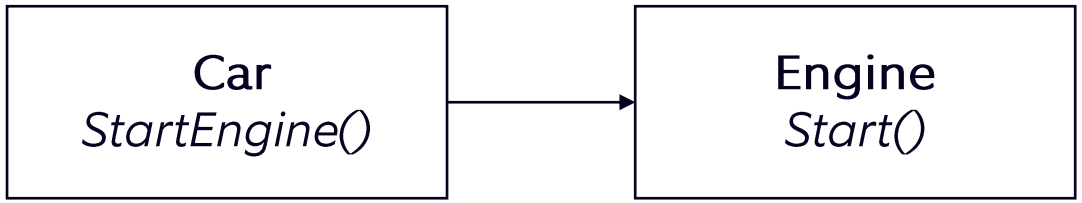
nagarro

Dependency Injection

Dependency Injection – Basic Example

Without dependency injection




```
public class Engine
{
    public void Start()
    {
        // Start the engine
    }
}
```

```
public class Car
{
    public void StartEngine()
    {
        Engine engine = new Engine();
        engine.Start();
    }
}
```

How to unit test **StartEngine()** method?

How to control the Engine object from outside?

Dependency Injection – Basic Example

Allow constructor injection

```
public class Car
{
    private readonly Engine engine;

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

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

Test method can now inject an Engine mock

```
public class EngineDummy : Engine
{
    public override void Start()
    {

    }
}
```

```
[Fact]
public void ShouldCallStartEngine()
{
    Engine mock = new EngineDummy();
    Car car = new Car(mock);
    car.StartEngine();
    // Verify that Start was called
}
```

Dependency Injection – Basic Example

A better solution using interfaces

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

public class DieselEngine : IEngine
{
    public void Start()
    {
        // Start the engine
    }
}

public class GasolineEngine : IEngine
{
    public void Start()
    {
        // Start the engine
    }
}
```

```
public class Car
{
    private readonly IEngine engine;

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

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

```
static void Main(string[] args)
{
    IEngine dieselEngine = new DieselEngine();
    Car car = new Car(dieselEngine);
}
```

Types of Dependency Injection

Constructor injection

```
// Constructor injection
public Car(Engine engine)
{
    this.engine = engine;
}
```

```
static void Main(string[] args)
{
    Engine engine = new DieselEngine();
    Car car = new Car(engine);
}
```

Setter (property) injection

```
// Setter (method) injection
public void SetEngine(Engine engine)
{
    this.engine = engine;
}
```

```
static void Main(string[] args)
{
    Engine engine = new DieselEngine();
    Car car = new Car();
    car.SetEngine(engine);
}
```

Types of Dependency Injection

- **Constructor injection** is preferred as it allows to implement *immutable* **objects** and to ensure that required dependencies are not null. It also allows to easily identify when a class has to many dependencies.
- **Setter injection** on the other hand can be handy for optional dependencies or when re-configuration (*re-injection*) is a desired feature, but null-checks must be performed whenever the dependency is used in code. With setter injection, required dependencies cannot be clearly communicated to the caller.

Inversion of Control Containers

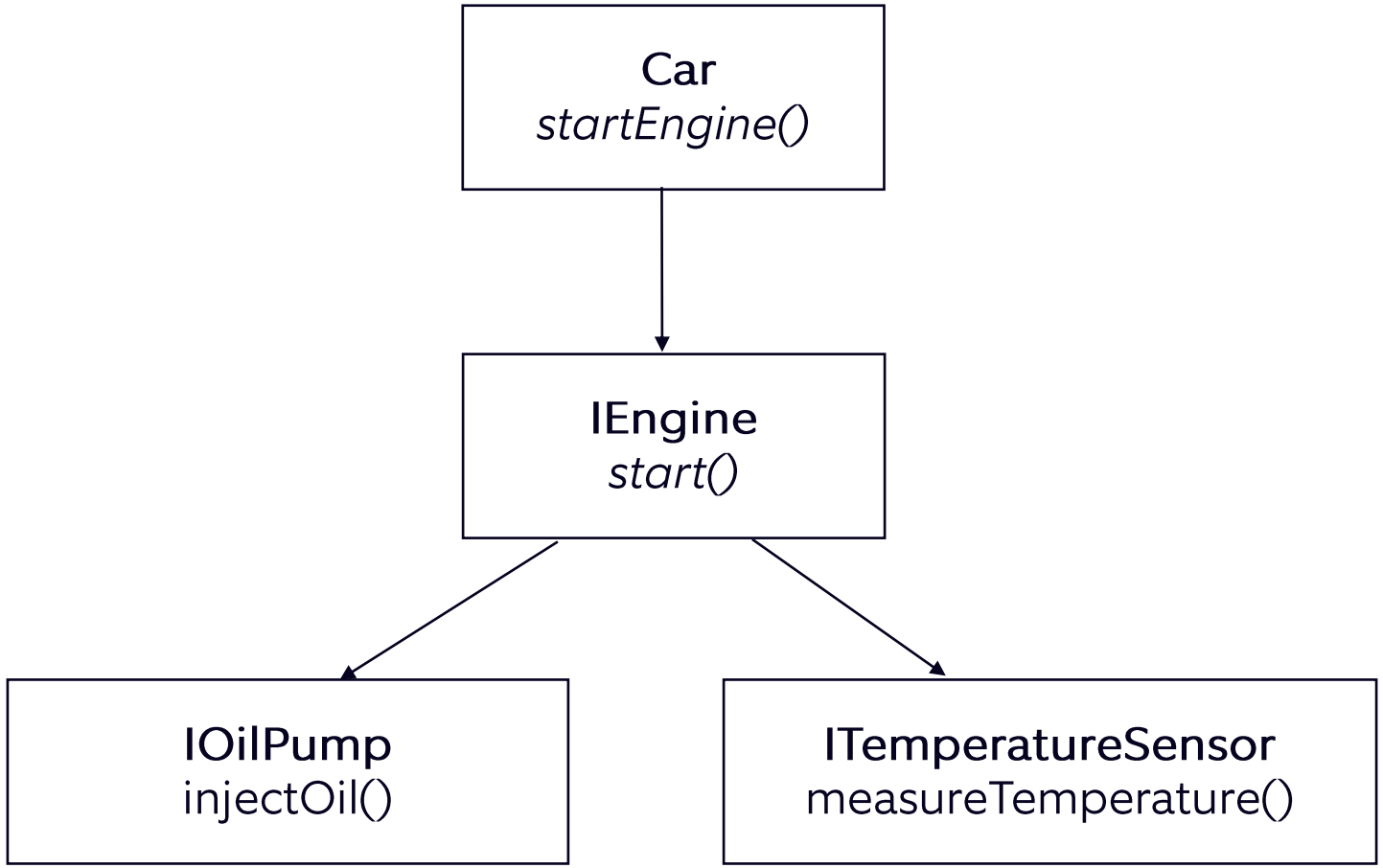
- An IoC container (or: Dependency Injection Container) is a framework that can **create dependencies and injects them automatically when required**
- It also takes care of the **lifetime and destruction** of the dependencies it creates
- While in simple projects a “manual” approach to DI seems nice enough, in more complex projects this could become a nightmare.
- Thus, an **IoC container helps us manage our dependencies in a simple and easy way**

```
public class Car
{
    private Engine engine;

    public Car(Engine engine)
    {
        this.engine = engine;
    }
    public void StartEngine()
    {
        engine.Start();
    }
}
```

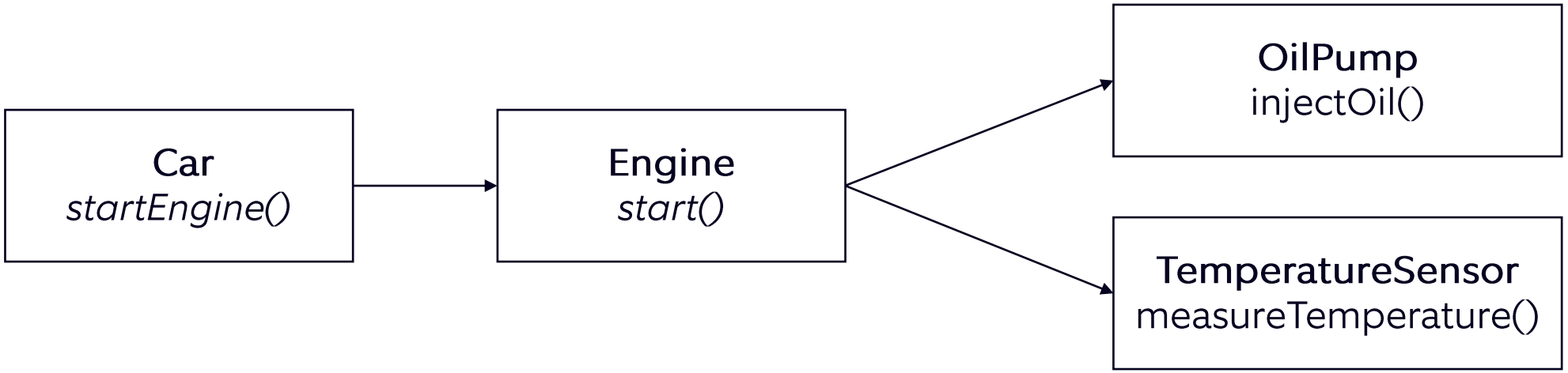
Example with .NET Core
Built-In Dependency Injection Framework

```
static void Main(string[] args)
{
    IServiceCollection services = new ServiceCollection();
    services.AddTransient<Car, Car>();
    services.AddTransient<Engine, GasolineEngine>();

    var serviceProvider = services.BuildServiceProvider();
    Car car = serviceProvider.GetService<Car>();
    car.StartEngine();
}
```

Dependency Injection – Chain of Dependencies

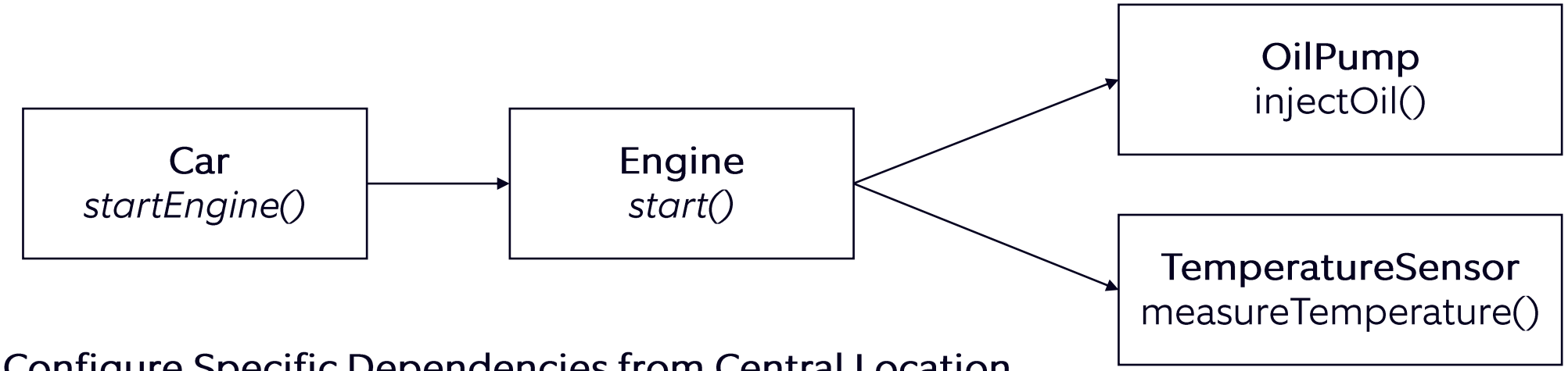


Dependency Injection – Chain of Dependencies




```
public Car GetCar()
{
    TemperatureSensor tempSensor = new CelsiusTemperatureSensor();
    OilPump oilPump = new MercedesOilPump();
    Engine engine = new DieselEngine(oilPump, tempSensor);
    Car car = new Car(engine);

    return car;
}
```

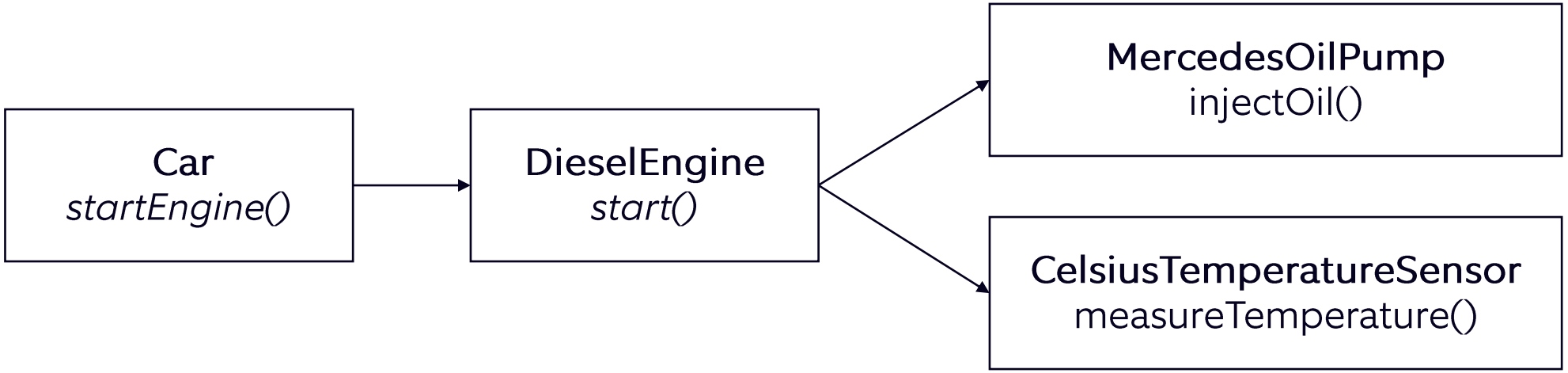
Dependency Injection – Chain of Dependencies



Configure Specific Dependencies from Central Location

```
public static void ConfigureServices()
{
    IServiceCollection services = new ServiceCollection();
    services.AddTransient<TemperatureSensor, CelsiusTemperatureSensor>();
    services.AddTransient<OilPump, MercedesOilPump>();
    services.AddTransient<Engine, DieselEngine>();
    services.AddTransient<Car, Car>();

    ServiceProvider = services.BuildServiceProvider();
}
```

Dependency Injection – Chain of Dependencies



Resolve Specific Dependencies from Arbitrary Location and Separate Configuration from Access

```
public class MercedesCarServiceStation
{
    private Car car;

    public void CheckInNewCar()
    {
        var serviceProvider = ServiceRegistry.ServiceProvider;
        car = serviceProvider.GetService<Car>();
    }
}
```