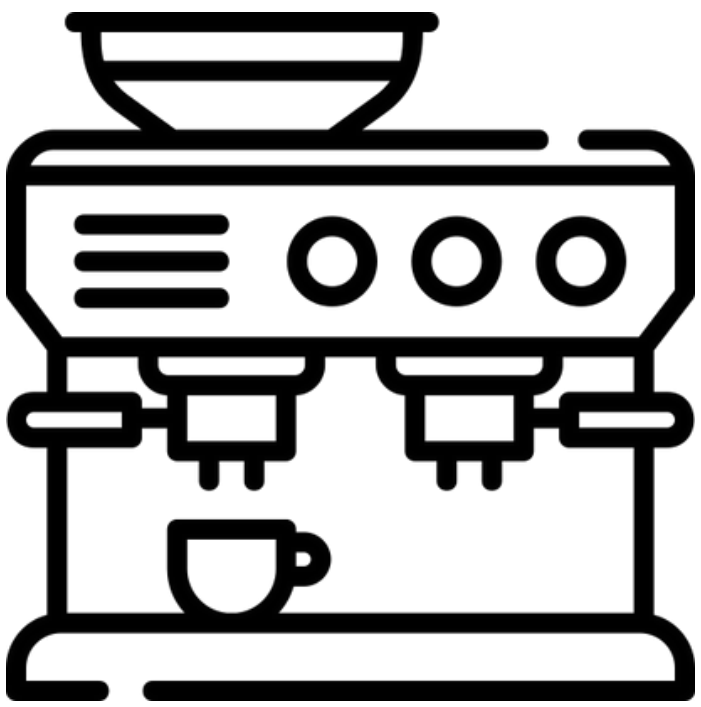
SOLID Principles

Reminder for clean code



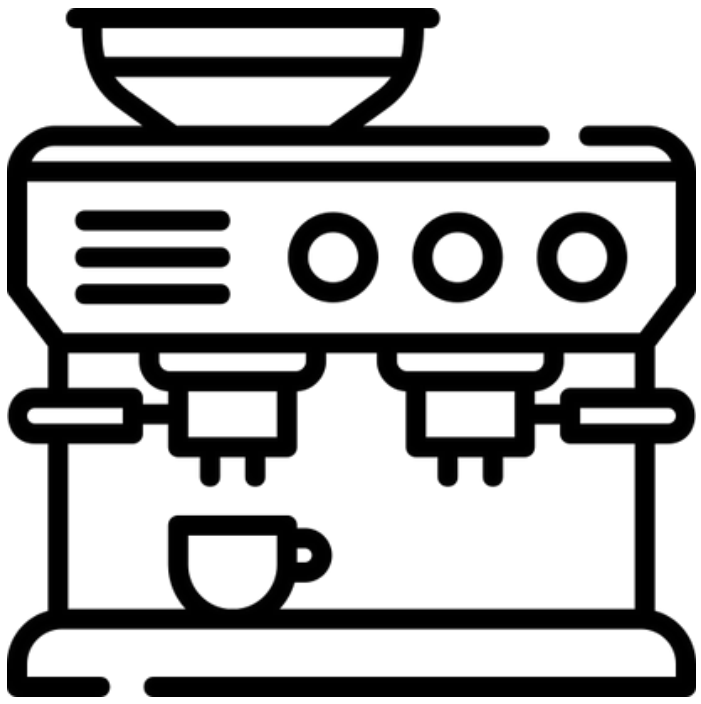
with Coffee machine

Analogies

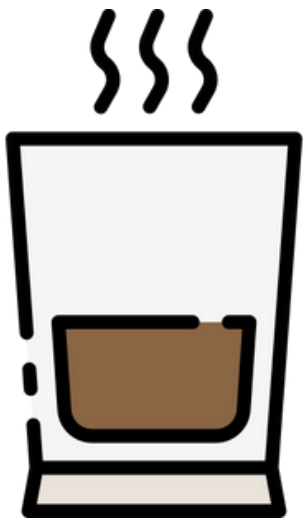
@Romain-OD

Single Responsibility

Analogy



Imagine our coffee machine could serve Coffee, Macchiato, Ristretto...



The purpose is to not alter the class in charge of making Coffee if we want to change the way to make a Ristretto

@Romain-OD

Code example

```
public class CoffeeMaker
{
    public string MakeCoffee () { return "Making Coffee..."; }
    public string MakeRistreto() { return "Making Ristreto..."; }
    public string MakeMacchiato() { return "Making Macchiato..."; }
}
```

```
public class RistretoMaker : IBeverageMaker
{
    public string Make() { return "Making Ristreto..."; }
}
```

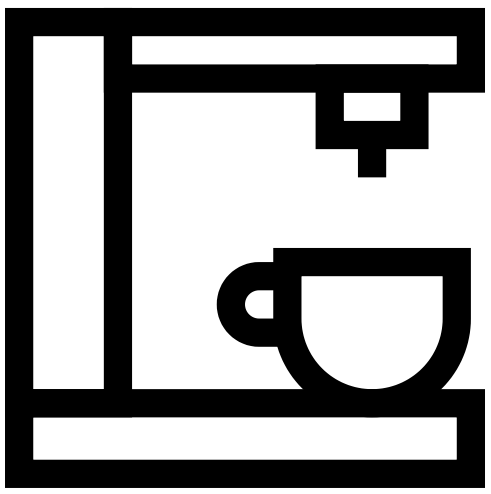
```
public class CoffeeMaker: IBeverageMaker
{
    public string Make() { return "Making Coffee..."; }
}
```

```
public class MacchiatoMaker : IBeverageMaker
{
    public string Make() { return "Making Macchiato..."; }
}
```

@Romain-OD

Open-Closed Principle

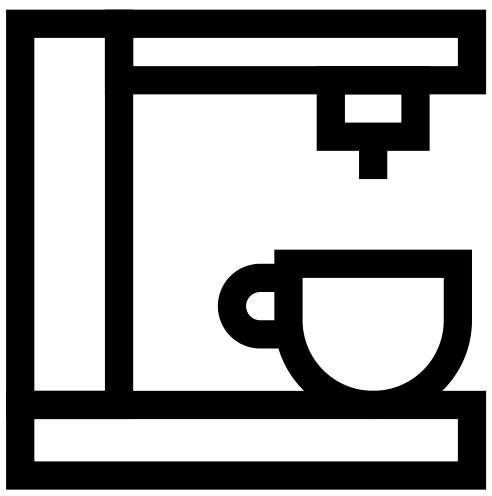
Analogy



Machine Brew

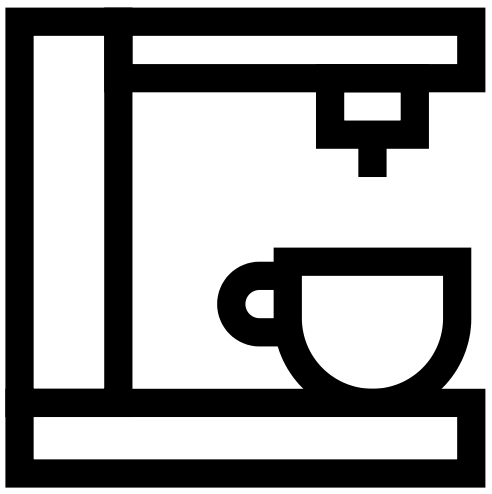
- coffee

updating



Machine Brew

- latte



Machine Brew

- coffee
- latte

Someday you have to improve your machine to also do Capucino you don’t want to affect your brew method

@Romain-OD

Violating OCP

```
public class CoffeeMachine
{
    public void BrewCoffee(string coffeeType)
    {
        if (coffeeType == "Espresso")
        {
            Console.WriteLine("Brewing Espresso...");
        }
        else
        {
            Console.WriteLine("Unknown coffee type.");
        }
    }
}
```

This class is closed for extension because if we want to add support for another type of coffee, like Latte, we would need to modify the BrewCoffee method to handle the new case.

@Romain-OD

using polymorphism

```
public interface IBrewable
{
    string Brew();
}
```

```
public class Espresso : IBrewable
{
    public string Brew()
    {
        return "Brewing Es
    }
}
```

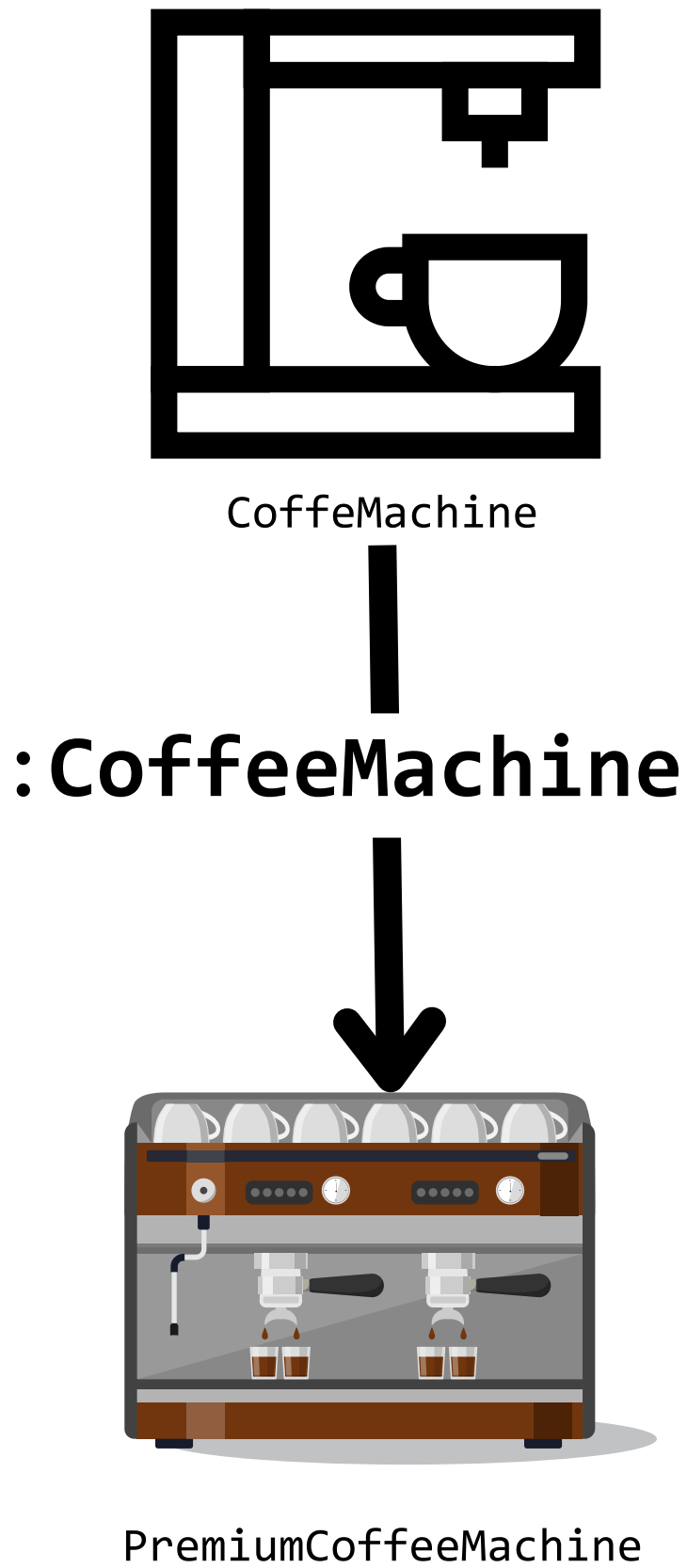
```
public class Latte : IBrewable
{
    public string Brew()
    {
        return "Brewing Latte...";
    }
}
```

With this approach, adding support for a new type of coffee (e.g., Cappuccino) becomes as simple as creating a new class that implements IBrewable and adding it to the CoffeeMachine.

@Romain-OD

Liskov Substitution Principle

Analogy



Clients should be able to use both seamlessly

@Romain-OD

Violating LSP

```
public class CoffeeMachine
{
    public virtual void Brew()
    {
        Console.WriteLine("Brewing coffee...");
    }
}

public class PremiumCoffeeMachine : CoffeeMachine
{
    public override void Brew()
    {
        Console.WriteLine("Grinding beans...");
        // Additional steps for grinding beans
        Console.WriteLine("Brewing premium coffee...");
    }
}
```

The PremiumCoffeeMachine's Brew method includes an additional step (grinding beans) that the CoffeeMachine does not perform. This violates LSP because substituting a PremiumCoffeeMachine for a CoffeeMachine changes the behavior of the program in an unintended manner.

@Romain-OD

Refactoring with Composition

While inheritance is a powerful tool for code reuse and expressing "is-a" relationships, it can sometimes lead to tight coupling between classes, making the code harder to maintain and extend.

Composition, on the other hand, promotes loose coupling and flexibility, allowing for easier modification and extension of the codebase

To adhere to LSP, we can refactor the design to use composition.

This involves creating separate components for brewing and grinding, which can be combined as needed.

@Romain-OD

1 - Defining Components

```
public interface IBrewer
{
    void Brew();
}
```

```
public interface IBeanGrinder
{
    void GrindBeans();
}
```

2 - Implement components

```
public class StandardBrewer : IBrewer
{
    public void Brew()
    {
        Console.WriteLine("Brewing coffee...");
    }
}
```

```
public class BeanGrinder : IBeanGrinder
{
    public void GrindBeans()
    {
        Console.WriteLine("Grinding beans...");
    }
}
```

@Romain-OD

3 - Composing

```
public class CoffeeMachine
{
    protected IBrewer _brewer;
    protected IBeanGrinder _beanGrinder;

    public CoffeeMachine(IBrewer brewer, IBeanGrinder beanGrinder)
    {
        _brewer = brewer;
        _beanGrinder = beanGrinder;
    }

    public void Brew()
    {
        _brewer.Brew();
    }

    public void GrindAndBrew()
    {
        _beanGrinder.GrindBeans();
        Brew();
    }
}
```

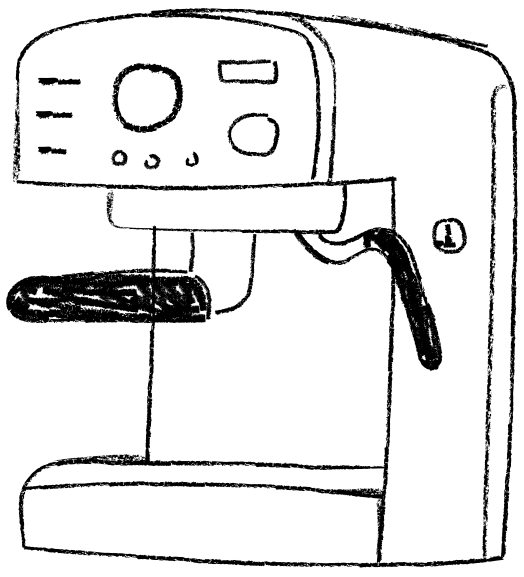
```
public class PremiumCoffeeMachine : CoffeeMachine
{
    public PremiumCoffeeMachine() : base(new StandardBrewer(), new BeanGrinder())
    {
    }
}
```

@Romain-OD

Interface Segregation Principle

Analogy

```
interface ICoffee {
 GrindCoffeeBeans();
 BoilWater();
 PourCoffee();
 AddCream();
}
```



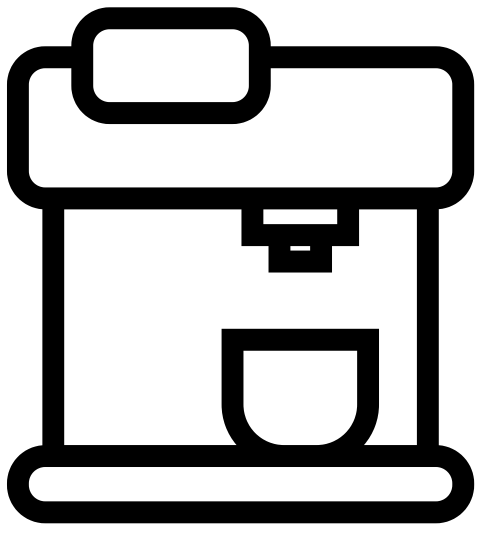
cream adder

:IGrinder

:IKettle

:IPourer

:ICreamAdder



no cream adder

:IGrinder

:IKettle

:IPourer

```
interface IGrinder {..}
```

```
interface IKettle {..}
```

```
interface IPourer {..}
```

```
interface ICreamAdder {...}
```

With small interface, we can compose any sort of coffee machine

@Romain-OD

Violating ISP

```
public interface ICoffeeMachine
{
    string AddCream();
    string BoilWater();
    string GrindCoffeeBeans();
    string PoureCoffee();
}
```

This example forces any implementing class to define methods for adding cream, boiling water, grinding coffee beans, and pouring coffee, even if the class does not require all these functionalities, leading to unnecessary implementation details and increased complexity

@Romain-OD

Split in small interface

```
public interface ICreamAdder
{
    string AddCream()
}
```

```
public interface IGrinder
{
    string GrindCoffeeBeans();
}
```

```
public interface IKettle
{
    string BoilWater();
}
```

```
public interface IPourer
{
    string PoureCoffee();
}
```

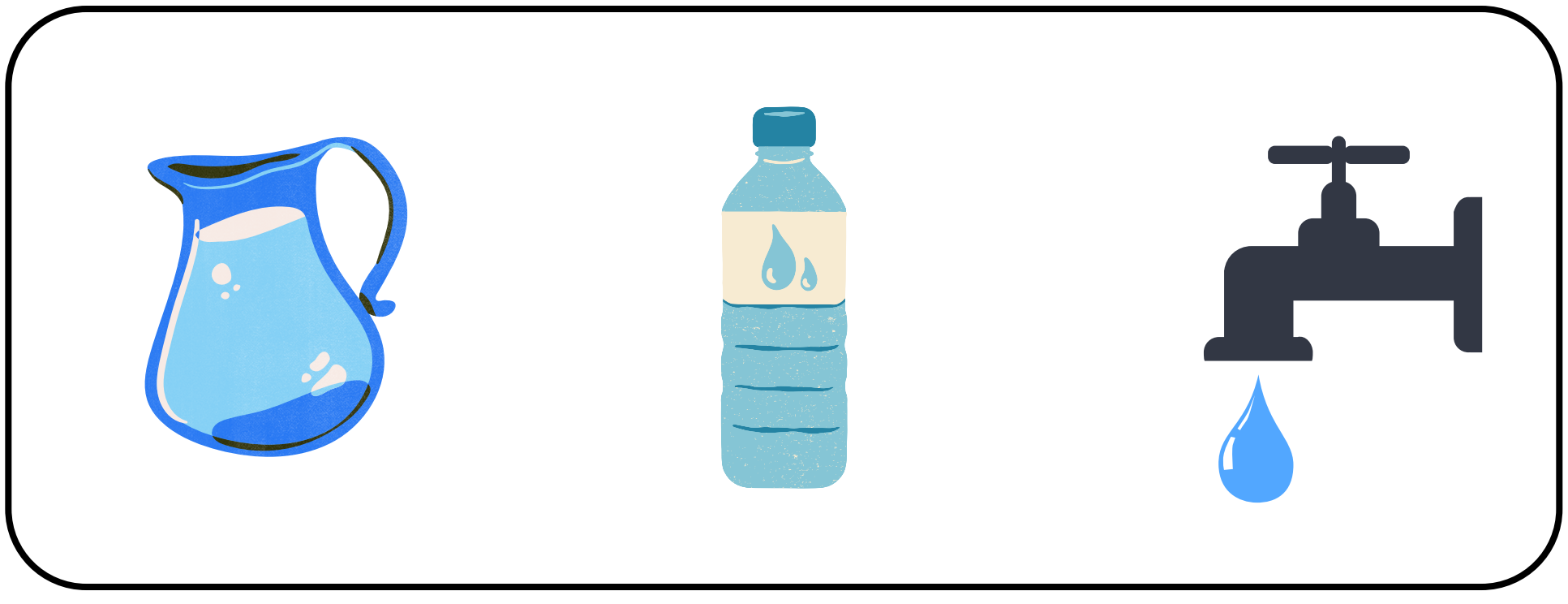
This way we can even implement a simple ketlle and serve hot water

@Romain-OD

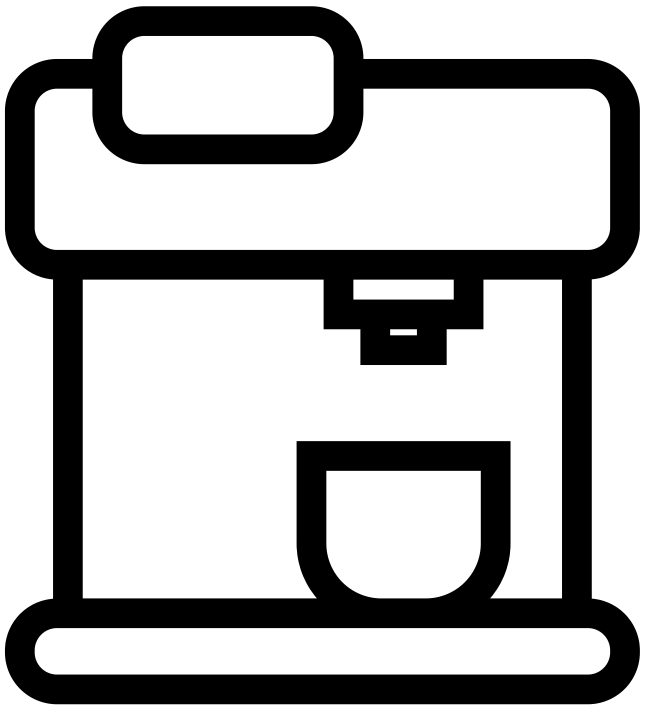
Understand DI & IoC

Analogy

`:IWaterProvider`

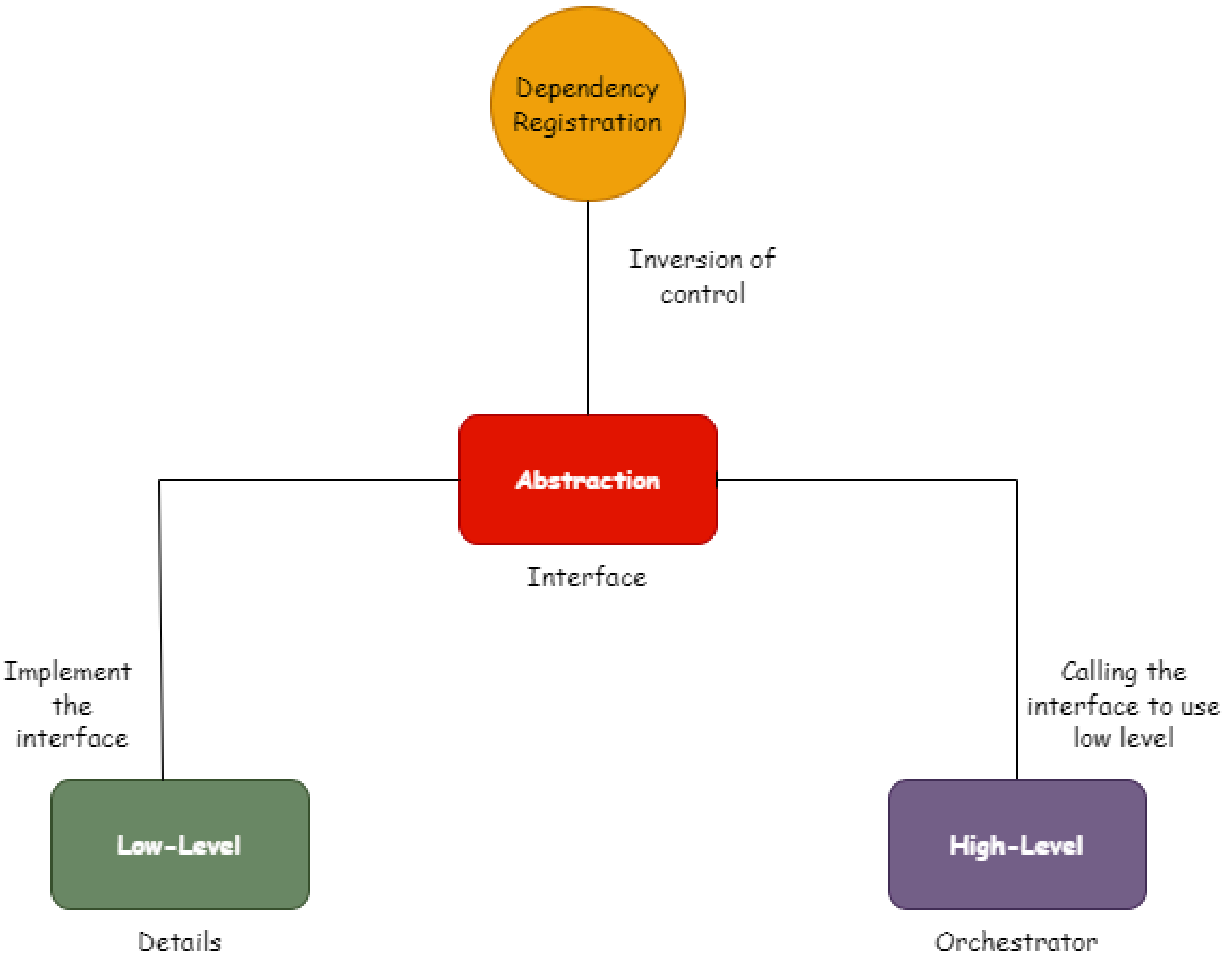


A coffee machine needs water to boil, but it doesn't matter who supplies the water



@Romain-OD

How it’s works



@Romain-OD

Abstraction

```
public interface IWaterProvider
{
    public Water GetWater();
}
```

@Romain-OD

Low-Level

```
public class Carafe : IWaterProvider
{
    public Water GetWater()
    {
        //Implement logic of a Carafe
        //let's rocks you imaganation
        return new Water();
    }
}
```

@Romain-OD

High-Level

```
public class CoffeeMachine(IWaterProvider waterProvider)
{
    public Tank WaterTank { get; set; }
    public void FillTank()
    {
        WaterTank = new Tank
        {
            Content = waterProvider.GetWater()
        };
    }
}
```

@Romain-OD

Dependence Registration

```
services.AddSingleton<IWaterProvider, Carafe>();
```

OR

```
services.AddScoped<IWaterProvider, Carafe>();
```

OR

```
services.AddTransient<IWaterProvider, Carafe>();
```

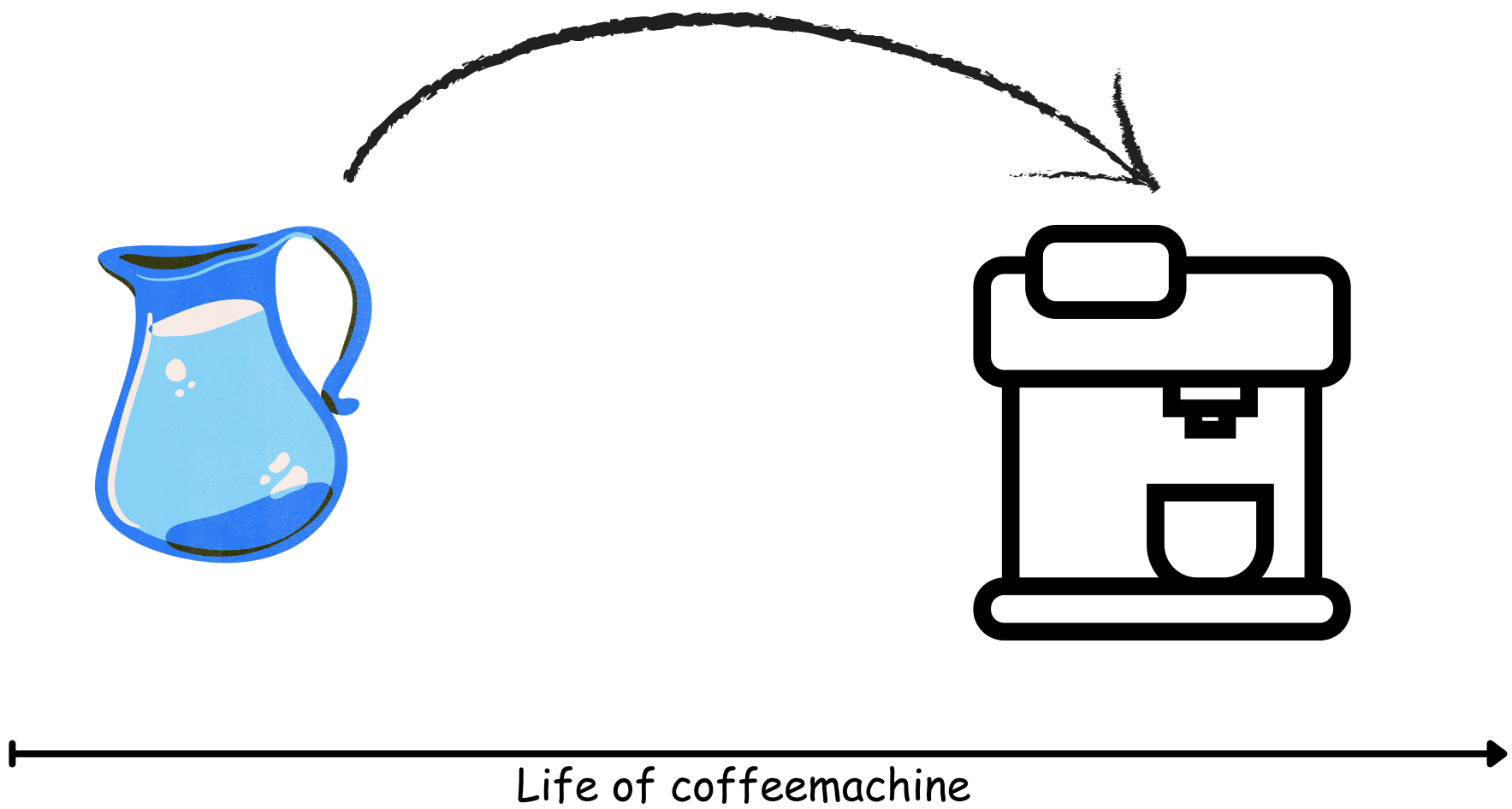
Which one choose ?

@Romain-OD

Singleton

A single instance of the dependency is created and shared across the entire application.

The singleton approach is lifetime, it means every day you use the coffee machine you use a carafe to fill the water tank.

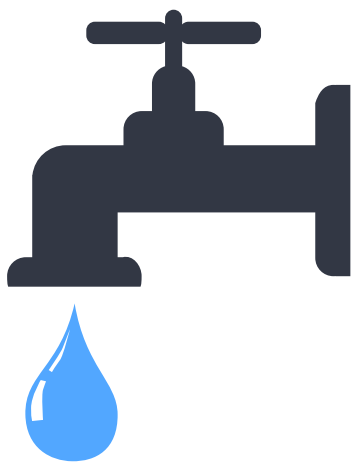


@Romain-OD

Scoped

A new instance of the dependency is created once per scope. A scope can be thought of as a unit of work.

The scoped approach, it's like today I committed to use the carafe, tomorrow I will use a bottle.

Monday	Tuesday	Wednesday
		

@Romain-OD

Transient

A new instance of the dependency is created each time it is requested

With transient approach, we take the first provider we found to fill the water tank each time we use the coffee machine.

8:00	8:45	10:00	12:35

@Romain-OD

What about reference ?

Can we use Scoped service in Singleton?
Can we use Singleton service in Transient?
Can we use Transient service in Scoped?
Can we use Singleton service in Scoped?
Can we use Scoped service in Transient?
...

@Romain-OD

Captive Dependency

Captive Dependency is a dependency with an incorrectly configured lifetime. A service should never depend on a service that has a shorter lifetime than its own.

	Transient	Scoped	Singleton
Transient	✓	✓	✓
Scoped		✓	✓
Singleton	✗	✗	✓

@Romain-OD

Can we use a different provider to fill the Water Tank and rinse the filter ?

.NET 8 introduce keyed services

Keyed services managing Dependency Injection (DI) services by associating them with keys for registration and retrieval.

We use methods **AddKeyedSingleton**, **AddKeyedScoped**, or **AddKeyedTransient** to register a service with a specific key.

We use the **[FromKeyedServices]** attribute and specify the corresponding key to access a registered service

@Romain-OD

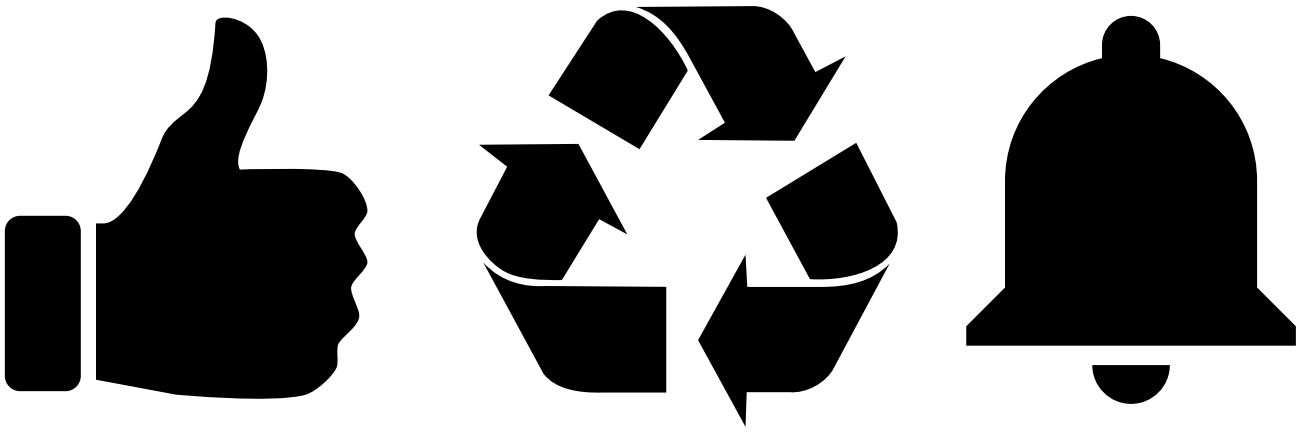
Keyed services

```
services.AddKeyedSingleton<IWaterProvider, Faucet>("Faucet");
services.AddKeyedSingleton<IWaterProvider, Carafe>("Carafe");
```

```
public class Filter([FromKeyedServices("Faucet")] IWaterProvider waterProvider)
{
    public void Rinse()
    {
        Water liquid = waterProvider.GetWater();
        //Use liquid to Rinse the filter
    }
}
```

```
public class WaterTank([FromKeyedServices("Carafe")]IWaterProvider waterProvider)
{
    public Water Content  { get; set; }
    public void FillTank()
    {
        Content = waterProvider.GetWater();
    }
}
```

@Romain-OD

@Romain-OD

Thanks for reading