ DỤ

JAVA

```
class Car {
  private MCLWheel wheel = new MCLWheel();
  ...
}
```

VÍ DỤ

JAVA

```
class Car {
  private MCLWheel wheel;
  public Car(final MCLWheel wheel) {
   this.wheel = wheel;
  }
}
```

VÍ DỤ

JAVA

```
class MCLWheel{
   String name;
   ... //Constructor
   public void run() {
      ... running mcl wheel
   }
}
```

```
class Car {
  private MCLWheel wheel
  public Car(final MCLWheel wheel) {
   this.wheel = wheel;
  }
}
```

VÍ DỤ

JAVA

```
class MCLWheel                    {
  String name;
  ... //Constructor

  public void run() {
    ... running mcl wheel
  }
}
```

```
class WheelA                      {
  String name;
  ... //Constructor

  public void run() {
    ... running wheel A
  }
}
```

```
class WheelB                      {
  String name;
  ... //Constructor

  public void run() {
    ... running wheel B
  }
}
```

VÍ DỤ

JAVA

```java
interface Wheel{
  public void run();
}
```

```java
class MCLWheel                    {
  String name;
  ... //Constructor

  public void run() {
    ... running mcl wheel
  }
}
```

```java
class WheelA                      {
  String name;
  ... //Constructor

  public void run() {
    ... running wheel A
  }
}
```

```java
class WheelB                      {
  String name;
  ... //Constructor

  public void run() {
    ... running wheel B
  }
}
```

VÍ DỤ

JAVA

```java
interface Wheel{
  public void run();
}


class MCLWheel implements Wheel    {
  String name;
  ... //Constructor
  @Overide
  public void run() {
    ... running mcl wheel
  }
}
```

```java
class WheelA  implements Wheel  {
  String name;
  ... //Constructor
  @Overide
  public void run() {
    ... running wheel A
  }
}
```

```java
class WheelB  implements Wheel  {
  String name;
  ... //Constructor
  @Overide
  public void run() {
    ... running wheel B
  }
}
```

VÍ DỤ

JAVA

```
interface Wheel{
  public void run();
}
```

```
class Car {
  private Wheel wheel;

  public Car(Wheel wheel) {
    this.wheel = wheel;
  }

  public void run() {
    wheel.run();
}
```

Thiết kế phần mềm -
KTPM3

ROLES IN DEPENDENCIES INJECTION

- Services:
- Clients:
- Interfaces:
- Injector:

ROLES

SERVICE & CLIENT

Service:

- **Service** là thành phần cung cấp chức năng cho **Client**
- **Service** không cần biết ai đang sử dụng nó
- Có thể có **nhiều implementation khác nhau** của một Service, miễn là chúng tuân theo cùng một interface.

Thiết kế phần mềm - KTPM3

ROLES

SERVICE & CLIENT

Client:

- **Client** là thành phần cần một **Service** để hoạt động.
- **Client** không tự tạo **Service**, mà nhận Service thông qua **Dependency Injection**

VÍ DỤ

JAVA

```java
class MCLWheel {
  String name;
  public void run() {
    // running mcl wheel
  }
}

class Car {
  private MCLWheel wheel;

  public Car(MCLWheel wheel) {
   this.wheel = wheel;
  }
}
```

VÍ DỤ

JAVA

```java
class MCLWheel {
  String name;
  public void getName() {
    return this.name;
  }
}

class Car {
  private MCLWheel wheel;

  public Car(MCLWheel wheel) {
   this.wheel = wheel;
  }
}
```

```java
class User {
  private Car userCar;
  public User (Car car) {
   this.car = car;
  }
```

ROLES

INTERFACE

- Interface là **abstraction** để tách biệt Client khỏi **Implementation details** của Service
- Giúp dễ dàng thay thế **Service** khác mà không cần Client

VÍ DỤ

JAVA

```
interface Wheel{
  public void getName();
}


public class Main {
    public static void main(String[] args) {
        Wheel myWheel = new MCLWheel();
        Car myCar = new Car(myWheel);
        ...
    }
}
```

VÍ DỤ

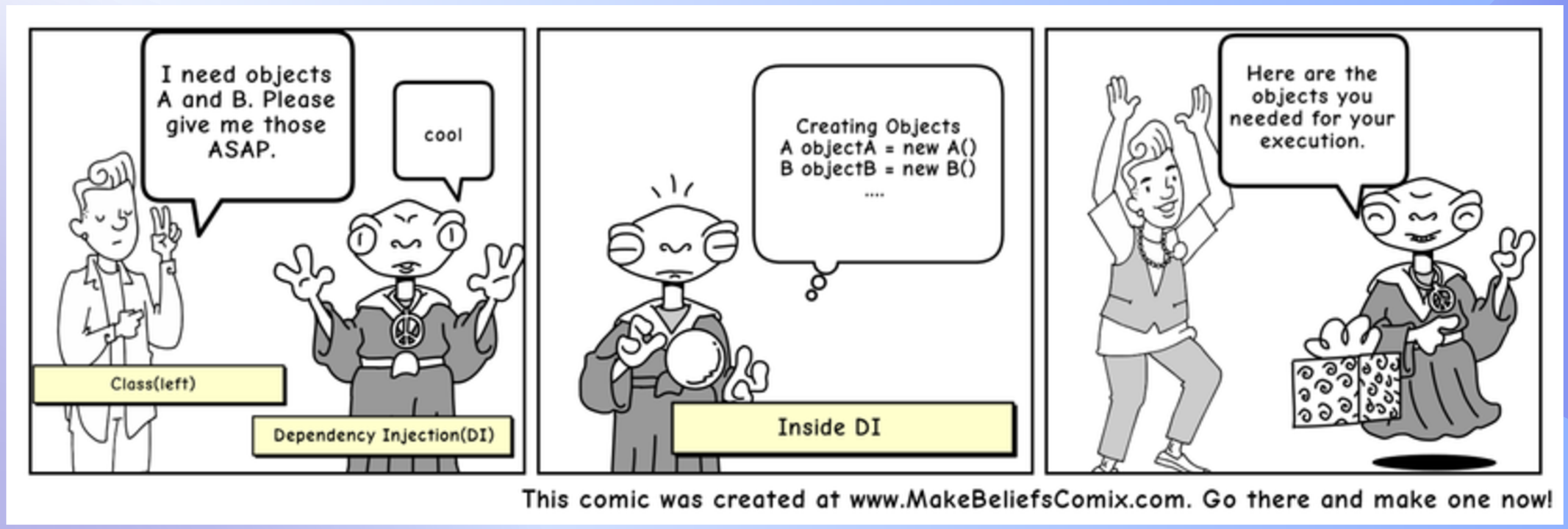
JAVA

```
interface Wheel{
  public void getName();
}


public class Main {
    public static void main(String[] args) {
        Wheel myWheel1 = new WheelA();
        Car myCar = new Car(myWheel1);
        ...
    }
}
```

ROLES

INJECTOR

Thiết kế phần mềm - KTPM3



INJECTOR

Thiết kế phần mềm - KTPM3

Wikipedia

Injectors [edit]

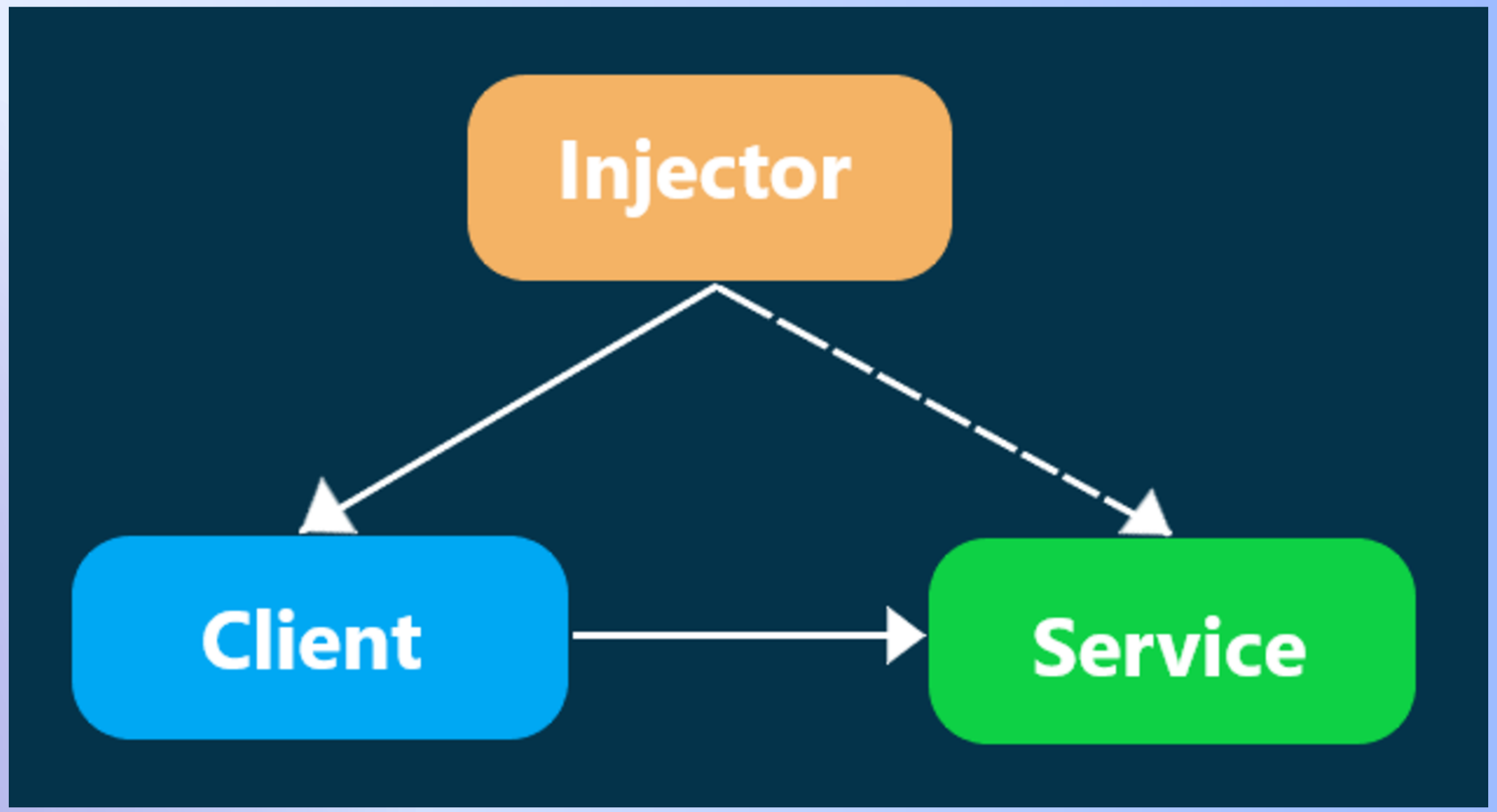
The **injector**, sometimes also called an assembler, container, provider or factory, introduces services to the client.

The role of injectors is to construct and connect complex object graphs, where objects may be both clients and services. The injector itself may be many objects working together, but must not be the client, as this would create a circular dependency.

Because dependency injection separates how objects are constructed from how they are used, it often diminishes the importance of the `new` keyword found in most object-oriented languages. Because the framework handles creating services, the programmer tends to only directly construct value objects which represents entities in the program's domain (such as an `Employee` object in a business app or an `Order` object in a shopping app).[13][14][15][16]

INJECTOR

Thiết kế phần mềm - KTPM3



“Injector như một container”

Manual **Framework**

```
public class UserServiceContainer {
    private WheelA wheelA = new WheelA();
    private Car userCar = new Car(wheelA);
    public User user = new User(userCar);
}
```

Manual

Thiết kế phần mềm -
KTPM3

PHÂN LOẠI

DEPENDENCY INJECTION

4

LOẠI “TIỆM”

PHÂN LOẠI DEPENDENCY INJECTION

4

LOẠI "TIÊM"

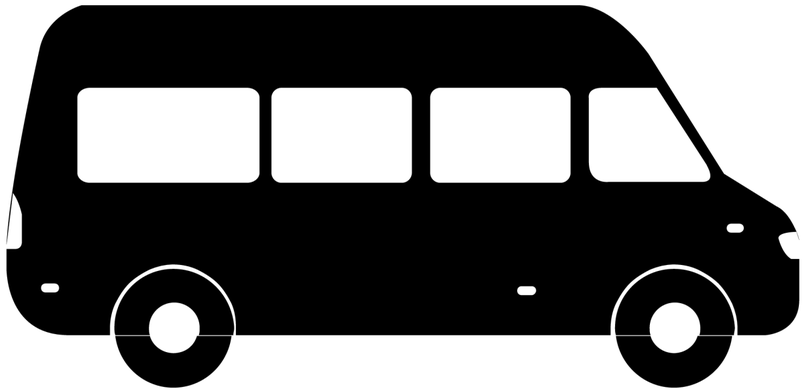
CONSTRUCTOR
INJECTION

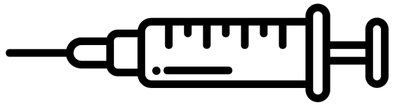
METHOD
INJECTION

SETTER
INJECTION

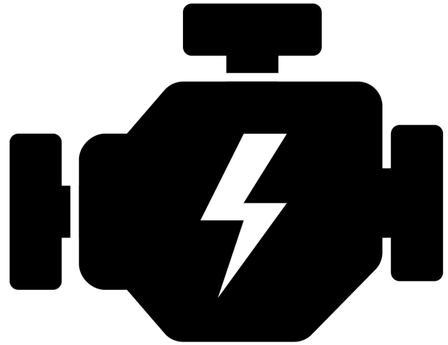
INTERFACE
INJECTION

CLIENT

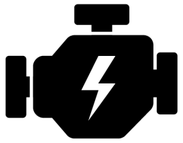




DEPENDENCY



CONSTRUCTOR



Tiêm Dependency vào Client thông qua Constructor khi khởi tạo Client .

CONSTRUCTOR INJECTION

Dependency - Hardcode

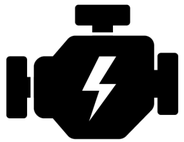
```
public class Car
{
    private IEngine engine;

    public Car()
    {
        // Hardcode dependency bên trong constructor
        engine = new DieselEngine();
    }

    public void StartCar()
    {
        engine.Start();
    }
}
```

Client - Constructor Injection

```
public class Car
{
    private IEngine engine;

    public Car(IEngine engine)
    {
        this.engine = engine;
    }

    public void StartCar()
    {
        engine.Start();
    }
}
```

```
var car = new Car(new DieselEngine());
var car_1 = new Car(new EletricEngine());
var car_2 = new Car(new HybridEngine());
```

METHOD



Tiêm Dependency vào Client thông qua các phương thức.

Thiết kế phần mềm - KTPM3

METHOD INJECTION

Client - Constructor Injection

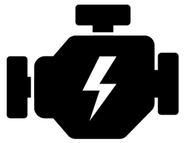
```
public class Car
{
    private IEngine engine;

    public Car(IEngine engine)
    {
        this.engine = engine;
    }

    public void StartCar()
    {
        engine.Start();
    }
}
```

Client - Method Injection

```
public class Car
{
    private IEngine engine;

    public void StartCarWithEngine(IEngine engine) {
        this.engine = engine;
        engine.Start();
    }

    public void StartCar()
    {
        engine.Start();
    }
}
```

```
var car = new Car();
car.StartCarWithEngine(new DieselEngine());
car.StartCarWithEngine(new HybridEngine());
```

SETTER



Tiêm Dependency vào Client thông qua Setter Properties của đối tượng.

SETTER INJECTION

Client - Method Injection

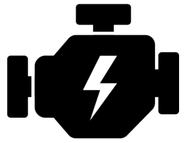
```
public class Car
{
    private IEngine engine;

    public void StartCarWithEngine(IEngine engine) {
        this.engine = engine;
        engine.Start();
    }

    public void StartCar()
    {
        engine.Start();
    }
}
```

Client - Setter Injection

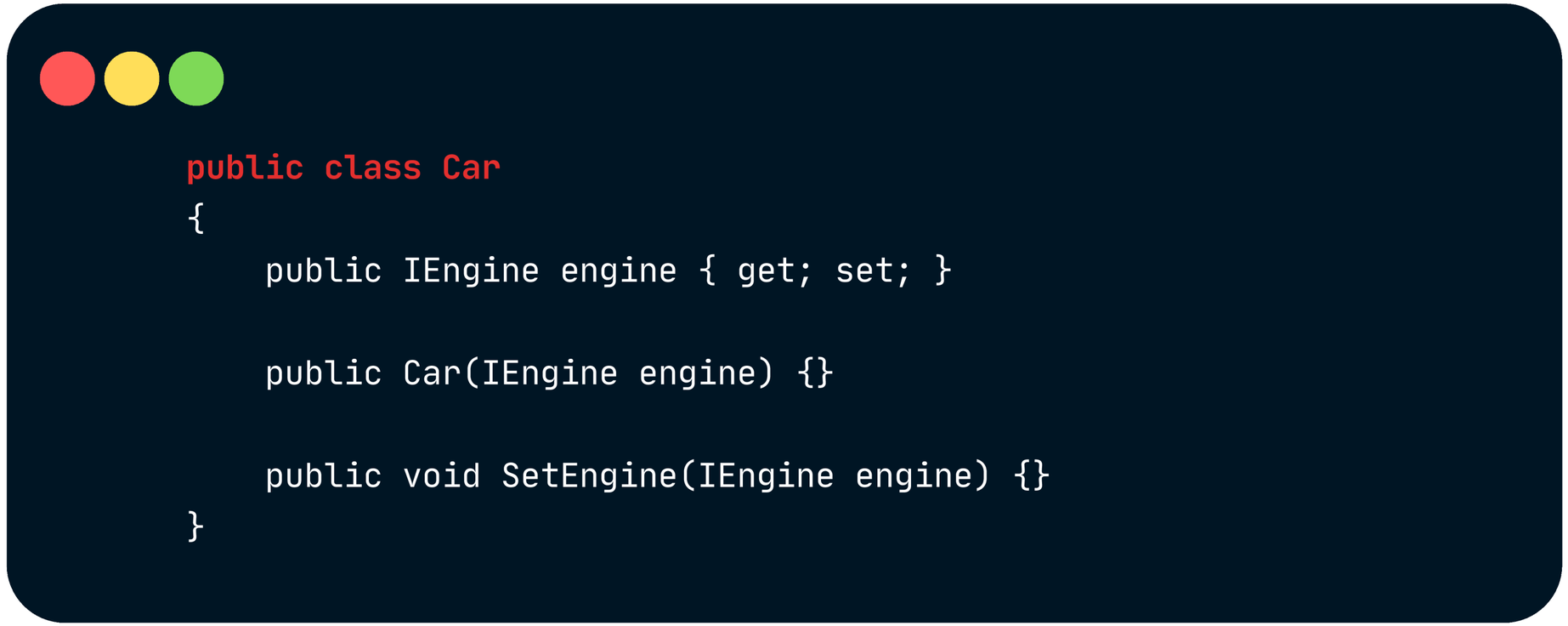
```
public class Car
{
    public IEngine engine { get; set; }

    public void StartCar()
    {
        if (engine == null) {
            Console.WriteLine("Engine is not set!");
        }
        else
        {
            engine.Start();
        }
    }
}
```

```
var car = new Car();
car.engine = new DieselEngine();
```

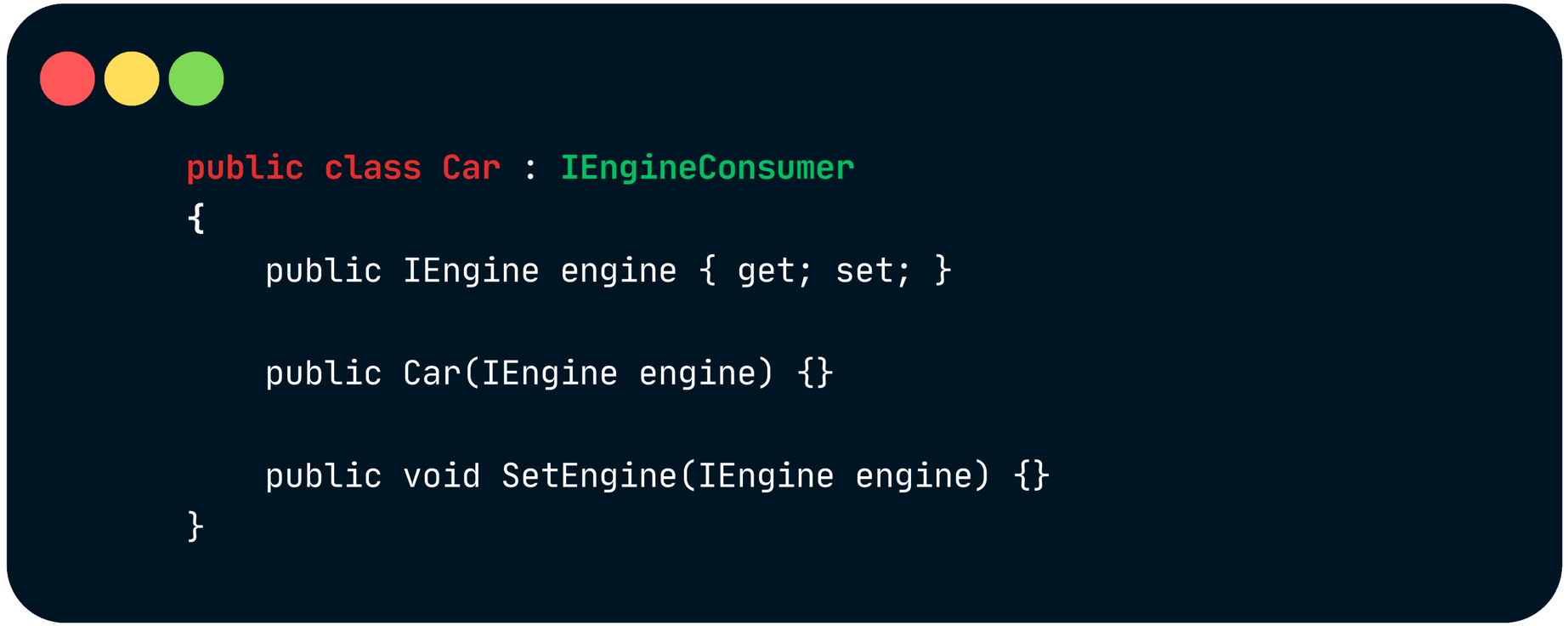
INTERFACE



Tiêm Dependency vào Client thông qua việc cài đặt một interface có phương thức để nhận các phụ thuộc.



```
public class Car
{
    public IEngine engine { get; set; }

    public Car(IEngine engine) {}

    public void SetEngine(IEngine engine) {}
}
```



```
public class Car : IEngineConsumer
{
    public IEngine engine { get; set; }

    public Car(IEngine engine) {}

    public void SetEngine(IEngine engine) {}
}
```

Client : Interface

Có một method có thể đưa *Dependency* vào

INTERFACE INJECTION

```
public interface IEngineConsumer
{
    void SetEngine(IEngine engine);
}
```

INTERFACE INJECTION

```
public class Car : IEngineConsumer
{
    public IEngine engine;

    public void SetEngine(IEngine engine) {
        this.engine = engine;
    }
}


Car car = new Car();
car.SetEngine(new DieselEngine());
```

CONSTRUCTOR

Inject qua hàm tạo constructor

SETTER

Inject qua property (get; set;)

METHOD

Inject qua tham số của một method cụ thể

INTERFACE

Client triển khai 1 interface chứa hàm SetDependency()

Thiết kế phần mềm - KTPM3

DEPENDENCIES INJECTION VIDEO

https://www.youtube.com/watch?v=J1f5b4vcxCQ&t=109s

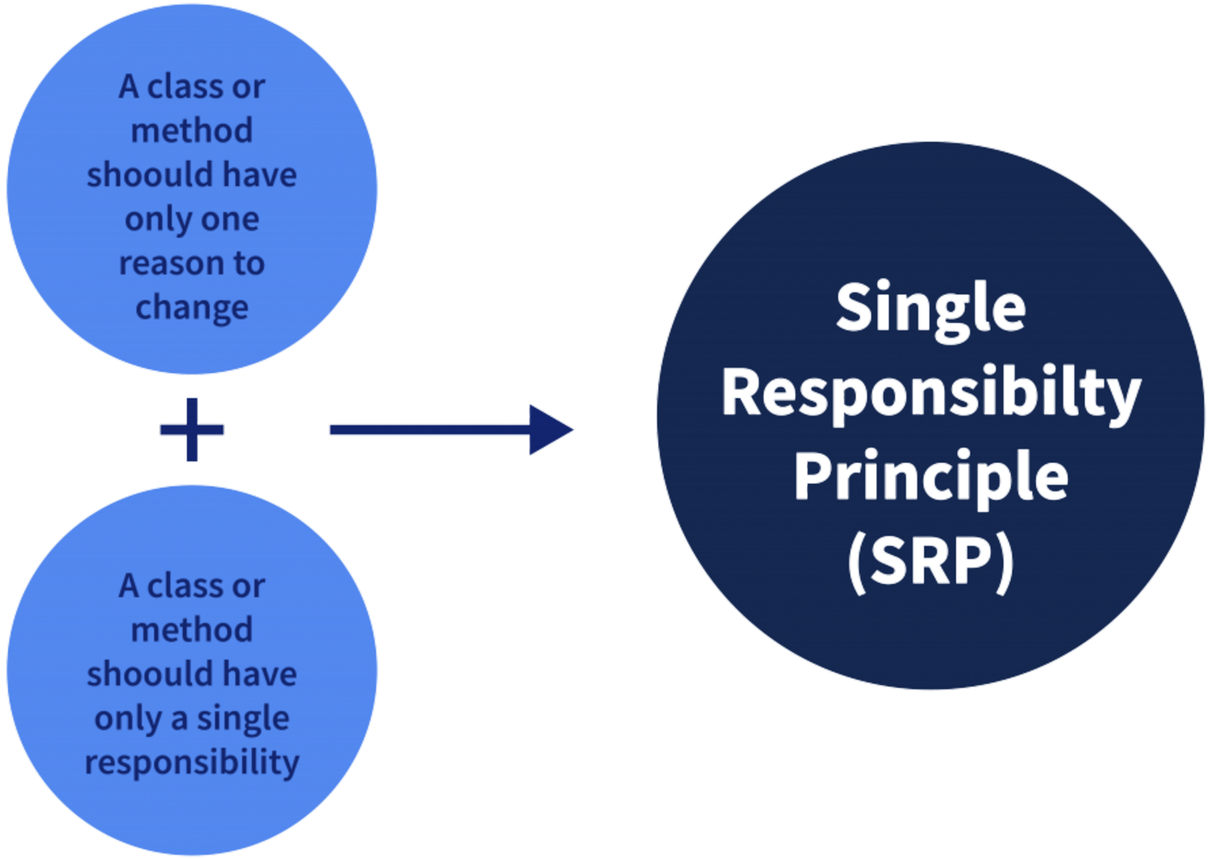
Thiết kế phần mềm -
KTPM3

DI & SOLID

S O L I D

ingle Responsibility Principle

Nguyên tắc Trách nhiệm Đơn: Mỗi lớp (hoặc phương thức) chỉ nên nhận một trách nhiệm duy nhất



```
public class Student {
    public string Name { get; set;}
    public int Age { get; set;}

    // Format class này dưới dạng text
    public string GetStudentInfoText() {
         return "Name: " + Name + ". Age: " + Age;
    }
    // Lưu trữ xuống database, xuống file
    public void SaveToJSONFile() {
        dbContext.Save(this);
    }
    public void SaveToCSVFile() {
        Files.Save(this, "fileName.txt");
    }
}
```

```
public class Student {
    public string Name { get; set;}
    public int Age { get; set;}
}
```

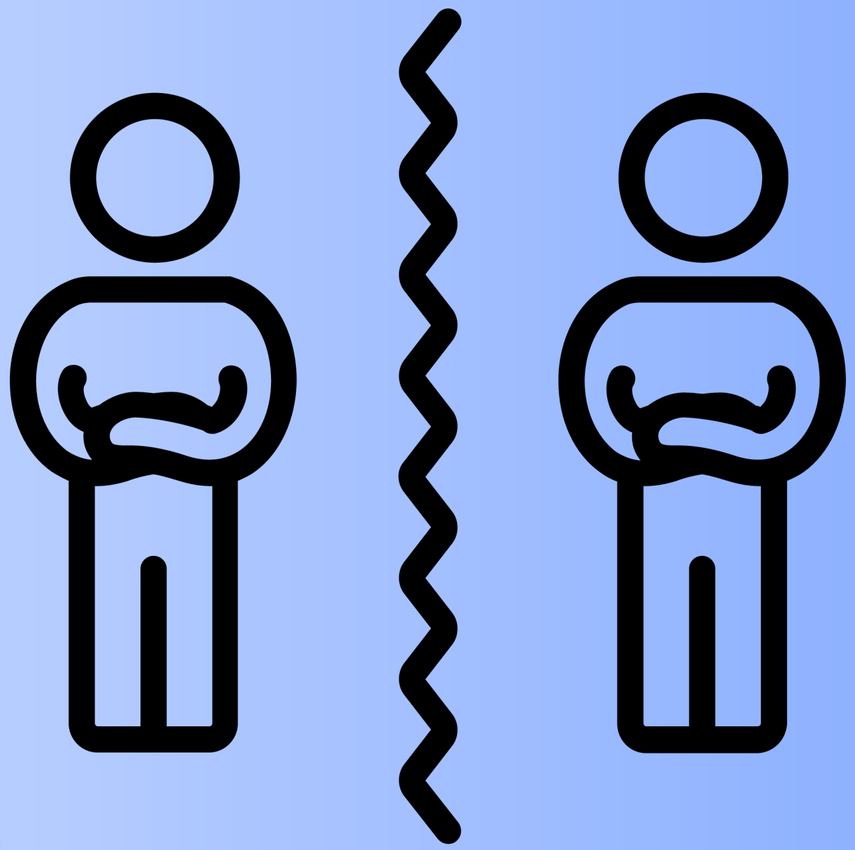
```
// Class này chỉ format thông tin hiển thị student
public class Formatter {
    public string FormatStudentText(Student std) {
      return "Name: " + std.Name + ". Age: " + std.Age;
    }
}
```

```
// Class này chỉ lo việc lưu trữ
public class Store {
    public void SaveToJSONFile(Student std) {
      dbContext.Save(std);
    }
    public void SaveToCSVFile(Student std) {
      Files.Save(std, "fileName.txt");
    }
}
```

DI & SINGLE RESPONSIBILITY PRINCIPLE

1 Single Responsibility

Dependency injection (DI) promotes the Single Responsibility Principle (SRP) by allowing a class to focus solely on its core functionality. When you use DI, you delegate the task of creating dependencies to an external entity, typically an injector or a container. This means that your class doesn't need to manage the creation or configuration of the objects it depends on, which aligns perfectly with SRP's advocacy for classes to have only one reason to change.



S

O

L

I

D

pen-Closed Principle

Nguyên tắc mở đóng: Một module nên mở rộng được nhưng đóng lại với việc chỉnh sửa.

DI & OPEN-CLOSED PRINCIPLE

Nội dung nguyên lý

OCP yêu cầu code entities phải có tính “OPEN” cho việc mở rộng code và “CLOSE” cho việc thay đổi.

DI & OCP

Cung cấp các interfaces, injector giúp clients không cần quan tâm sự thay đổi của service, kể cả mở rộng

S

O

L

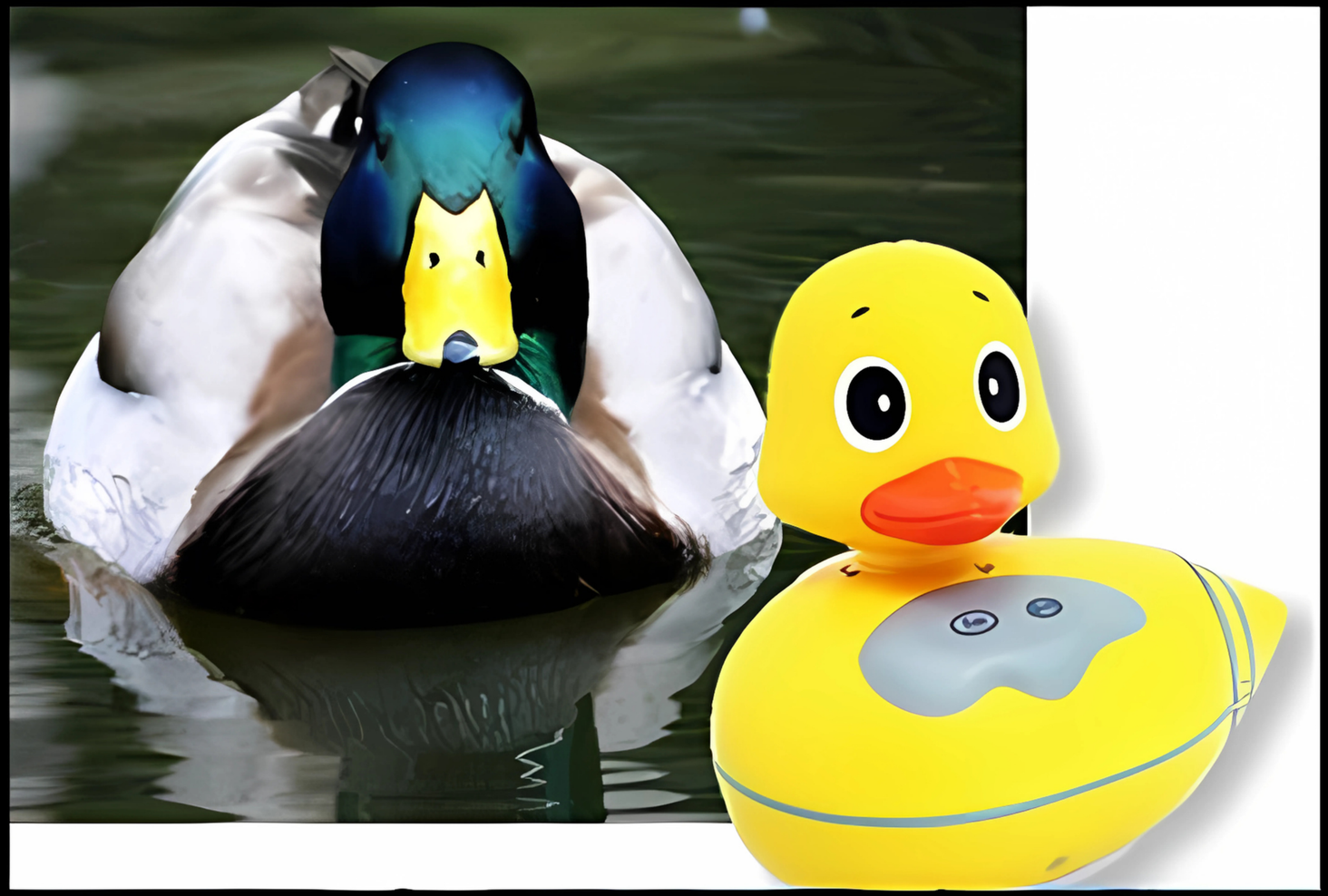
I

D

iskov Substitution Principle

Nguyên tắc thay thế Liskov: Các đối tượng của lớp con nên có khả năng thay thế các đối tượng của lớp cha mà không làm thay đổi tính đúng đắn của chương trình

DI
&



Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries,
you probably have the wrong abstraction.

LISKOV SUBSTITUTION PRINCIPLE

Nội dung nguyên lý

Trong một chương trình, các object của class con có thể thay thế class cha mà không làm thay đổi tính đúng đắn của chương trình

Giải thích nguyên lý

Để giữ tính đúng đắn của chương trình, class con phải thay thế được class cha

LISKOV SUBSTITUTION PRINCIPLE

Ví dụ minh hoạ

```
public class Bird {
  public virtual void Fly() { Console.Write("Fly"); }
}
public class Eagle : Bird {
  public override void Fly() { Console.Write("Eagle Fly"); }
}
public class Duck : Bird {
  public override void Fly() { Console.Write("Duck Fly"); }
}
```

```
public class Penguin : Bird {
  public override void Fly() { throw new NoFlyException(); }
}


var birds = new List { new Bird(), new Eagle(), new Duck(), new Pengui
foreach(var bird in birds) bird.Fly();
// Tới pengiun thì lỗi vì cánh cụt quăng Exception
```

LISKOV SUBSTITUTION PRINCIPLE

Ví dụ minh hoạ

```
public class Rectangle
{
    public int Height { get; set; }
    public int Width { get; set; }

    public virtual void SetHeight(int height)
    {
        this.Height = height;
    }
    public virtual void SetWidth(int width)
    {
        this.Width = width;
    }
    public virtual int CalculateArea()
    {
        return this.Height * this.Width;
    }
}
```

```
public class Square : Rectangle
{
    public override void SetHeight(int height)
    {
        this.Height = height;
        this.Width = height;
    }

    public override void SetWidth(int width)
    {
        this.Height = width;
        this.Width = width;
    }
}
```

LISKOV SUBSTITUTION PRINCIPLE

Cách khắc phục

- Tạo ra các interface cho method

```
Interface Flyable() {
    public void fly();
}
```

```
Class Eagle extends Bird implements Flyable {
 @Overide
 public void fly() {
    System.out.println(“Eagle is flying”);
}
```

```
Class Penguin extends Bird {
    ......
}
```

LISKOV SUBSTITUTION PRINCIPLE

Cách khắc phục

- Tạo ra một abstract class tổng quát hơn

```
Class Shape() {

}
```

```
Class Rectangle extends Shape {
  public void setSides(int s1, int s2) {
    this.width = s1;
    this.height = s2;
}
```

```
Class Square extends Shape {
   public void setSides(int s) {
    this.side = s;
   }
}
```

DI & LISKOV SUBSTITUTION PRINCIPLE

- DI hỗ trợ developer “care” hơn đến abstraction (interface)

- DI giúp developer dễ nhận biết bug liên quan đến LSP hơn ở runtime

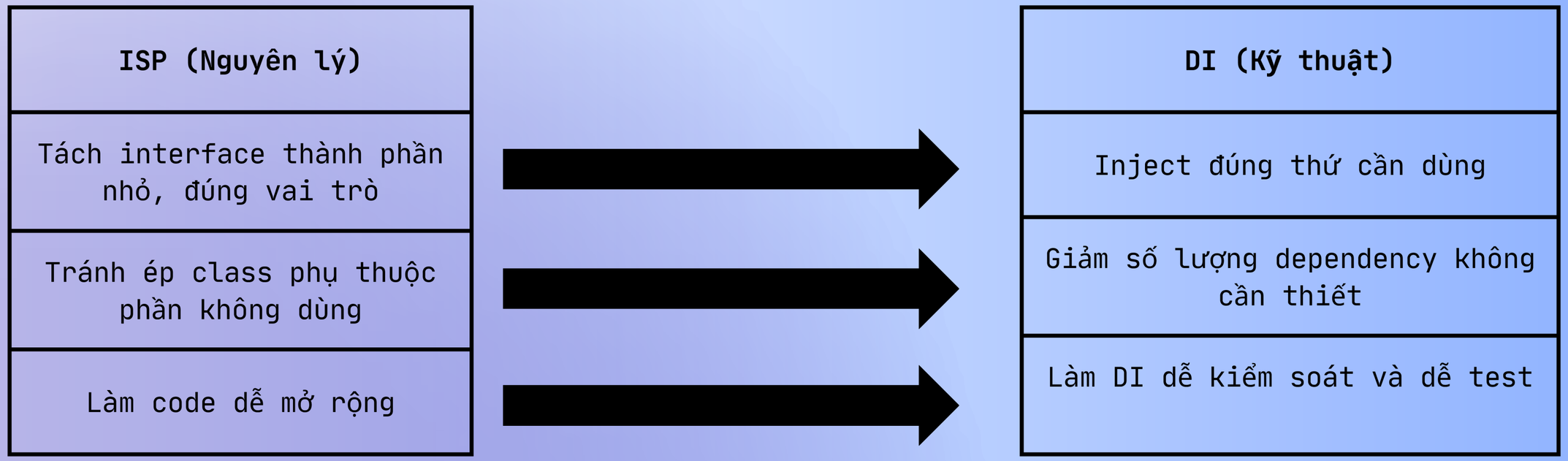
S O L I D

nterface Segregation Principle

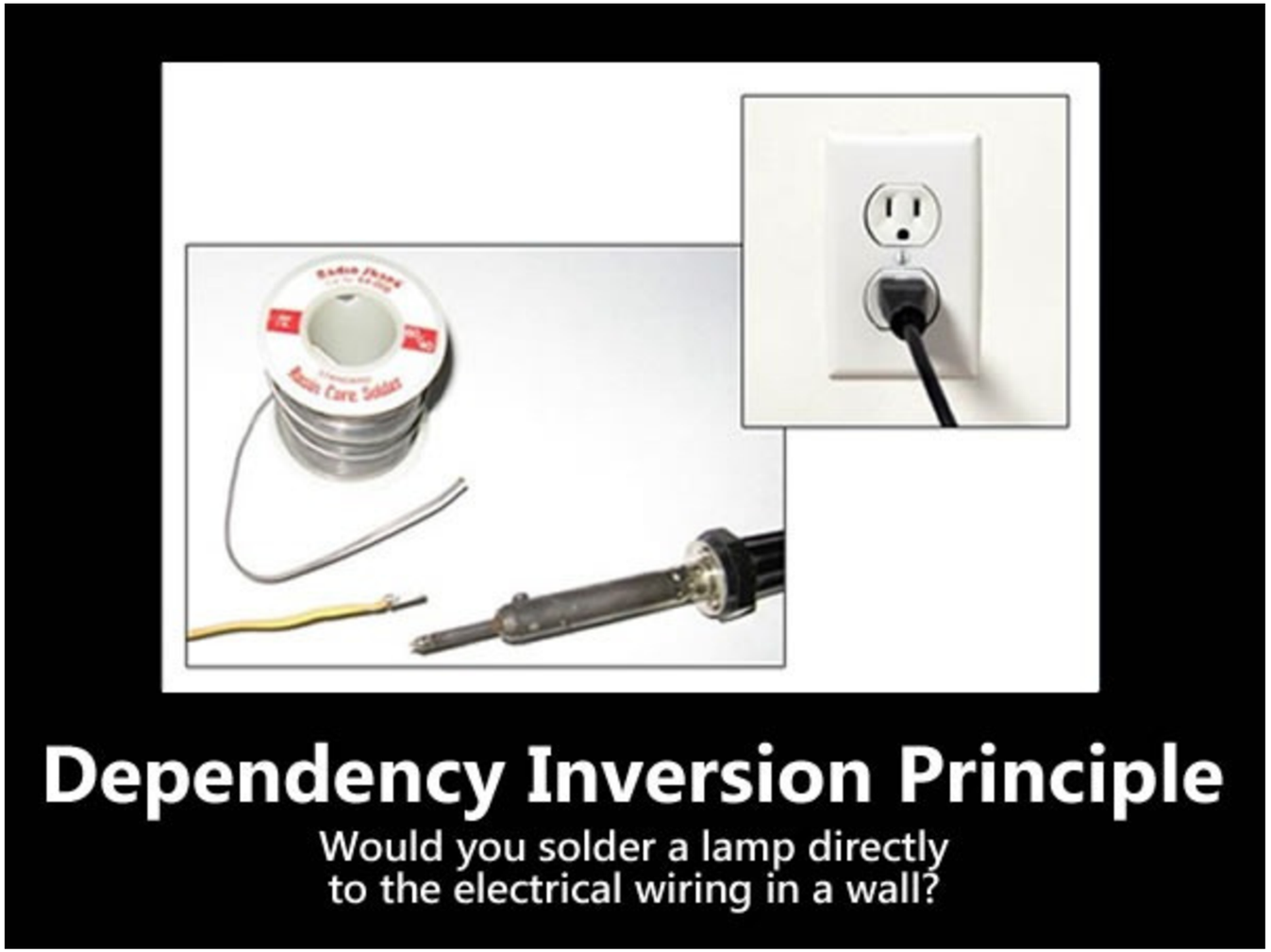
Nguyên tắc Phân Tách Giao Diện: Chia nhỏ giao diện thành các giao diện chuyên biệt và phù hợp với nhu cầu sử dụng.

DI & INTERFACE SEGREGATION PRINCIPLE

ISP và DI tương hỗ nhau trong clean code



S O L I



Dependency Inversion Principle

Nguyên tắc Đảo Ngược Phụ Thuộc: Các module cấp cao không nên phụ thuộc vào các module cấp thấp.

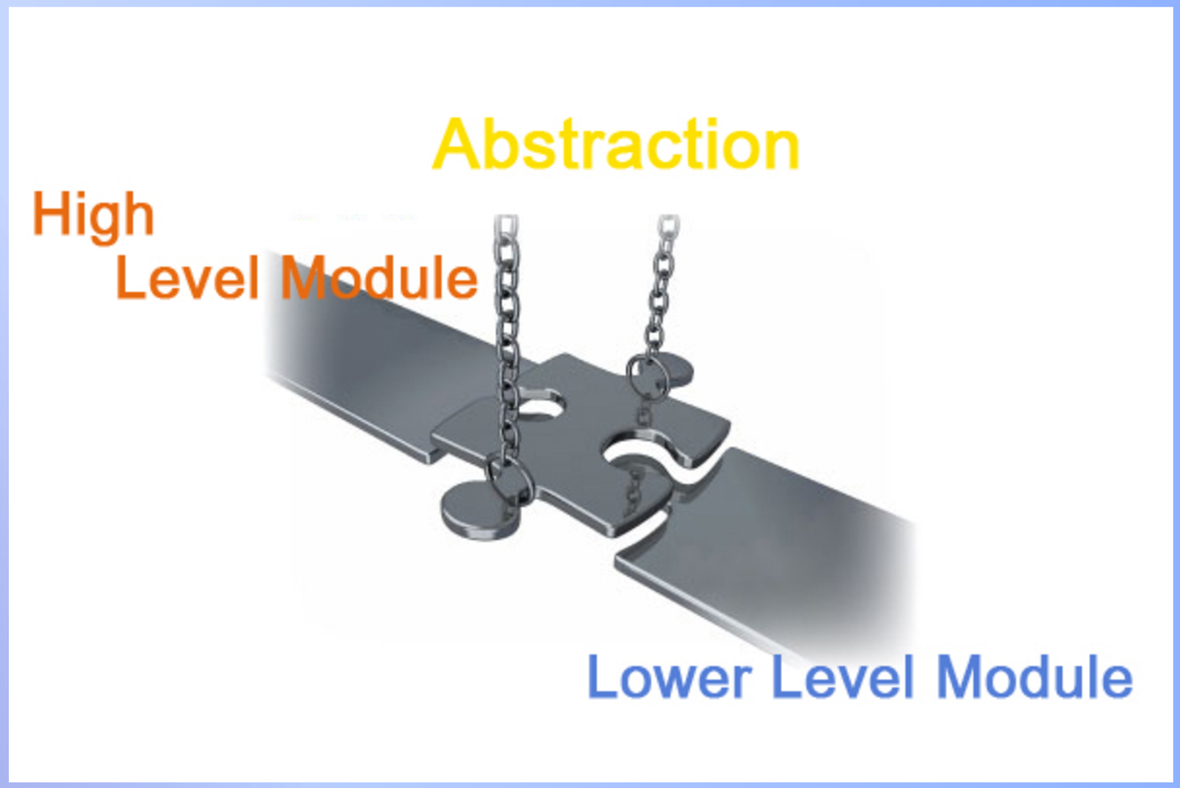
DEPENDENCY INVERSION PRINCIPLE

Nội dung nguyên lý

1. Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả 2 nên phụ thuộc vào abstraction.

2. Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại.

DEPENDENCY INVERSION PRINCIPLE



DI VS. DIP

DEPENDENCY INVERSION PRINCIPLE

- Design principle (in SOLID)

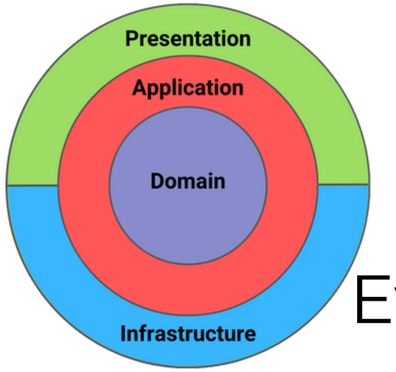
DEPENDENCY INJECTION

- Programming technique (implementation mechanism)

INVERSION OF CONTROL

Delegate

ServiceLocator



Event

MỘT SỐ DEPENDENCY INJECTION FRAMEWORK PHỔ BIẾN

JAVA SPRING

Spring dùng IoC Container để:

- Tự động quản lý vòng đời bean
- Inject các Bean vào nhau dựa trên annotation

```
@Component
public class EmailService {
    public void send(String msg) {
        System.out.println("Email: " + msg);
    }
}
```

```
@Service
public class UserService {
    @Autowired
    private EmailService emailService;

    public void register(String name) {
        emailService.send("Welcome " + name);
    }
}
```

JAVA SPRING

Một số annotation

Annotation	Ý nghĩa
@Component	Bean tổng quát
@Service	Bean chứa logic nghiệp vụ
@Repository	Bean truy xuất dữ liệu
@Controller	Bean dùng xử lý web
@Autowired	Tiêm Bean vào nơi cần dùng
@RestController	Tạo API REST
@SpringBootApplication	Đánh dấu class main khởi động app

JAVA ANDROID

Dagger Hilt

- Hilt được xây dựng dựa trên Dagger 2 để đơn giản hóa việc inject phụ thuộc vào các thành phần của Android

JAVA ANDROID

Mục tiêu của Hilt

- Tự động hóa việc khởi tạo và truyền các dependency
- Quản lý vòng đời dependency
- Tích hợp tốt với Android Jetpack (Lifecycle, ViewModel, Navigation)

JAVA ANDROID

Ví dụ:

```
public class AuthService {
    @Inject
    public AuthService() {
        ....
    }
}
```

```
@AndroidEntryPoint
public class LoginActivity extends AppCompatActivity {
    @Inject
    AuthService authService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        authService.login();
    }
}
```

JAVA ANDROID

Các annotation trong Hilt

Annotation	Ý nghĩa
@HiltAndroidApp	Bean tổng quát
@AndroidEntryPoint	Cho phép inject vào Activity,...
@Inject	Cho phép Hilt tạo dependency
@Module + @Provides	Cấu hình cách tạo các object
@HiltViewModel	Inject vào ViewModel
...	...

Microsoft.Extensions.DependencyInjection

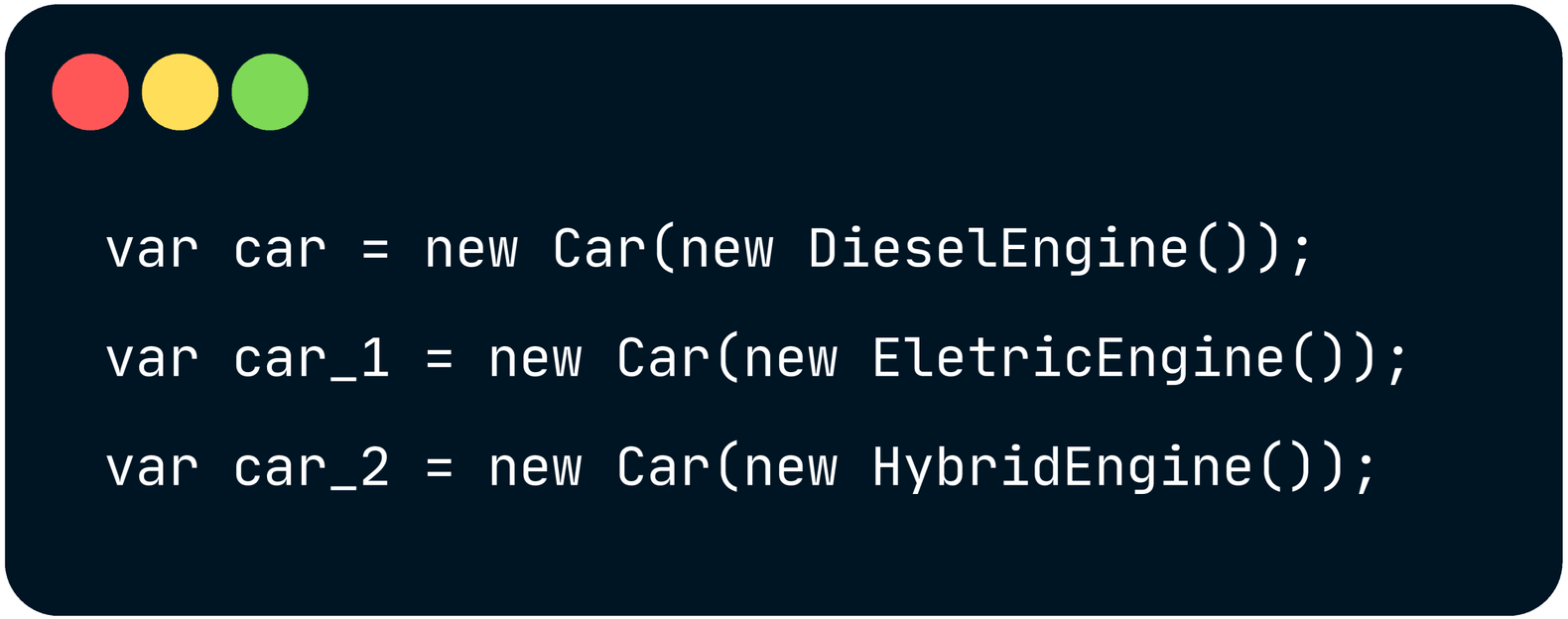
services.AddTransient<IEngine, DieselEngine>();	Đăng ký interface IEngine với implementation DieselEngine (vòng đời transient).
services.AddTransient<Car>();	Đăng ký lớp Car vào container (vòng đời transient).
var provider = services.BuildServiceProvider();	Xây dựng DI Container (ServiceProvider) từ các đăng ký đã thực hiện.
var car = provider.GetService<Car>();	Resolve đối tượng Car từ container và tự động inject dependency đã đăng ký vào constructor.

ƯU ĐIỂM VÀ NHƯỢC ĐIỂM

ƯU ĐIỂM VÀ NHƯỢC ĐIỂM

A. ƯU ĐIỂM

1, Giảm sự phụ thuộc chặt chẽ (Loose Coupling)

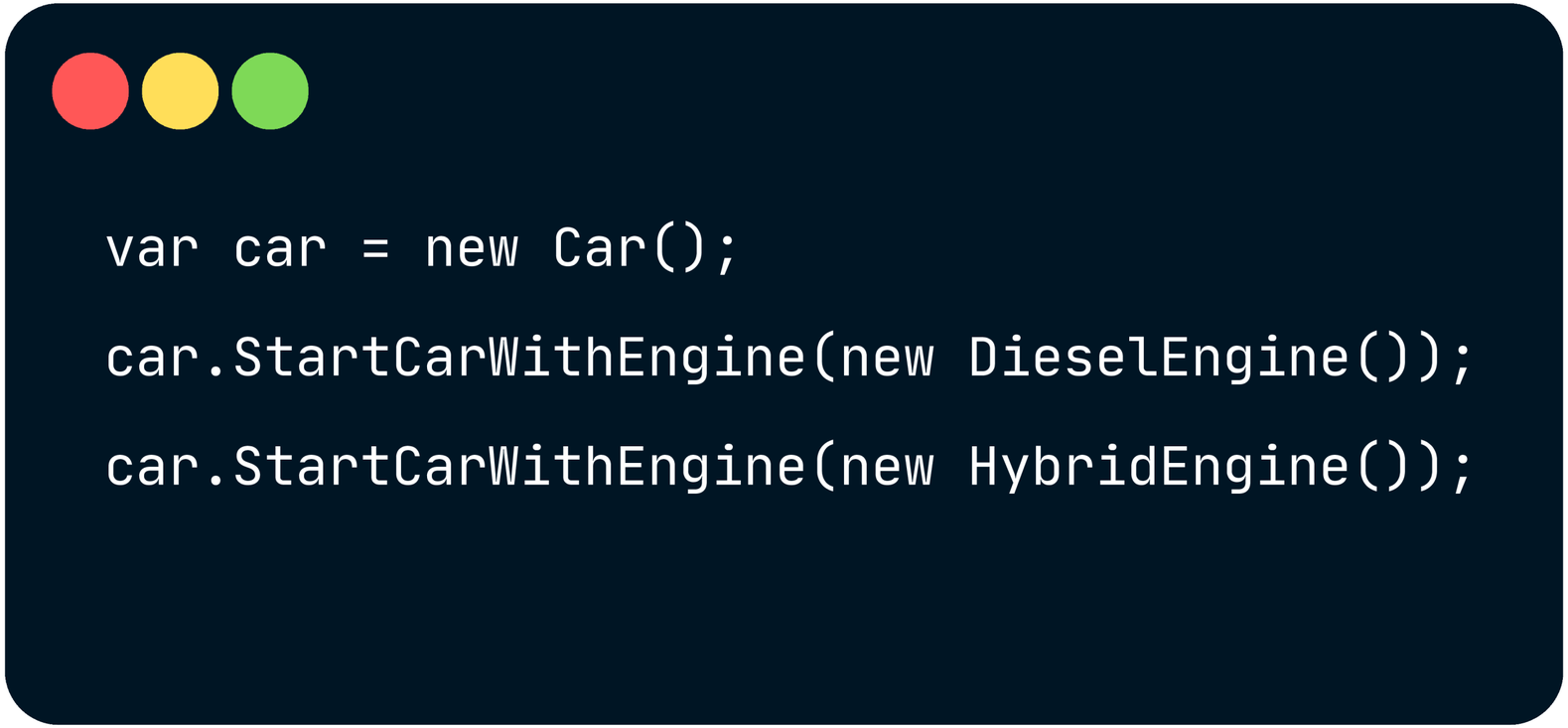


```
var car = new Car(new DieselEngine());

var car_1 = new Car(new EletricEngine());

var car_2 = new Car(new HybridEngine());
```

ƯU ĐIỂM VÀ NHƯỢC ĐIỂM

A. ƯU ĐIỂM

2, Tăng tính linh hoạt



```
var car = new Car();

car.StartCarWithEngine(new DieselEngine());

car.StartCarWithEngine(new HybridEngine());
```

ƯU ĐIỂM VÀ NHƯỢC ĐIỂM

A. ƯU ĐIỂM

3, Tăng khả năng tái sử dụng, kiểm thử và bảo trì

```
public class FakeEngine : IEngine
{
    public void Start() => Console.WriteLine("[TEST] Fake engine started.");
}
var testCar = new Car(new FakeEngine());
testCar.StartCar(); // Output: [TEST] Fake engine started.
```

ƯU ĐIỂM VÀ NHƯỢC ĐIỂM

B. NHƯỢC ĐIỂM

1, Yêu cầu cấu hình chi tiết

```
// Phải cấu hình từng lớp một
var engine = new GasEngine();
var brake = new ABSBrake();
var car = new Car(engine, logger);
```

```
Nếu Car lại phụ thuộc vào ILogger, IEngine, IBrakeSystem, IGPS,
v.v. → cấu hình tay rất phức tạp.
```

ƯU ĐIỂM VÀ NHƯỢC ĐIỂM

B. NHƯỢC ĐIỂM

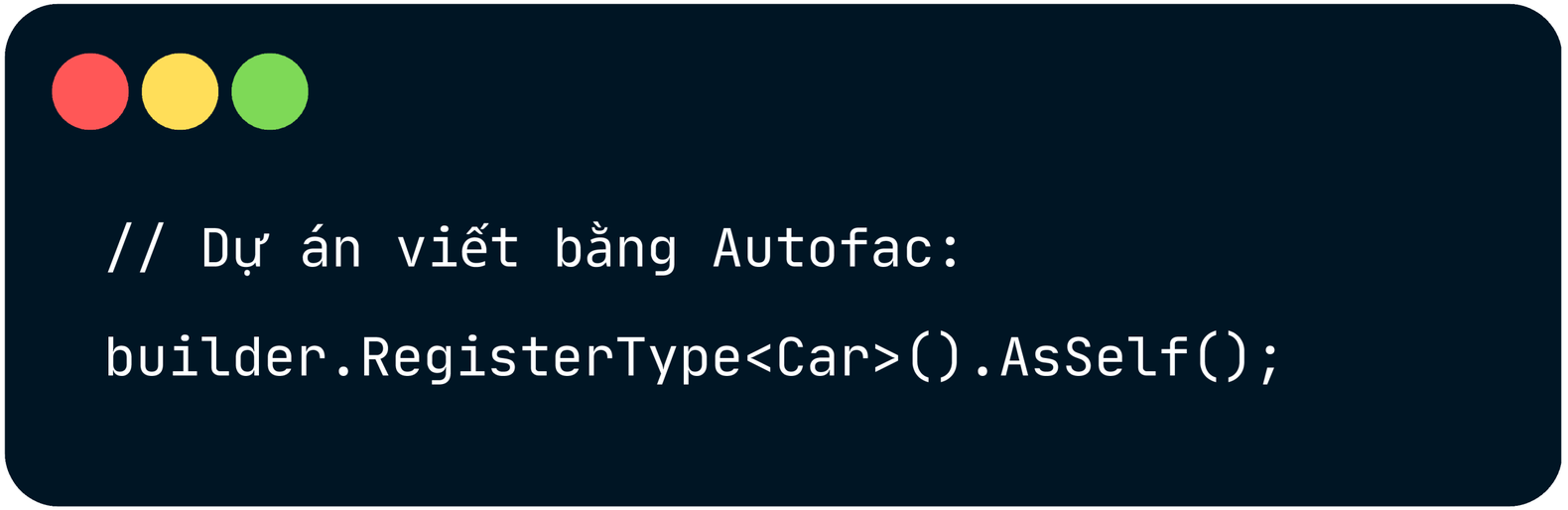
2, Khó theo dõi luồng thực thi sử dụng, kiểm thử và bảo trì

```
// Code bạn chỉ gọi:
var car = serviceProvider.GetService<Car>();
car.StartCar();
```

ƯU ĐIỂM VÀ NHƯỢC ĐIỂM

B. NHƯỢC ĐIỂM

3, Cần đầu tư công sức ban đầu và tăng sự phụ thuộc vào framework DI:

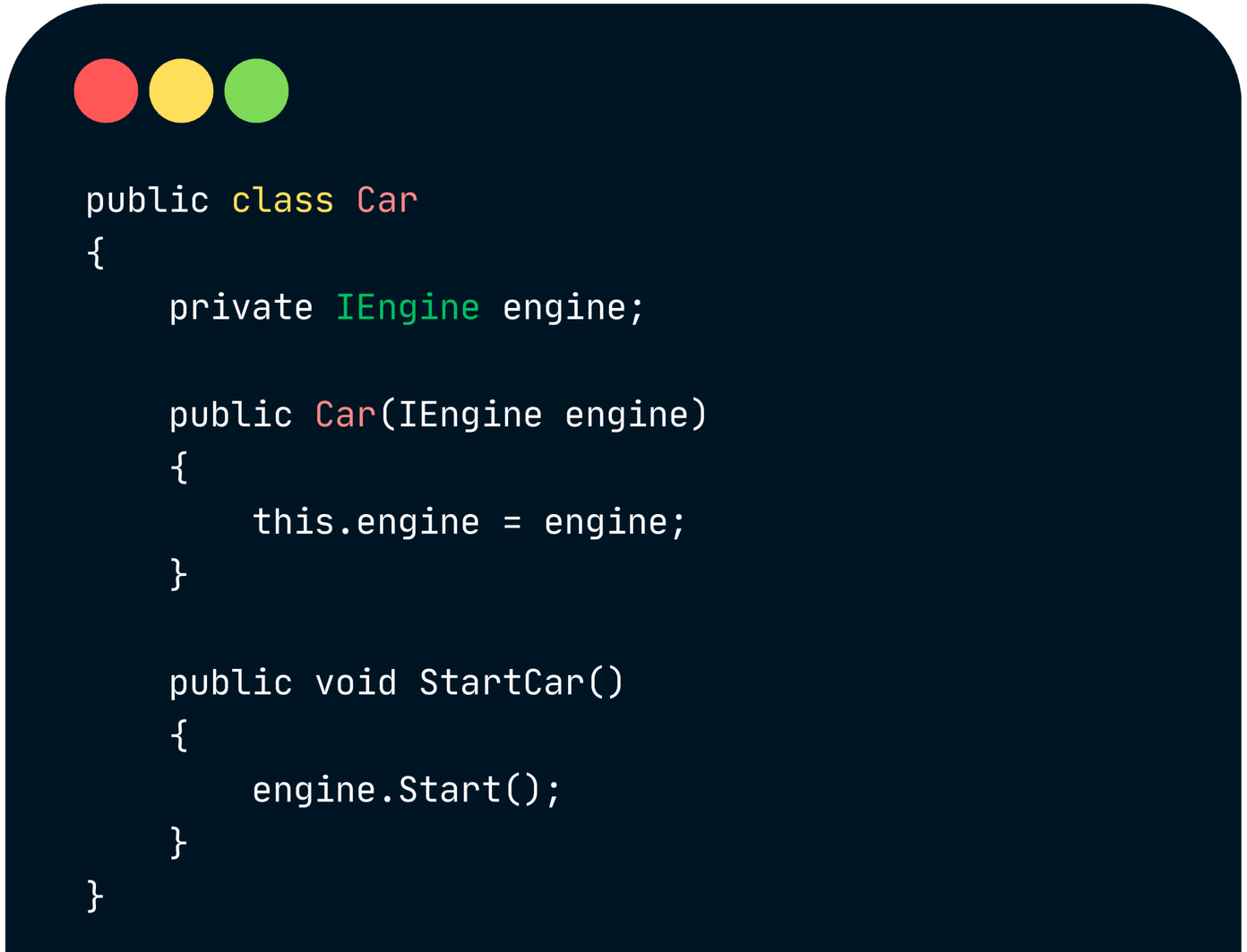


```
// Dự án viết bằng Autofac:

builder.RegisterType<Car>().AsSelf();
```

BEST PRACTICES KHI SỬ DỤNG DI

BEST PRACTICES

✅ Ưu tiên Constructor Injection



```
public class Car
{
    private IEngine engine;

    public Car(IEngine engine)
    {
        this.engine = engine;
    }

    public void StartCar()
    {
        engine.Start();
    }
}
```

BEST PRACTICES

✅ Dùng Interface Cho Dependency

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

```
public class Car {
    private readonly IEngine _engine;
    ...
```

```
public class DieselEngine : IEngine {
    public void Start() => Console.WriteLine("Diesel engine started.");
}
public class ElectricEngine : IEngine {
    public void Start() => Console.WriteLine("Electric engine started.");
}
```

BEST PRACTICES

✅ Chỉ Inject khi thật sự cần thiết

```
public CarManager(IEngine engine, ITransmission transmission, IBrakeSystem brakeSystem,
                  IInfotainment infotainment, IGPS gps, IClimateControl climateControl)
{
    _engine = engine;
    _transmission = transmission;
    _brakeSystem = brakeSystem;
    _infotainment = infotainment;
    _gps = gps;
    _climateControl = climateControl;
}
```

BEST PRACTICES

✅ Quản lý vòng đời dịch vụ hợp lý

Transient	Tạo một instance mới mỗi lần resolve
Scoped	Tạo một instance duy nhất trong phạm vi (scope) nhất định (ví dụ: mỗi request trong ASP.NET Core)
Singleton	Tạo một instance duy nhất cho toàn bộ vòng đời của ứng dụng

XIN CÁM ƠN THẦY VÀ CÁC BẠN ĐÃ CHÚ Ý LẮNG NGHE