Solid Principles and Decorator Pattern

Walter Moscatelli
walter.moscatelli@eoc.ch

Recipe

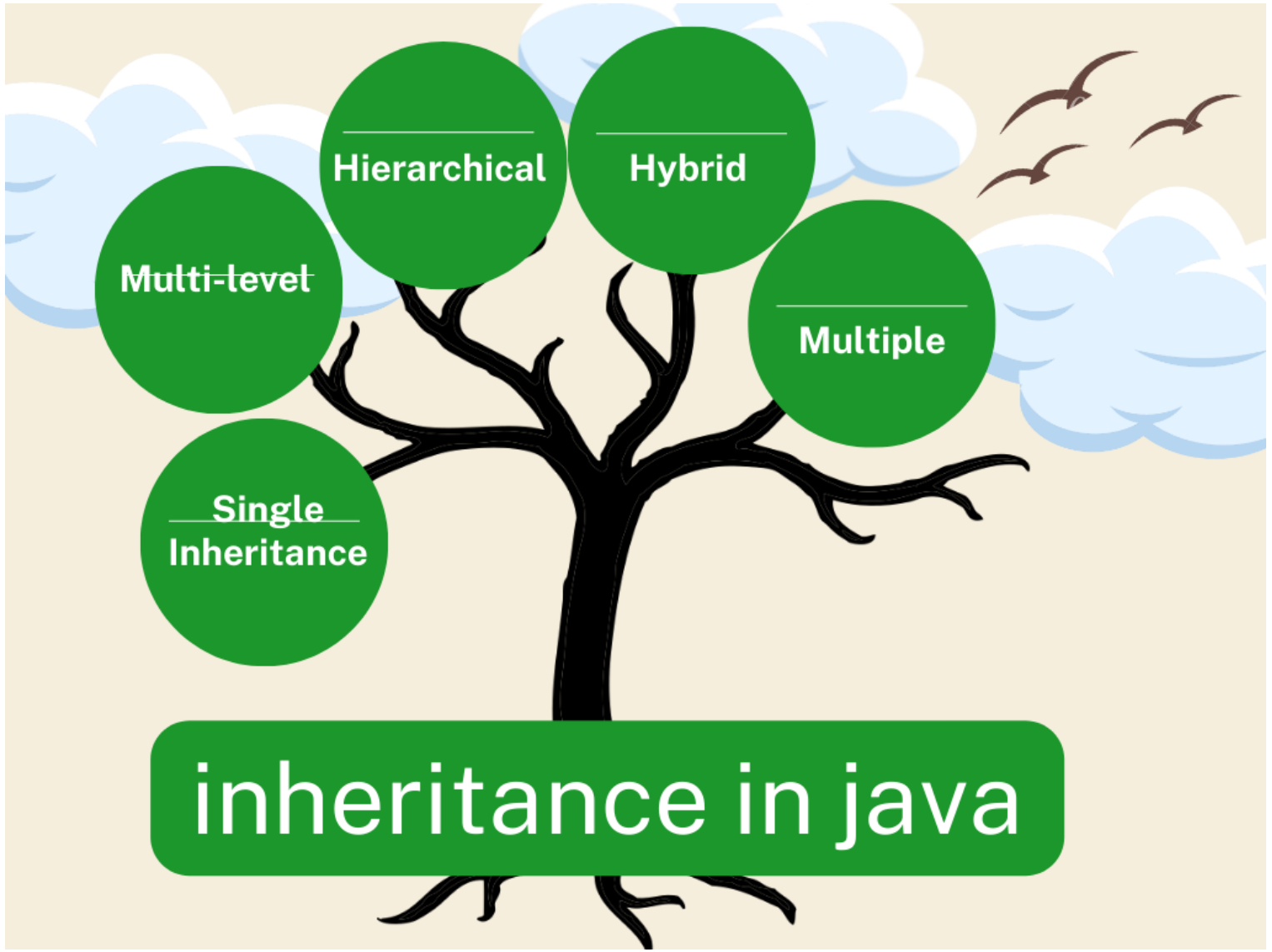
Ingredients :

- Inheritance and Exploding class hierarchy
- Design decorator pattern
- Practical application of this pattern
- Solid principles
- Conclusion

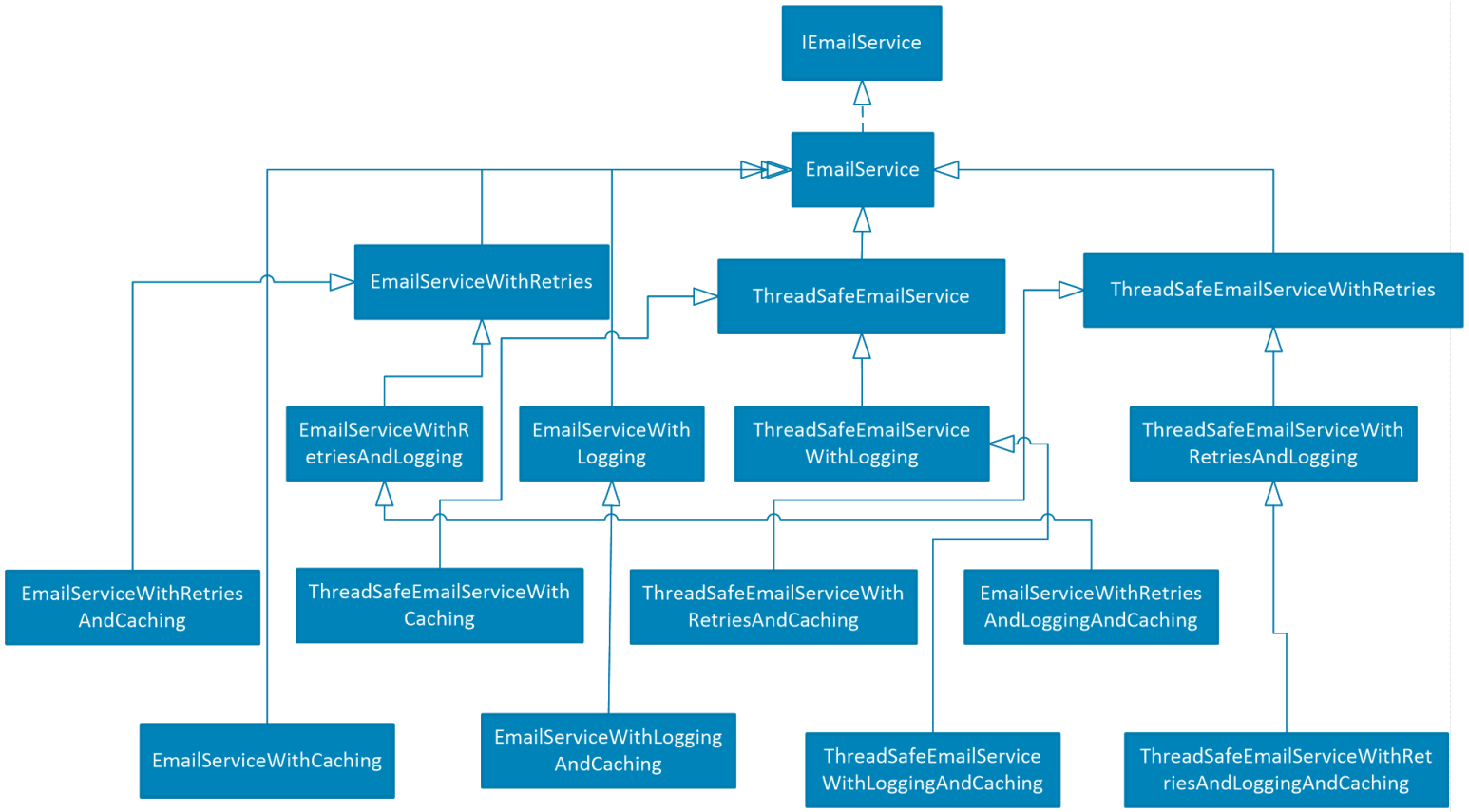
Inheritance

When object-oriented programming was introduced, inheritance was the primary pattern used to extend object functionality.

It has been shown that extending objects using inheritance often results in an exploding class hierarchy, known as **Exploding class hierarchy**.

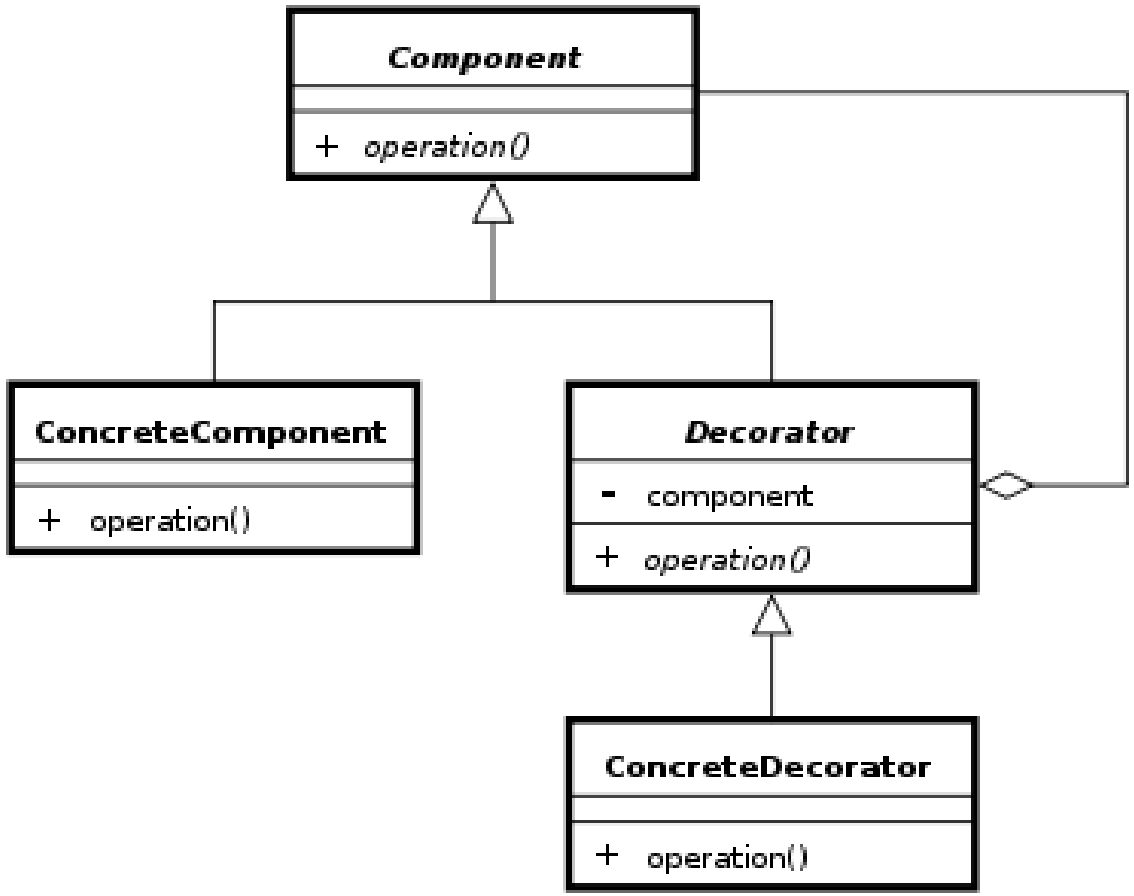


Exploding class hierarchy



Design pattern **Decorator**

- The decorator is a structural design pattern that uses **composition** instead of **inheritance**
- It provides a flexible alternative to sub-classing for extending functionality at runtime


Component
+ operation()
ConcreteComponent
+ operation()
Decorator
- component
+ operation()
ConcreteDecorator
+ operation()

- **Component —** This is basically an interface that describers behavior of concrete component as well as decorator. Depending on the existing project structure, this could be an interface or abstract class.
- **ConcreteComponent —** The actual object in which the new functionalities can be added dynamically. We can also wrap up decorators with other decorators.
- **Decorator —** Defines the interface for all the dynamic functionalities that can be added to the concrete component. The decorator **IS** a component and also **HAS** a component. This way components and decorators are interchangeable.
- **ConcreteDecorator —** Describes all the functionalities that can be added to the concrete components.

Kebabbaro

The main challenge is to create a system that allows for:

- Easy addition of new ingredients.
- Dynamic pricing based on kebab type and added condiments.
- Ensuring that changes to one part of the code do not affect other parts.
- Minimizing code duplication and adhering to the Open-Closed Principle.

Kebabbaro

- Kebab
- Kebab_Cipolla subclass of Kebab con Cipolla
- Kebab_Piccante subclass of Kebab con Piccante
- Kebab_CipollaPiccante subclass of Kebab con Cipolla e Piccante
- Falafel
- Falafel_Cipolla subclass of Falafel con Cipolla
- Falafel_Piccante subclass of Falafel con Piccante
- Falafel_CipollaPiccante subclass of Falafel con Cipolla e Piccante

Exploding class hierarchy

Actually we have 8 classes.

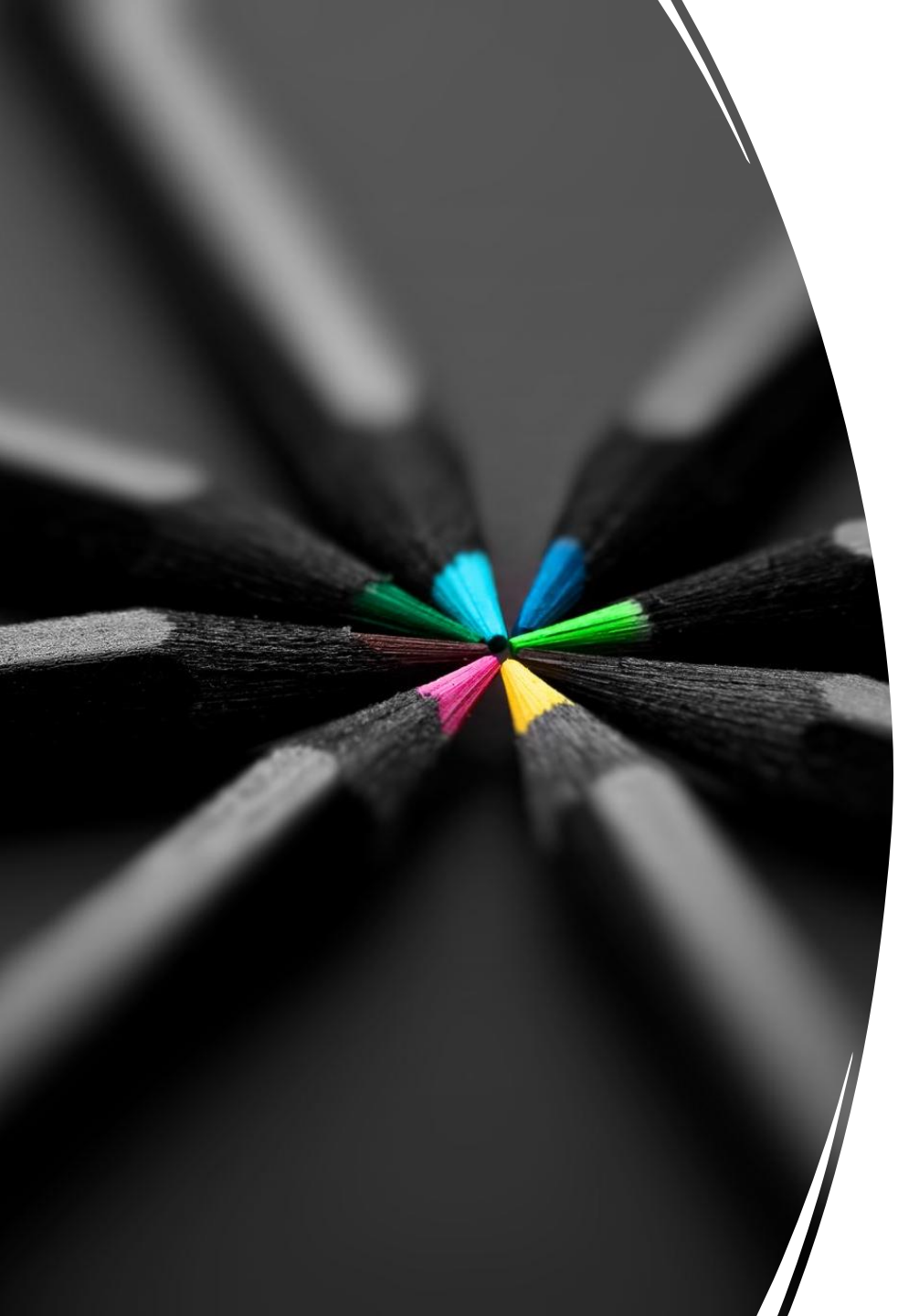
What happens if we add yogurt sauce or chips?

Let's try to solve this problem with the decorator pattern

Component : Pasto

We need a class to model the generic meal

```
public abstract class Pasto {
String nome = "";
public String getNome() {
return nome;
}
public abstract double getPrezzo();
}
```



Decorator SupplementiDecorator

We need a class for ingredient additions to our product which will be the basis for our Decorator

- public abstract class SupplementiDecorator extends Pasto {
- protected Pasto pasto;
- @Override
- public abstract String getNome();
- }

ConcreteComponent:

Kebab extends Pasto

```
public class Kebab extends Pasto {
 public Kebab() {
  nome = " Kebab ";
 }
@Override
 public double getPrezzo() {
  return 5.50;
 }
}
```

ConcreteComponent:

```
Falafel extends Pasto
public class Falafel extends Pasto {
 public Falafel() {
  nome = " Falafel ";
 }
@Override
 public double getPrezzo() {
  return 6.00;
 }
}
```

ConcreteDecorator Cipolla

```
public class ExtraCipollaDecorator extends SupplementiDecorator {
public ExtraCipollaDecorator(Pasto pasto){
this.pasto = pasto;
}
@Override
public String getNome()  {
return pasto.getNome()+  " con cipolla";
}
@Override
public double getPrice() {
return pasto.getPrice()+0.20;
}
}
```

ConcreteDecorator Piccante

- public class ExtraPiccanteDecorator extends SupplementiDecorator {
- public ExtraPiccanteDecorator(Pasto pasto){
- this.pasto = pasto;
- }
- @Override
- public String getNome() {
- return pasto.getNome()+ " con piccante";
- }
- @Override
- public double getPrice() {
- return pasto.getPrice()+0.40;
- }
- }

```
Pasto kebab = new Kebab();

System.out.println("Prodotto:" + kebab +" di prezzo " + String.format("%.2f", kebab.getPrice()));

Pasto falafel = new Falafel();

Pasto kebabconCipolla = new ExtraCipollaDecorator(kebab);

System.out.println("Prodotto:" + kebabconCipolla.getProductName() +" di prezzo " +
String.format("%.2f", kebabconCipolla.getPrice()));

Pasto kebabconCipollaePiccante = new ExtraPiccanteDecorator (new ExtraCipollaDecorator
(kebab));

System.out.println("Prodotto:" + kebabconCipollaePiccante.getProductName() +" di prezzo " +
String.format("%.2f", kebabconCipollaePiccante.getPrice()));}
```

Open-Closed Principle: The **Decorator** pattern adheres to the Open-Closed Principle, which means that we can extend the functionality of objects without modifying their source code. This promotes code stability and reduces the risk of introducing new bugs when adding new features.

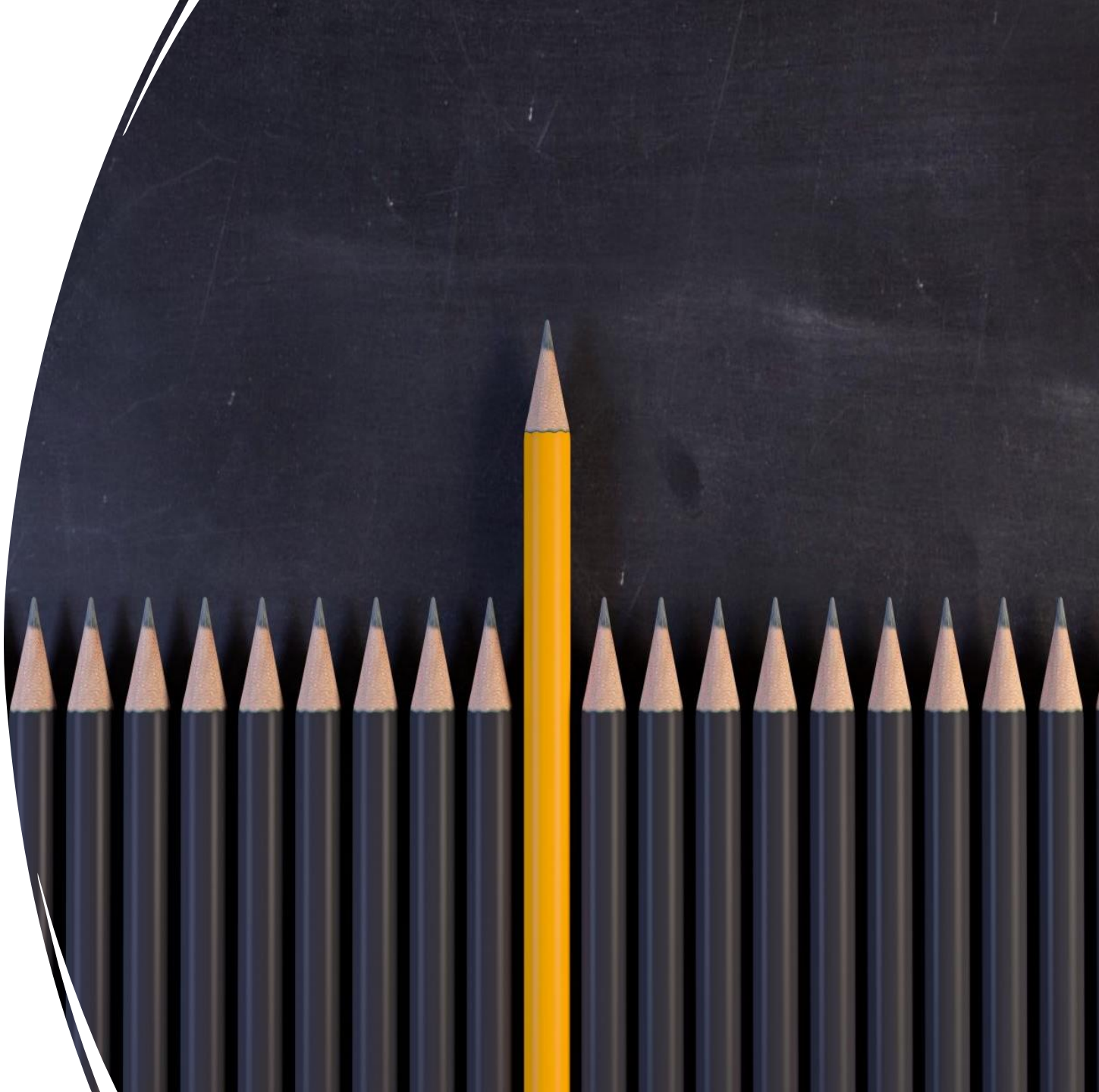
Single Responsibility Principle: Each decorator class has a single responsibility, which makes the code more maintainable and easier to understand. This aligns with the Single Responsibility Principle (SRP), one of the SOLID principles of object-oriented design.

Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP): Each decorator is responsible for a single aspect of behavior, ensuring that each class has a single reason to change.

Example: A meal class manage a product, while a separate Decorator add ingredients



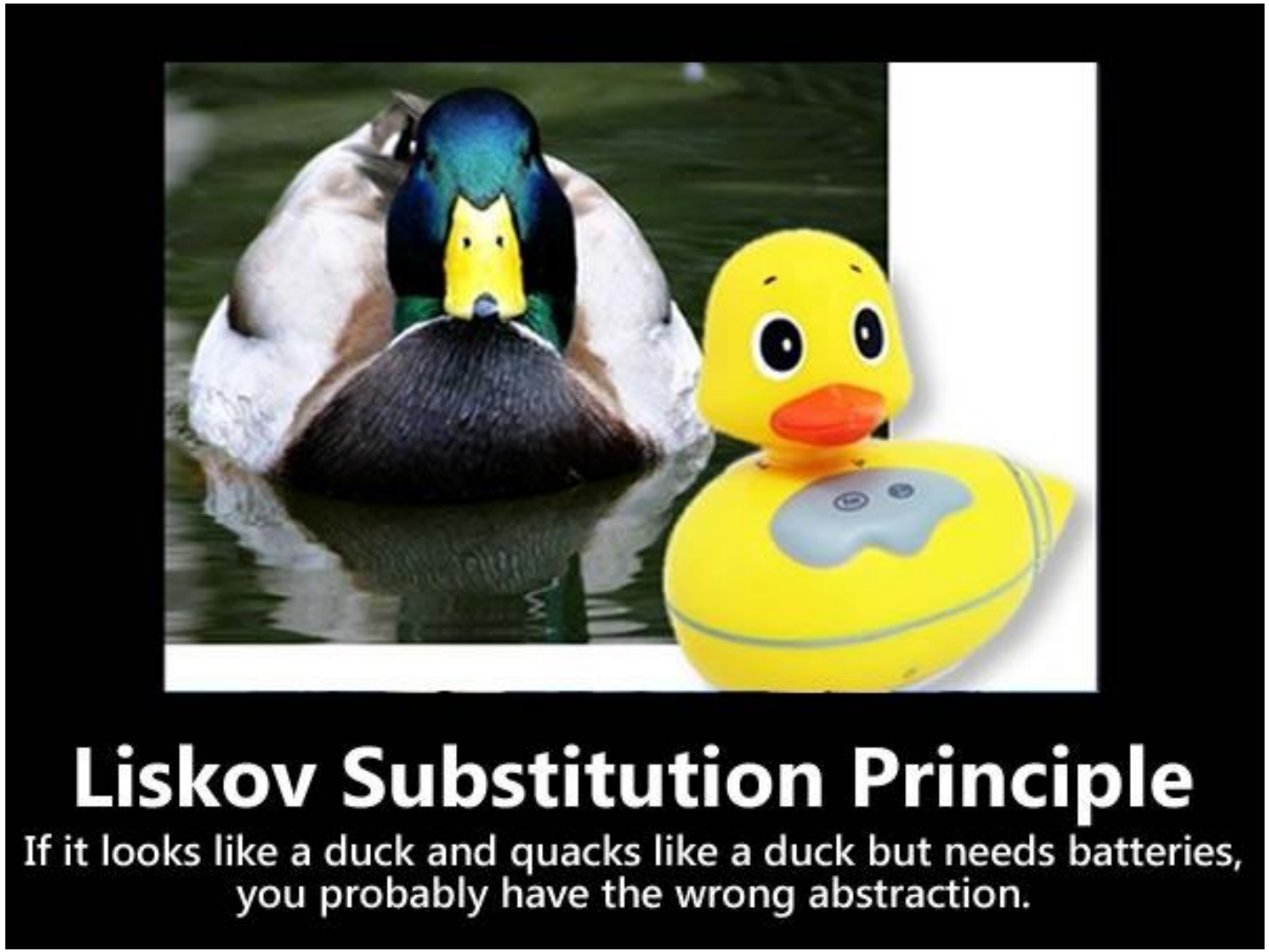
Open/Closed Principle (OCP)

Open/Closed Principle (OCP)

The component is open for extension through decorators but closed for modification, allowing you to add new behaviors without changing existing code.

Example: An ExtraCipollaDecorator can be added to product without modifying the original Kebab class.

Liskov Substitution Principle (LSP)



Liskov Substitution Principle (LSP)

Decorators and the original component are interchangeable because they adhere to the same interface.

Example: A Kebab can be substituted from Pasto without affecting the client code.

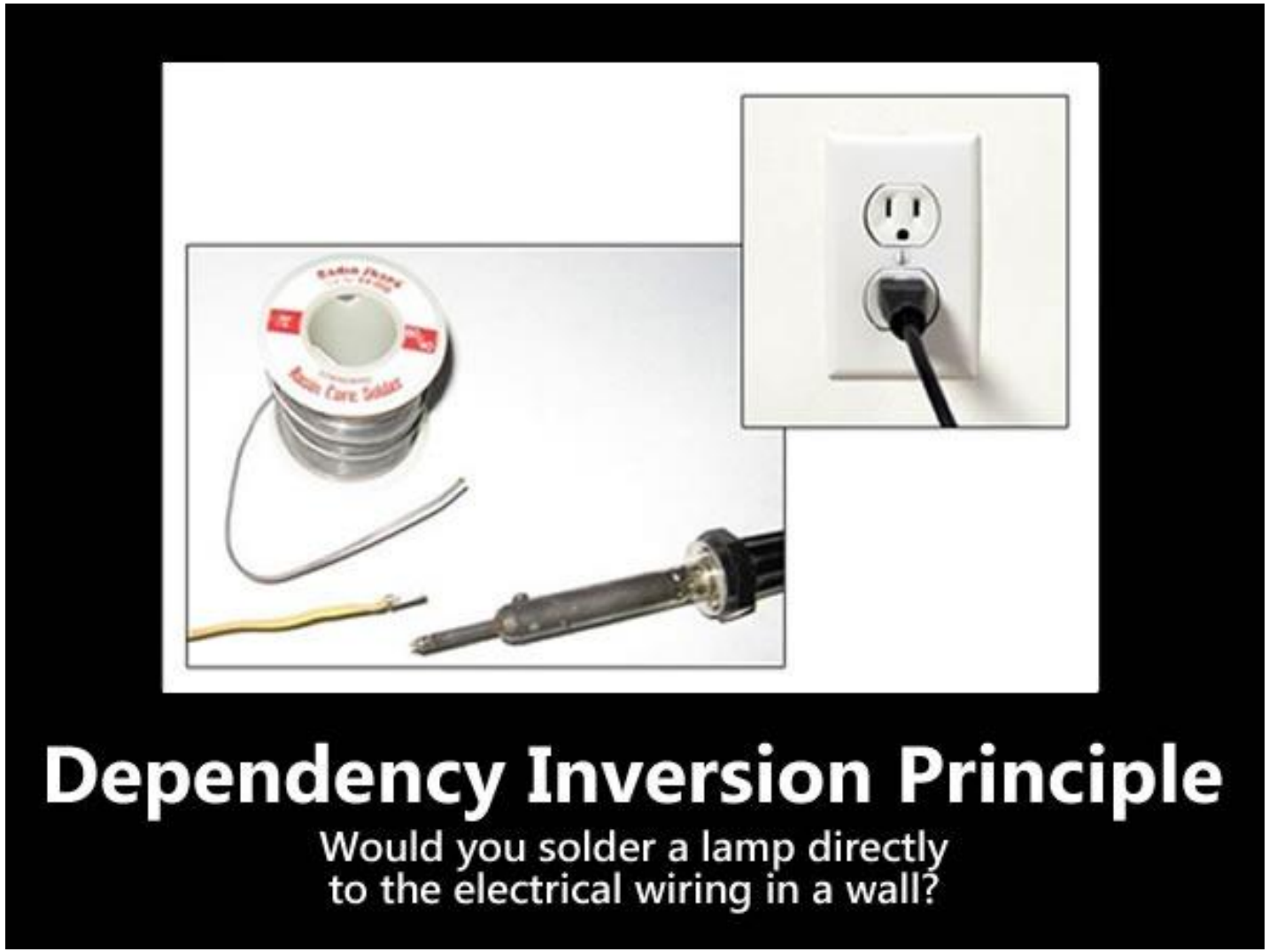
Interface Segregation Principle (ISP)

Interface Segregation Principle (ISP)

The interface for the component is kept simple and focused, avoiding the need for clients to depend on methods they don't use.

Example: The Pasto interface has a single getPrezzo method, avoiding unnecessary complexity.

Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP)

High-level modules depend on abstractions (interfaces) rather than concrete implementations, allowing decorators to be easily introduced.

Example: Products code depends on the Pasto interface, allowing the use of different decorators (e.g., ExtraCipollaDecorator, ExtraPiccanteDecorator) without changing the product code.

Advantages

- **Flexibility and Extensibility**: One of the key advantages of the decorator pattern is its ability to add new behaviors or modify existing ones without altering the underlying object's structure. This flexibility allows developers to extend functionality on-the-fly, making it easier to adapt to changing requirements.
- **Adherence to the Open/Closed Principle**: The decorator pattern adheres to the Open/Closed Principle, which states that software entities should be open for extension but closed for modification. By using decorators, we can introduce new features without modifying the original classes or interfaces.
- **Modular and Reusable**: Decorators are modular and can be reused across different components. This modularity promotes code reusability and reduces code duplication, leading to cleaner and more maintainable code.
- **Composable**: Decorators can be composed in various combinations to achieve different behaviors. This composability allows developers to mix and match decorators to create complex functionality without introducing code bloat.

Limitations

- **Increased Complexity**: While decorators provide flexibility, they can also introduce complexity, especially when many decorators are in the system. Developers need to be mindful of the order in which decorators are applied and how they interact with each other.
- **Overhead and Performance Impact**: Each decorator introduces an additional layer of indirection, which can lead to performance overhead. This impact may be negligible in most cases, but it's important to consider the potential performance implications in performance-critical scenarios.
- **Confusion with Inheritance**: The decorator pattern can sometimes be confused with inheritance, as both involve extending behavior. However, inheritance is a static relationship, while decoration is dynamic. Developers should carefully choose between the two based on the specific use case.
- **Limited Applicability**: The decorator pattern is not always the best solution for every scenario. For example, when behavior changes require modifications to the object's state or internal data, the decorator pattern may not be suitable.

Conclusion

Let's also let these **principles** and the **decorator pattern** help us on our journey as developers

But let's always remember to **use our heads** and not apply patterns and principles without thinking about it otherwise we could find ourselves with antipatterns or other code smells that we wanted to avoid.

Questions ?

- «Metto tutto ?»
- «Vuoi piccante ?»
- «Cibola vuoi?»

Thank you

Thanks to all the kebab shops who made my adolescence better