Structural Pattern: Bridge

Kevin Dockx
Architect

@KevinDockx | www.kevindockx.com

Coming Up

Describing the bridge pattern

Implementation:

- Restaurant cash register

Structure of the bridge pattern

Coming Up

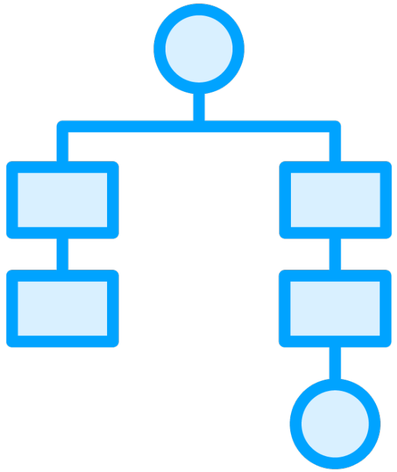
Use cases for this pattern

Pattern consequences

Related patterns

Describing the Bridge Pattern

Creational



Structural

Behavioral

Bridge

The intent of this pattern is to decouple an abstraction from its implementation so the two can vary independently

Describing the Bridge Pattern

Separate abstraction from implementation

- A means to replace an implementation with another implementation without modifying the abstraction

Describing the Bridge Pattern

Think of an “abstraction” as “a way to simplify something complex”

- Abstractions handle complexity by hiding the parts we don’t need to know about

```
public class VegetarianMenu {
     public int CalculatePrice() { … } }

public class MeatBasedMenu {
     public int CalculatePrice() { … } }
```

Describing the Bridge Pattern

```
public class VegetarianMenu {
     public int CalculatePrice() { … } }

public class MeatBasedMenu {
     public int CalculatePrice() { … } }
```

Describing the Bridge Pattern

```
public abstract class Menu {
     public abstract int CalculatePrice(); }

public class VegetarianMenu : Menu {
     public override int CalculatePrice() { … } }

public class MeatBasedMenu : Menu {
     public override int CalculatePrice() { … } }
```

Describing the Bridge Pattern

Abstract base class could also be an interface (e.g.: `IMenu`)

```
public class VegetarianMenu : Menu {
     public override int CalculatePrice() { … } }

public class VegetarianMenuWithOneEuroCoupon : VegetarianMenu { … }

public class VegetarianMenuWithTwoEuroCoupon : VegetarianMenu { … }

public class MeatBasedMenu : Menu {
     public override int CalculatePrice() { … } }

public class MeatBasedMenuWithOneEuroCoupon : MeatBasedMenu { … }

public class MeatBasedMenuWithTwoEuroCoupon : MeatBasedMenu { … }
```

Describing the Bridge Pattern

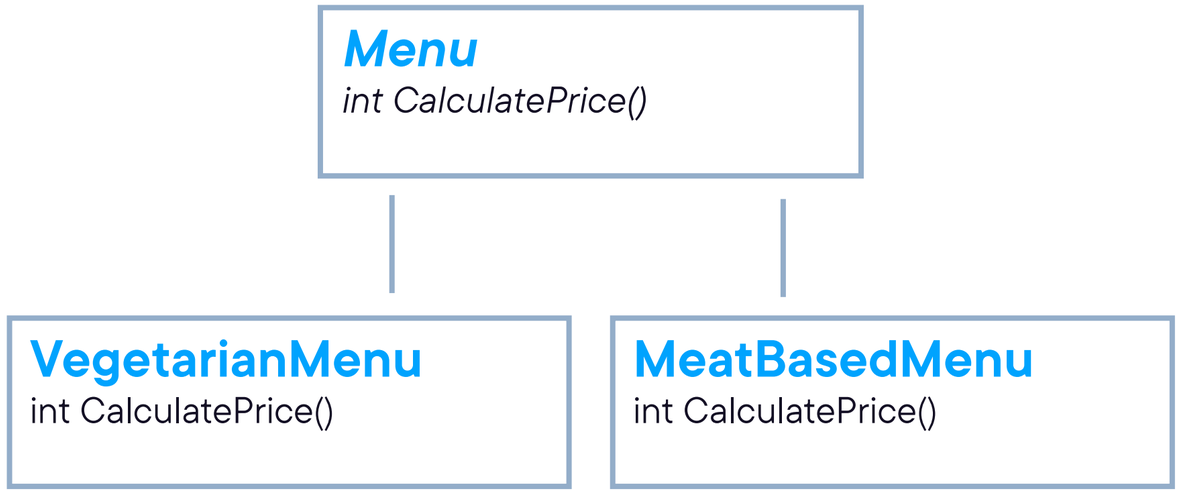
Subclassing is one way to add functionality
Subclassing tends to add avoidable complexity

Describing the Bridge Pattern

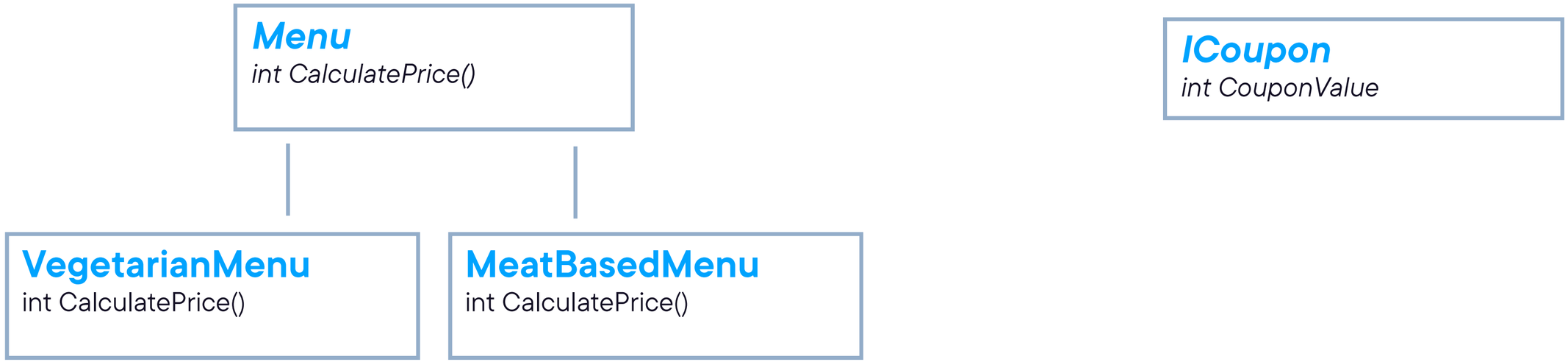
VegetarianMenu
int CalculatePrice()

MeatBasedMenu
int CalculatePrice()

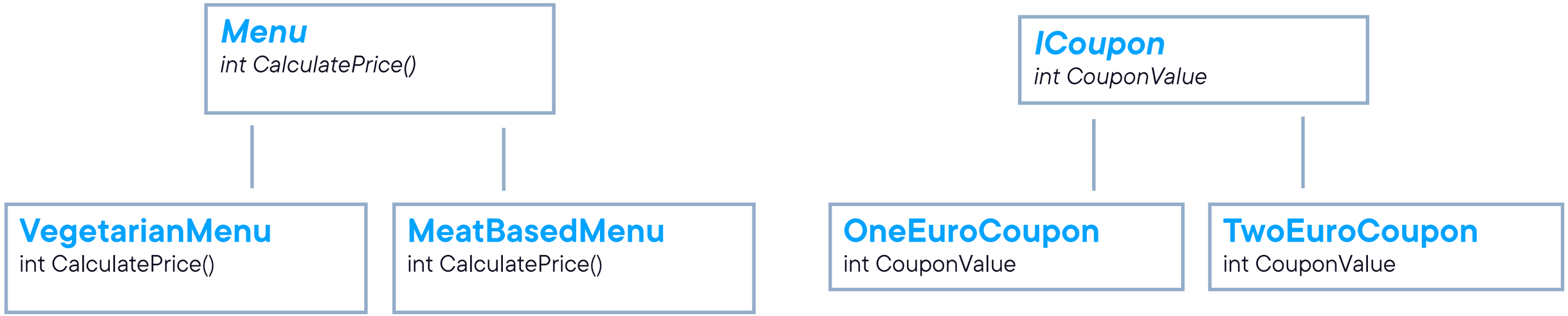
Describing the Bridge Pattern



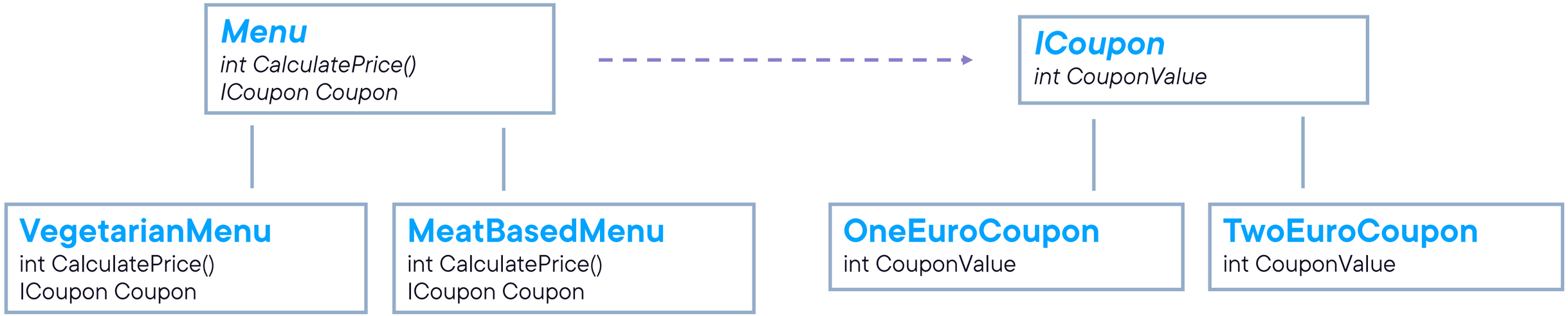
Describing the Bridge Pattern



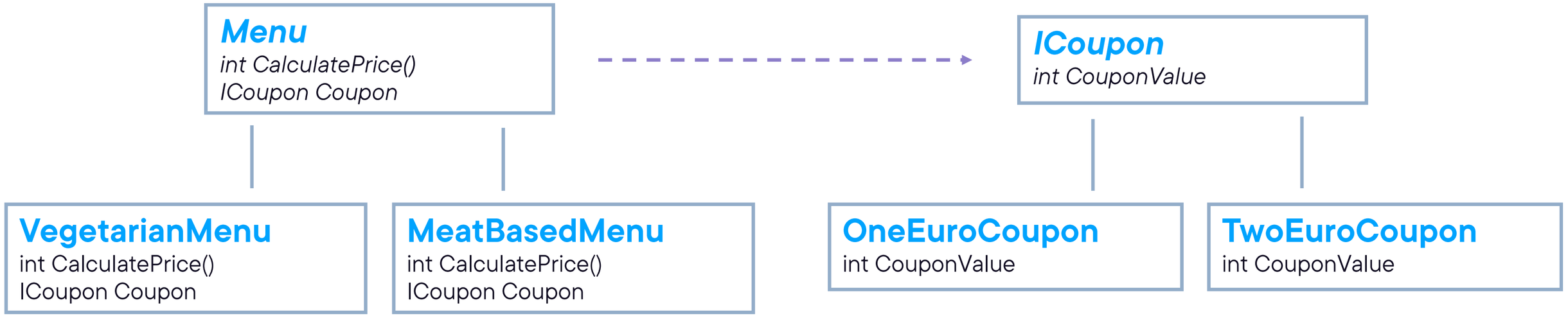
Describing the Bridge Pattern



Describing the Bridge Pattern

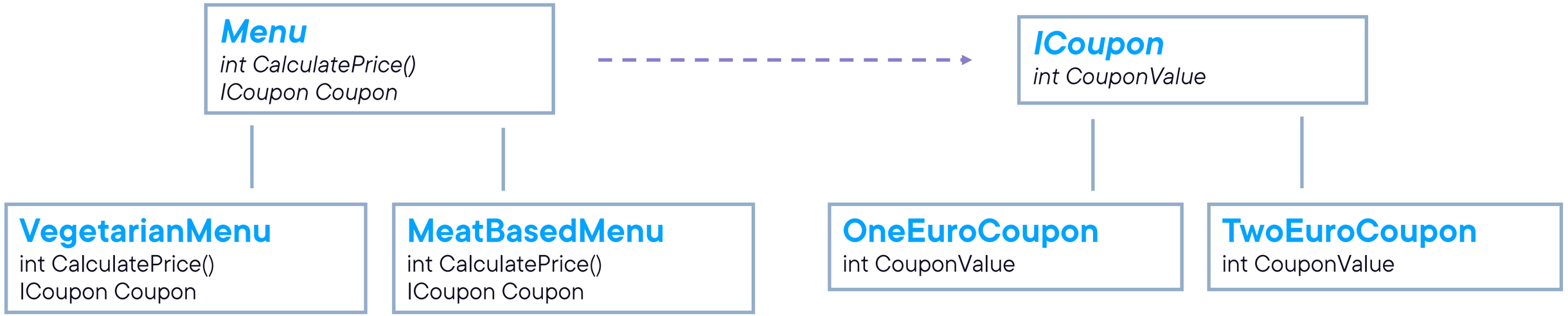


Structure of the Bridge Pattern

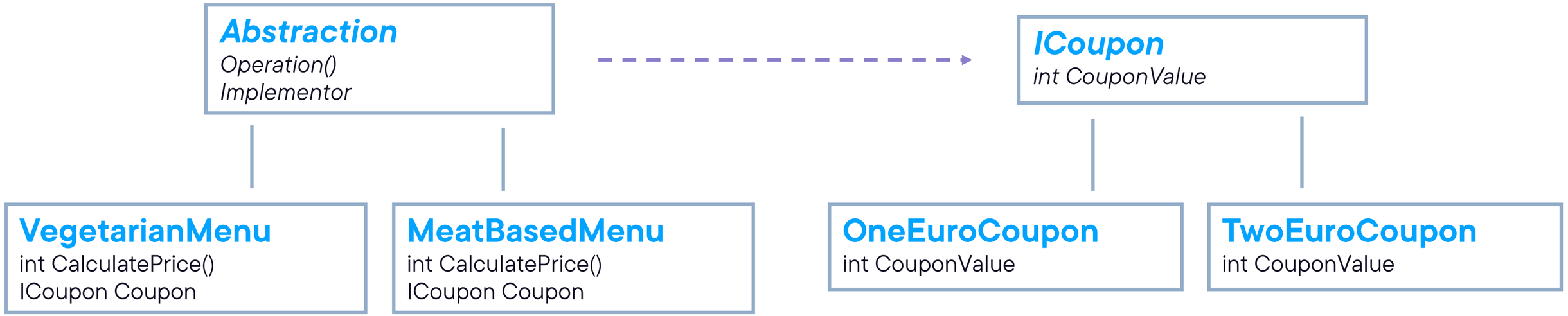


Abstraction **defines the abstraction's interface and holds a reference to the** Implementor

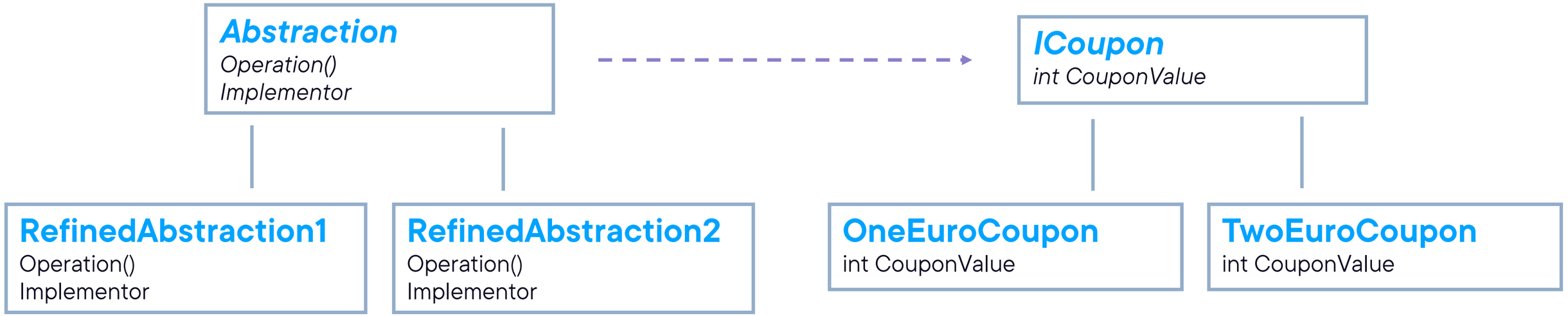
Structure of the Bridge Pattern



Structure of the Bridge Pattern

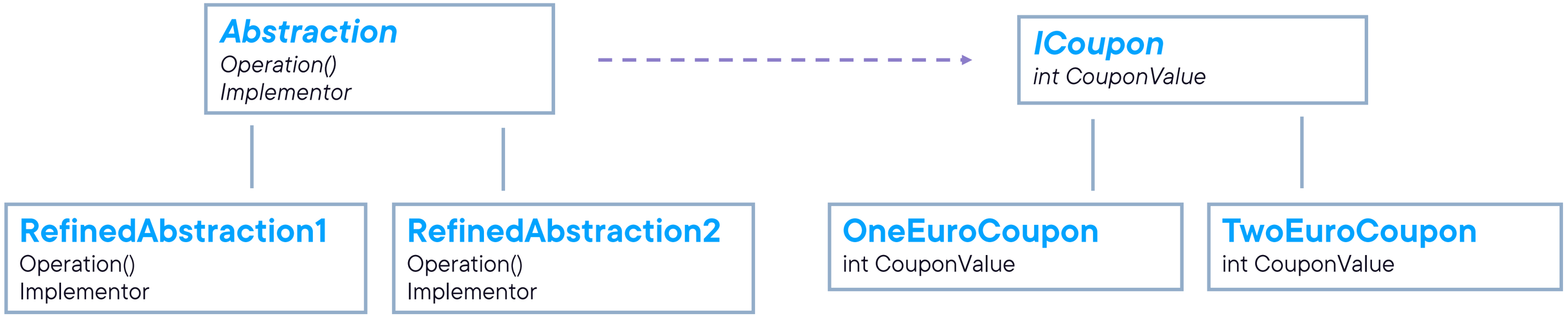


Structure of the Bridge Pattern

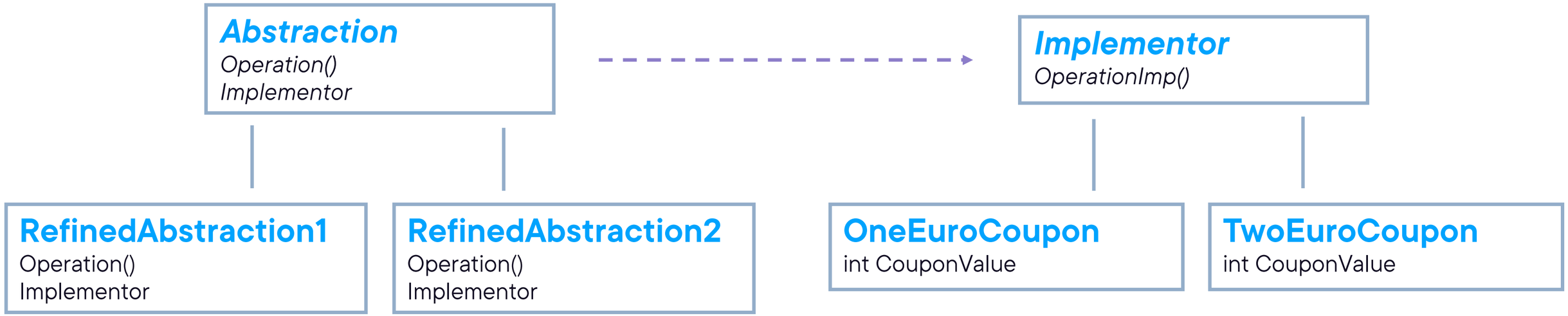


RefinedAbstraction **extends the interface defined by** Abstraction

Structure of the Bridge Pattern

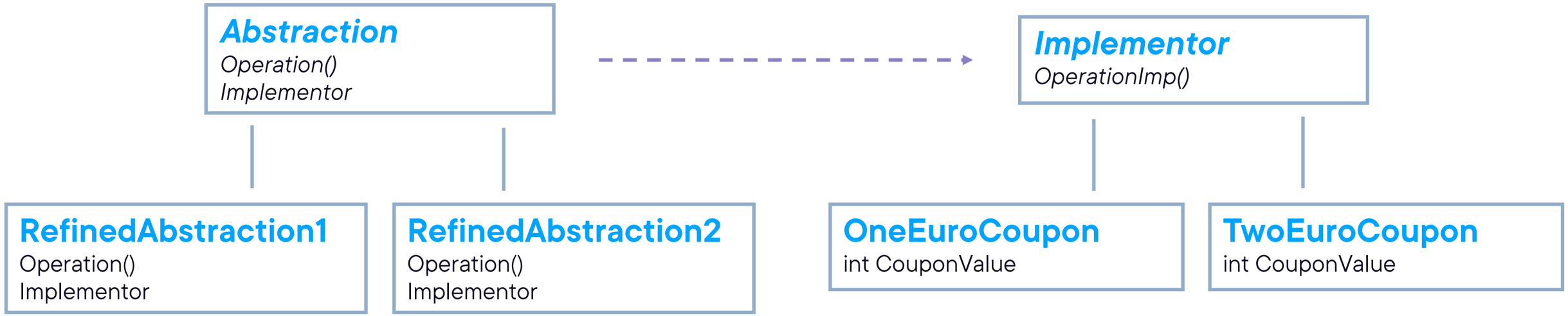


Structure of the Bridge Pattern

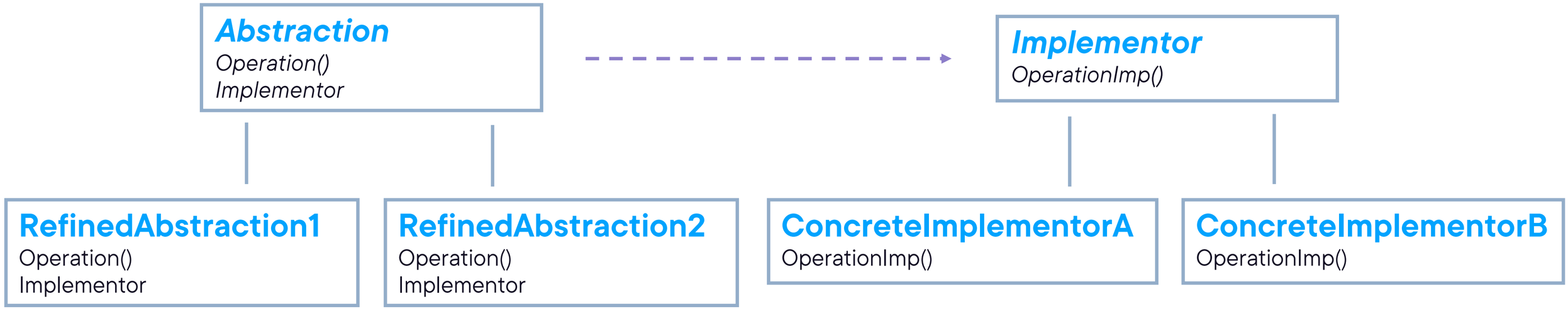


`Implementor` **defines the interface for implementation classes**

Structure of the Bridge Pattern

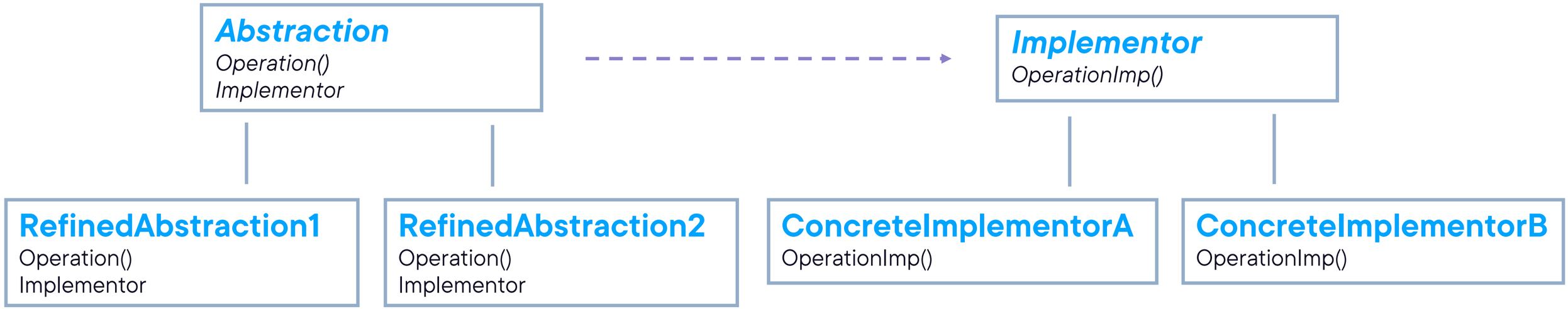


Structure of the Bridge Pattern



`ConcreteImplementor` **implements the** `Implementor` **interface and defines its concrete implementation**

Structure of the Bridge Pattern



Demo

Implementing the bridge pattern

Use Cases for the Bridge Pattern

When you want to avoid a permanent binding between an abstraction and its implementation (to enable switching implementations at runtime)

When abstraction and implementations should be extensible by subclassing

When you don’t want changes in the implementation of an abstraction have an impact on the client

Use Cases for the Bridge Pattern

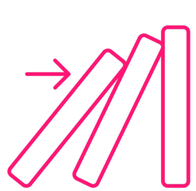
Separating notification mechanisms in a notification system

Separating streaming protocols in audio/video streaming

Separating UI components from rendering code

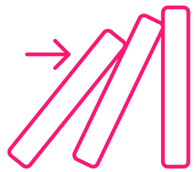
Separating functionalities from a specific device in universal remote-control applications

Pattern Consequences

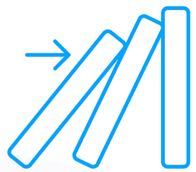


Decoupling: the implementation isn't permanently bound to the abstraction

As the abstraction and implementation hierarchies can evolve independently, new ones can be introduced as such: open/closed principle

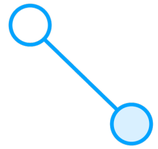


You can hide implementation details away from clients

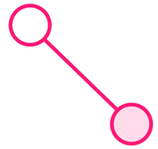


You can focus on high-level logic in the abstraction, and on the details in the implementation: single responsibility principle

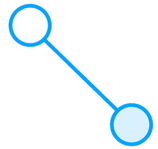
Related Patterns



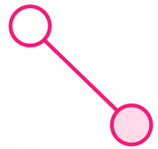
Abstract factory
Factory can create and configure a bridge



Adapter
Adapter lets unrelated classes work together, bridge lets abstractions and implementations vary independently



Strategy
Based on composition, like bridge



State
Based on composition, like bridge

Summary

Intent of the bridge pattern:

- Decouple an abstraction from its implementation so the two can vary independently

Summary

Main consequences:

- Decoupling
- Improved extensibility
- Hidden implementation details

Up Next:

Structural Pattern: Decorator